European UNIX® systems User Group

Autumn '90 Conference Proceedings

22-26 October 1990

at

Nice Acropolis

Nice

France

EUUG

European UNIX® systems User Group

Proceedings of the Autumn 1990 EUUG Conference

October 22–26, 1990
Nice Acropolis,
Nice, France

ISBN 0 9513181 8 7

Further copies of the proceedings may be obtained from:

EUUG Secretariat
Owles Hall
Buntingford
Herts
SG9 9PL
United Kingdom

These proceedings were typeset in Times Roman and Courier on a Linotronic L300 PostScript phototypesetter driven by `129.31.82.99`. PostScript was generated using **refer, tt, grap, pic, psfig, tbl, sed, eqn, troff, pm** and **psdit**. Laser printers were used for some figures.

ACKNOWLEDGEMENTS

Hosted by:	AFUU	
Proceedings Production:	Jan-Simon Pendry	IBM United Kingdom Ltd.
	Stuart McRobert	Imperial College, London, UK

These proceedings were specially produced for the EUUG at the Department of Computing, Imperial College, London, using resources generously provided by the Computing Support Group.

– jsp, sm.

UNIX Conferences in Europe 1977–1990

UKUUG/NLUUG meetings

1977 May	Glasgow University
1977 September	University of Salford
1978 January	Heriot Watt University, Edinburgh
1978 September	Essex University
1978 November	Dutch Meeting at Vrije University, Amsterdam
1979 March	University of Kent, Canterbury
1979 October	University of Newcastle
1980 March 24th	Vrije University, Amsterdam
1980 March 31st	Heriot Watt University, Edinburgh
1980 September	University College, London

EUUG Meetings

1981 April	CWI, Amsterdam, The Netherlands
1981 September	Nottingham University, UK
1982 April	CNAM, Paris, France
1982 September	University of Leeds, UK
1983 April	Wissenschaft Zentrum, Bonn, Germany
1983 September	Trinity College, Dublin, Eire
1984 April	University of Nijmegen, The Netherlands
1984 September	University of Cambridge, UK
1985 April	Palais des Congres, Paris, France
1985 September	Bella Center, Copenhagen, Denmark
1986 April	Centro Affari/Centro Congressi, Florence, Italy
1986 September	UMIST, Manchester, UK
1987 May	Helsinki/Stockholm, Finland/Sweden
1987 September	Trinity College, Dublin, Ireland
1988 April	Queen Elizabeth II Conference Centre, London, UK
1988 October	Hotel Estoril-Sol, Cascais, Portugal
1989 April	Palais des Congres, Brussels, Belgium
1989 September	Wirtschaftsuniversität, Vienna, Austria
1990 April	Sheraton Hotel, Munich, West Germany
1990 October	Nice Acropolis, Nice, France

Table of Contents

Author Index

Project Hygiene

Vic Stenning

Anshar Limited,
Camberley,
England.

ABSTRACT

It is suggested that many of the difficulties encountered by systems and software projects are not the result of deep technical problems, but rather arise from a lack of basic *project hygiene* – from failure to enforce various elementary principles and disciplines that are self-evidently prerequisite to a successful outcome. Such disciplines are primarily concerned with the control and co-ordination of both project *activities* and project *products*.

Examination of some of the elementary principles of project hygiene suggests that the principles are often violated. However, while the failings may be obvious, the means for overcoming such failings are somewhat less so. Suggestions are made both on methods and procedures that might improve the standard of hygiene within a typical project, and on topics that demand particular attention.

Of course, simply focusing on hygiene will not of itself ensure project success. The successful development of computer systems and software demands a co-ordinated process incorporating effective methods and supported by effective tools. However, basic hygiene must be at the very heart of the process, and is essential to creating the conditions under which the various methods and tools can usefully be deployed.

Ultimately, project hygiene is rather like any other form of hygiene: nothing spectacular comes from its presence, but the effects of its absence can be dramatic.

1. Introduction

Concern here is with basic *project hygiene* – certain fundamental and obvious principles that would seem to be a prerequisite to any complex cooperative production activity proceeding in an orderly and effective fashion.

Section 2 below briefly discusses some of the more obvious principles of project hygiene. No suggestion is made that the list of principles is exhaustive; it merely serves to illustrate the kind of "common sense" requirements that come under the heading of hygiene. However, despite these principles being obvious, it is suggested that many development projects fail to adhere to them. (The term development is used both here and elsewhere to encompass not only initial development of some system but also subsequent modification to correct errors or evolve the system's capabilities.) Almost inevitably, failure to enforce hygiene will result either in timescale over-run, or in poor quality products, or both. Not infrequently, lack of basic hygiene leads to complete disaster.

Having identified some basic requirements of project hygiene, section 3 then discusses various ways in which these requirements might be addressed. Unfortunately, while the requirements are obvious, the means of addressing them are somewhat less so, and there is considerable scope for differences of opinion on how the various requirements should best be tackled. However, the important point is that the requirements be recognised and that they be addressed by some means; the specific means employed are of somewhat secondary importance.

Finally, section 4 presents some broad conclusions.

2. Some basic principles

A few basic principles of project hygiene are discussed below. These principles are all very simple and very much "common sense". However, in the author's experience, many projects – including all that end up as major disasters – violate one or more of these obvious principles. Furthermore, while the principles may be self-evident, it is by no means always clear how these principles can be respected, particularly in the case of large projects that require the involvement of a substantial number of people.

Principle 1

Everybody involved in the project should know the objectives of what they are doing.

Or, more bluntly, everybody should know what he or she is trying to do.

This is the fundamental principle of project hygiene, and should not need stating. However, it is surprising how frequently this fundamental principle is violated, if not totally then partially.

Problems of people not knowing what they are trying to do can arise at different levels of the project. At the broadest level, there can be problems with "fuzzy" requirements and with the lack of proper system specifications. These are not in themselves fundamental problems, provided that everyone concerned with the project recognises that the requirements are fuzzy and accepts the implications. The problems arise when the requirements are ill-defined but people proceed as if they were well-defined.

At the intermediate level, teams who are given responsibility for individual sub-systems can misunderstand the role of that sub-system within the whole. It is not unknown for teams to be confused about the functional requirements on their sub-system, and hence to deliver something that simply doesn't do what was expected. However, more commonly the problems at this level relate to non-functional aspects: performance, robustness, fault tolerance, and so on. While there are notable exceptions, it is still generally the case that non-functional requirements on individual sub-systems are inadequately specified, if indeed they are specified at all. The result is individual sub-systems that are under-engineered or – almost as bad – over-engineered. One of the classic symptoms of poor project hygiene is for a team to spend weeks carefully crafting and optimising some component even though, in the context of the system as a whole, the performance of that component is of no real significance.

At the detailed level of the design and implementation of individual modules, problems arise both from communication being too casual and from communication being ambiguous. All too frequently when there are "integration" problems, the cause is not that the faulty component fails to do what its author intended, but rather that the author did not have a sufficiently precise understanding of what was needed. At integration time, comments such as "I thought it meant...." or "I thought that you were going to...." or "Nobody told me that...." are all too common.

These various examples merely serve to illustrate the general point that project personnel often do not fully understand the objectives of what they are doing, particularly in the context of the project as a whole. Given a woefully inadequate specification, which seems to be the norm, the natural tendency is to subconsciously complete the specification in a way that seems consistent and sensible, and then just get on with the job. Often this works. Sometimes it raises problems that have to be fixed. And occasionally it's disastrous.

Principle 2

Achievement of the overall project objectives should follow immediately from achievement of all individual objectives.

This is the necessary complement to principle 1.

Occasionally one encounters projects where everybody seems to know what they are doing, everybody seems to be doing a good job, and yet the project as a whole has major problems. While the causes of such problems can sometimes be very complex, two simple causes seem to recur.

First, there is often a tendency to pay more attention to the sub-division of work than to the subsequent re-combination of the results of that work. This is particularly the case when a project has a tight timescale and the complete team is assigned from day one. Under these circumstances, there is considerable pressure to rapidly partition the work so that each team member has something "constructive" to do. However, given insufficient analysis and design, and insufficient planning for integration and testing, this early partitioning can prove to be more destructive than constructive. The end result can be a "system" that is virtually impossible to integrate and literally impossible to test. More than one system has been completely redesigned during the supposed "integration" phase – another clear symptom of bad hygiene.

Second, in a large and complex project there is a tendency simply to "lose" things. Known requirements simply disappear. Assumptions that are implicit in the specification or design are subsequently overlooked, with the result that the system fails to operate as intended. Work done to explore possible design alternatives or to construct test scenarios is never documented and subsequently has to be repeated. And so on. Traceability is an essential aspect of project hygiene, but is all too easily lost unless effort is consciously invested in its preservation.

Principle 3

Both individual objectives and overall project objectives should be realistic.

People tend to be over-optimistic about what they are able to do and, most especially, about how quickly they are able to do it. Equally, people tend to be over-optimistic on other people's behalf. To a typical salesman, most technical jobs are trivial. And, for whatever reason, some senior managers repeatedly make commitments that range from the ambitious to the unachievable. They over-state what they can deliver, or how quickly they can deliver it, or both.

There is no single factor more damaging to project hygiene than unrealistic targets. By definition, a project with unreasonable objectives starts badly, but there is subsequently a very real danger of positive feedback, taking things from bad to worse. When trying to achieve the unachievable there is an obvious temptation to cut corners, to experiment with short cuts, and to avoid doing anything that does not address an immediate and urgent need. The end result is usually a complete loss of hygiene, leading to the project taking far longer than it would have done had it been properly planned and controlled, and delivering far worse products.

Of course, goals that were initially reasonable can either emerge as, or become, unrealistic as the project proceeds. Time can be lost to unforeseen problems or to changing requirements. Thus, basic hygiene demands regular monitoring of project status, not just in the narrow sense of progress against plan, but in the broader sense of whether the overall goals and strategy remain reasonable. Such a judgement cannot sensibly be made in isolation, considering only the ultimate goals, but rather must be based on a realistic assessment of the project's current status.

Principle 4

There should be a known method for addressing each individual objective.

Employing appropriate methods is an essential aspect of project hygiene – reliance on personal inspiration or hidden magic is unhygienic by definition.

Insistence on the use of known methods is not the same as insisting that we must know in advance a way of solving every unknown problem. When a new problem is encountered, we may well not know the method by which that problem should best be tackled. But under these circumstances we should know how we will explore the problem and determine the method by which the problem should be addressed – that is, we should have a method for selecting methods. In general, the methods that are employed at any step of the project may be governed by the results of previous steps, rather than prescribed in advance.

Thus, project hygiene does not demand the existence of some universal or infallible method, nor does it prohibit the use of experimental or exploratory methods. It simply requires that in addition to knowing what we are trying to do, we also know how we are trying to do it.

Principle 5

Changes should be controlled, visible and of known scope.

In software projects, change is the norm. Even in a project that is developing a completely new system, only a small proportion of the effort is devoted to original creation. The major part of the effort is concerned with modification of something that already exists, for example when performing a design iteration or when correcting program errors.

The fact that change is continuous makes it difficult to preserve consistency between the project's various products. The problem is exacerbated by the fact that frequently a large number of changes are in progress simultaneously. Changes arise from a number of different sources – changing requirements, detected errors, new design decisions, problem reports, and so on – and typically such changes cannot be handled serially. The interactions between the different changes, and the impact on the different variants of the system, then become extremely difficult to co-ordinate.

Inadequate change control is one of the most common forms of poor hygiene, and its symptoms are all too familiar. "Minor" changes have an impact that was totally unanticipated. The ramifications of a change are not followed through, so that modules affected by the change are not properly updated. Incompatible changes are introduced simultaneously. And so on.

Several of the fundamental hygiene requirements are obvious. For any proposed change, it should be possible to assess the impact of the change, and the possible scope of its effects, before the change is made. Where changes potentially "interfere", they should either be serialised or merged into one combined change that can be properly controlled. Those whose work will be impacted by the change should receive prior warning before the change is made, rather than simply seeing its results.

However, while such basic requirements are easily identified, they are unfortunately hard to satisfy.

Principle 6

Both people and products should be insulated from the effects of changes that are not (currently) of relevance.

In an environment of constant change, it is necessary for individual activities to be insulated from the impacts of changes until they are ready to receive them. While an individual activity is in progress, it should proceed in a stable context.

This is not to say that any extant activity must always proceed to completion, irrespective of any changes that arise while that activity is in progress. Obviously such a constraint would be both undesirable and impractical. However, where an extant activity is to be impacted by some change, this should be explicitly recognised and explicitly co-ordinated. The specification of what the activity is to do should be updated appropriately, and the implications of this update taken into account by the project planning, monitoring and co-ordination functions.

Lack of proper control in this area usually manifests itself in two ways. Firstly, both activities and products are forced to accept changes at inopportune times, simply because there is no mechanism whereby the acceptance of such changes can be postponed. Second, activities are continually subject to externally-imposed change, but all such changes are introduced implicitly, and the project co-ordination function treats the activities as if they are proceeding normally in a stable context. Both classic examples of poor hygiene.

3. Means of improvement

Section 2 above outlined a few of the basic requirements of project hygiene, and also indicated that projects often fail to meet these requirements. Unfortunately, identification of the obvious problems is rather easier than identification of solutions. However, without in any way claiming to offer such solutions, this section suggests some basic ways in which hygiene can be promoted and the hygiene level of the "typical" project improved.

Suggestion 1

Focus on the process as a whole, rather than on the (final) product.

Despite the considerable progress over the past several years, our industry still seems to over-emphasise source code and under-emphasise almost everything else. There still tends to be a suspicion that a project hasn't properly started until at least some code is actually running. We still tend to regard "software management" as being synonymous with "source management", when really the topic should be far richer.

Hygiene is not an attribute of an end product, but rather is an attribute of the process by which that product is produced. To improve project hygiene, we must focus on this process. Furthermore, we must avoid the temptation to concentrate on some particularly interesting aspect of that process at the expense of other, duller aspects: hygiene comes more from achieving overall co-ordination of the various process components than from improving individual components in isolation.

Four major aspects of the process that must be co-ordinated are project management, technical development, configuration management, and quality assurance. Frequently, these different aspects are treated as separable elements, and each is addressed largely in isolation of the others. As a direct consequence, the process as a whole lacks coherence. Managers have little visibility either of intermediate technical objectives or of technical progress, and therefore have no basis on which to manage the project effectively. The configuration management procedures are more symbolic than effective: they apply only to the code, hinder desirable changes, and fail to prevent undesirable changes or to promote proper change co-ordination. And, in the absence of effective project management and effective configuration management, the quality assurance function is severely impaired. In short, many people involved in the project will be unable to do their jobs effectively, and some will not even know what they are trying to do.

Emphasising the co-ordination of the many different aspects of the process is a good first step in promoting improved hygiene. However, it should be recognised that the source code provides a very poor basis for such process co-ordination. This leads immediately to the second suggestion.

Suggestion 2

Invest more effort in "higher level" descriptions.

Far too many project still rush into code. Any time spent on higher level descriptions of the system – requirements, specifications, designs – is spent grudgingly and sparingly. Sometimes such descriptions are not produced at all. Sometimes they are produced after the event, documenting the implementation to meet some contractual commitment, rather than as an integral part of orderly development.

Unfortunately, the distance between some vague notion of what we want and actual running code is enormous. If the entire project basically takes the form of a single gigantic leap from one to the other, it is difficult to imagine how any reasonable level of hygiene is ever to be achieved.

On any non-trivial project it seems essential to employ a sequence of higher-level descriptions to bridge the gulf between concepts and code in a series of manageable steps. The number and nature of these descriptions may vary dependent upon the application, but the sequence in some form will always exist. The technical objective of the project is not then to produce the final code, but rather to produce this complete harmonious sequence, together with related information on design decisions, verification exercises, and so on.

Of course, some forms of higher level descriptions are already employed fairly extensively: natural language specifications and data flow diagrams, for example. However, project technical personnel often take a rather cynical view of such descriptions. If the code is working, and the code doesn't correspond to the diagram, then it's the diagram that's wrong. When an update is required, the code gets changed but the diagram doesn't, leading eventually to the diagram becoming completely obsolete. The attitude is very much that it's the code that really matters, and the rest is so much arm waving.

At least part of the reason for the preoccupation with code is that the code has complete and precise semantics, which is in sharp contrast to most forms of higher level description in common use. There is enormous value in precise semantics: we can determine exactly what the system does in a given set of circumstances, with no doubt or ambiguity. Unfortunately, in the case of code, this precise semantics is at a level of detail which is very difficult for people to understand and for either people or tools to manipulate.

The failing lies neither with the code, nor with our desire for precise semantics, but rather with the imprecision of our higher level notations. This failing can be addressed in various ways. We can improve the precision with which we use our established notations, usually by being more precise in our use of natural language. We can switch to related notations with richer and more precise semantics. But ultimately, to completely overcome the failing, we need to adopt formal notations with semantics as precise as those of our programming languages.

Of course, there are many objections to the use of formal notations: they are very difficult and expensive to use, they don't ultimately ensure high quality results, they lack flexibility and generality, and so on. Some of these objections have some basis, others are groundless. What seems clear is that project hygiene benefits enormously from – and perhaps even demands – precise higher-level descriptions of the system under development. With informal notations, even vaguely approaching the level of precision required demands huge effort. Formal notations are the only known solution to the problem.

Of course, once the idea of formal notations has been accepted, there is scope for further improvement with the introduction of rigorous development and proof, typically on a selective basis. However, one step at a time: the initial requirement is to employ higher level descriptions and to make these descriptions as precise as is possible in the context of the particular project.

Suggestion 3

Co-ordinate activities as well as products.

As has already been discussed, there has been a traditional tendency in the software industry to focus more on products than on process, and particularly on the program code.

Presumably as a consequence of this tendency, most of the co-ordination mechanisms that have been developed have been concerned with the co-ordination of products. We have change control mechanisms, general configuration control mechanisms, interface checking mechanisms, and so on. But there is a noticeable lack of mechanisms for co-ordinating project activities: the few mechanisms that are available seem to be adjuncts to product co-ordination systems, rather than specifically directed at the co-ordination of activities.

There is an obvious duality between activities and (intermediate and final) products. One can either view a project as consisting of a product structure that is manipulated by activities, or one can view it as consisting of a set of activities that "produce" and "consume" products. Ideally, consistency between the two views should be enforced, and it should be possible to switch between the two as appropriate. Certainly the typical current situation – where the activity view is the province of the project management function and the product view is the province of the configuration management function, and never the two shall meet – is highly undesirable.

Recognition of the activity/product duality suggests the use of a common model encompassing both. The individual views can then be obtained by appropriate projections from this common model. Such an arrangement would promote basic hygiene, in that a minimum level of consistency would be guaranteed. Potentially, such a common model could also provide a foundation for configuration management, as discussed under "suggestion 5" below.

Suggestion 4

Adopt a strict policy on project phases.

Section 2 discussed the problem of project hygiene being completely destroyed by unrealistic objectives. Unfortunately, it is not always possible at the outset of a project to judge the realism of the objectives, simply because there is insufficient information on which to base such a judgement.

Under these circumstances, where the full scope and difficulty of the project are unknown, hygiene can be promoted by sub-dividing the project into self-contained phases – so self-contained that each phase is, in effect, a separate project. While the overall objectives and projected timescale may be known in outline, only the objectives and timescale for the current phase will be defined in detail. The objectives and timescales for the next phase will then be defined towards the end of the current phase. And, dependent upon knowledge gained during the current phase, the overall objectives and projected timescale can always be modified – perhaps drastically.

Each phase will typically produce the next description in the sequence of higher level descriptions discussed earlier. Each phase will identify new problems and generate new knowledge, and thus will provide the basis for a proper definition of the next phase. Hygiene can be preserved, because the project team never needs to commit to solving an unknown problem.

Of course, such an approach requires not just the support of the project team, but also the co-operation of the "customers". While the latter may initially be reluctant to accept what might be seen as an open-ended commitment, they can usually be persuaded that the approach is in their own best interest – which indeed it is.

Suggestion 5

Provide more semantic information to the configuration management and build systems.

As mentioned in section 2, the co-ordination of continuous and concurrent changes is one of the most difficult aspects of project hygiene. While many mechanisms and tools have been developed to address various aspects of this problem, and many of these have been extremely valuable, there are still no complete solutions available. Indeed, given the nature of the problems, complete solutions are difficult even to imagine: one can only hope for steady improvement, both in terms of the range of problems that are addressed and in the effectiveness with which they are addressed.

Of necessity, in order to provide general and accessible solutions to urgent problems, most of the available change control mechanisms are based upon very simple models. The fact that something has changed may be measured, and the direct impact of that change assessed, by the use of date/time stamps. Concurrent changes to individual modules are restricted by the use of check-out/check-in mechanisms. And so on.

Clearly such mechanisms perform a useful function, but equally clearly they address only part of the problem. Assessing changes on the basis of date/time stamps can become extremely inconvenient when one wishes to make a limited change to some previous release of the system. Check-out mechanisms alone do not address the issues of maintaining a consistent set of modules when a change to one module may impact several others.

Over the past several years, there has been considerable progress in the field of change management – in the context of such languages as Mesa, Modula and Ada – from exploiting the syntactic structure of inter-component interfaces. Several of the more trivial, and more common, problems of change can be eradicated in an environment where syntactic interface checking is enforced.

There is considerable scope for further improvement by extending the checking to encompass not just syntax, but also semantics. Such extension of course requires precise formal specification of the kind discussed earlier. By performing semantic checks it would be possible to ensure that a component module was functionally compatible with the remainder of the system or, in the case where it is not, to indicate the other components that are impacted.

Like other aspects of the overall process discussed earlier, the field of change control and configuration management has suffered from an undue preoccupation with source code. Fortunately, there is now widespread recognition that configuration management of higher level descriptions, of user documentation, of test histories, and so on, is just as important as configuration management of code.

Eventually one would like to see complete project/product graphs that encompass all the intermediate and final products with which the project is concerned, their components, all the versions and variants, and the semantic relationships between them. Such a graph would obviously be both large and complex, and would require some form of automated consistency preservation system. Some of the relationships would initially be asserted by members of the project team, and as products begin to emerge they would be checked for consistency with those assertions. Other relationships would be derived from the products themselves. The impact of proposed or actual changes, to either a higher level description or a code module, would be assessed both syntactically and semantically. The mechanisms would be able to report the impact of the change on all products of interest, including "old" variants.

Such a system, which could make a huge contribution to this very important aspect of project hygiene, may still be some way off. But steady progress is being made in this general direction, and we have grounds for optimism.

4. Final remarks

One of the reasons for poor hygiene being so prevalent is that people tend to become so engrossed in the technical demands and details of what they are doing that they lose sight of the broader picture. Not infrequently, opportunities for improving basic hygiene can be identified simply by standing back from the day-to-day business of the project and asking a few fundamental questions of the kind suggested above; for example

- are the objectives of all project activities clearly and precisely defined?
- are there any activities for which the objectives are completely unrealistic?
- how will the results of the individual activities be combined to meet the objectives of the overall project?
- are there explicit procedures and mechanisms to ensure traceability?
- are there effective mechanisms for co-ordinating changes and limiting the scope of their impact?

Sometimes, when the answers to such questions are not entirely satisfactory, one can readily identify obvious minor enhancements to the process being employed that will yield some rapid improvement. Often, the purpose of such enhancements is to make explicit some aspect of the overall process that was previously handled implicitly. For example, it may be appropriate to require explicit specification of objectives for all individual project activities; or to introduce a new procedure for authorising changes to some level of description; or to insist that specifications encompass non-functional characteristics as well as functional ones; or to instigate a formal "internal" release mechanism to complement the external one. Anything that will promote awareness of the real objectives, stability in the face of change, and realistic assessment of the true position, can only enhance hygiene. Such modest changes may not entirely solve the perceived problems, but they can yield immediate benefit.

Considerable progress can often be made by insisting that what was previously implicit and assumed be made explicit, by introducing basic procedures for co-ordination and communication where previously none existed, and by extending the scope of established QA review procedures. All this can be done at the "process glue" level, without changing the individual technical and management methods that are employed within the process.

However, regardless of the position from which one starts, and regardless of the route by which progress is made, the need will eventually emerge to make higher-level descriptions more precise. At some stage, further progress with project hygiene will require the introduction of formal specification and design notations with precise semantics.

Probably the most difficult area of project hygiene is the co-ordination and control of the many versions of the various intermediate and final products, the components of those products and their relationships. While much valuable progress has been made with mechanisms to address various individual aspects of this huge problem, there has been little sign of a coherent mechanism to address the problem in its entirety. The "project/product graphs" envisaged in section 4 above could offer one route to a solution, though perhaps not the only one or the best one. Of all the research topics that could usefully be addressed under the broad heading of project hygiene, this is probably the most urgent and the most important.

Washing Behind Your Ears:

Principles of Software Hygiene

David M. Tilbrook
John McMullen

Nixdorf Computer Canada Ltd.
Suite 1800
2235 Sheppard Ave. East
Willowdale, Ontario, Canada

ABSTRACT

This paper presents a discussion of the objectives of and impediments to software hygiene, and a list of suggestions to be followed. The suggestions are aimed at mid-sized projects, although we believe the basic ideas are applicable to all sizes of efforts.

This paper is prepared to accompany Vic Stenning's "Project Hygiene". We borrow heavily from that paper with respect to the basic theme of achieving better project quality through the application of some simple principles. We apply Stenning's Principles and Suggestions to source management, as well as developing our own disciplines of software hygiene.

1. Introduction

This paper is concerned with software hygiene, which we feel is similar to personal hygiene, where software hygiene is the science concerned with maintaining healthy software. Like personal hygiene, software hygiene is most conspicuous in its absence. We do not claim that good software hygiene will help you win friends and influence people, but poor hygiene will certainly cause you to lose clients and discourage users.

Like personal hygiene, software hygiene is largely a matter of simple practices, the programming equivalent of washing behind your ears and flossing your teeth regularly. In this paper, we're going to discuss some of those practices, and (like your mother once did) try to point out some of the dire results of poor hygiene.

Our examples and our **principles** of software hygiene concentrate on medium-scale projects involving about a dozen programmers and a million lines of code, as most of our experience is in that range. However, these principles can be scaled up and down to other projects. Some examples are taken from the authors' current project, which does follow these principles. (Specific details can be found in the Appendix.)

The title of this paper was inspired by Vic Stenning's paper "Project Hygiene" [Ste89a], which was to have been presented at the Software Management Workshop, April 1989 in New Orleans. Unfortunately, Vic was unable to attend, but, fortunately, both his paper and this one are being given as joint keynotes at this conference.

Stenning's paper addressed the problems of project control and management, whereas we concentrate on the control, use, and management of project source. Despite this difference, Stenning's abstract is directly applicable.

> "It is suggested that many of the difficulties encountered by systems and software projects are not the result of deep technical problems, but rather arise from a lack of basic project hygiene – from failure to enforce various elementary principles and disciplines that are self-evidently prerequisite to a successful outcome. Such disciplines are primarily concerned with the control and co-ordination of both project *activities* and project *products*."

We will focus primarily on those activities concerned with the development, protection, application, and distribution of project product in source form. However, the disciplines we outline can be considered to be special cases of those that Stenning outlines. Stenning continues with:

> "Examination of some of the elementary principles of project hygiene suggest that the principles are often violated. However, while the failings may be obvious, the means for overcoming such failings are somewhat less so. Suggestions are made both on methods and procedures that might improve the standard of hygiene with a typical project, and on topics that demand particular attention."

We apply Stenning's Principles and Suggestions to the realm of source management, evolving our own disciplines of software hygiene.

> "Of course, simply focusing on hygiene will not of itself ensure project success. The successful development of computer systems and software demands a co-ordinated process incorporating effective methods and supported by effective tools. However, basic hygiene must be at the very heart of the process, and is essential to creating the conditions under which the various methods and tools can usefully be deployed."

We will discuss some characteristics of the methods and tools that we think are necessary.

> "Ultimately, project hygiene is rather like any other form of hygiene: nothing spectacular comes from its presence, but the effects of its absence can be dramatic."

Unfortunately in the UNIX world, demonstrations of the last statement are far too common. Many software projects are bogged down in a morass of internal inconsistencies, poor documentation, versions which cannot be reproduced for testing, and frantic periods of retrofitting when it turns out that the finished product, as a whole, can not be constructed, either at the development site or (worse) at a client site.

We hold (and hope) that our software hygiene principles will help.

2. Characteristics of Software Hygiene

The characteristics of good software hygiene are relatively simple.

- **Cleanliness:** the ability to produce the product, the whole product, and nothing but the product. The procedure used to deterministically reproduce the product, in either its current form or as it existed at any time in the past, should be consistent and easily performed at short notice.
- **Consistency:** the product must be consistent with respect to its source, its installation, and the tests performed upon it. When an instance of the product is created, its behaviour must truly reflect any other instance of the product produced from the same source, modulo environmental differences (e.g., host machine and/or operating system). When the source is changed, the product is changed appropriately. The reconstructed product, incorporating the changed source, must be equivalent to the product that would be produced by a complete construction from source (a construction from the bare metal, so to speak). Further, any testing performed on the product must be relevant to the product as delivered. In other words: what has been tested is what is delivered, and what is delivered, has been tested.
- **Adaptable:** the facility with which one can transform the product to adapt to changing requirements imposed by the user and/or target environments, while preserving and protecting the consistency of the product.
- **Universally Helpful:** support is provided for all activities concerning the project not just those conventionally associated with source, construction, and configuration management.

A sure sign that some degree of good software hygiene has been achieved is **confidence**; confidence in the validity of the testing that has been performed, in the ease of correct installation, and in the consistency of the product with respect to its source. When the product is produced, one should be confident that it is what was supposed to be produced.

The above characteristics do not directly address the issues of product *correctness*. While our ultimate objective is correct software, we are actually concentrating on the consistency of the product, be it correct or incorrect. We want to ensure that problems with the product are not due to errors in the construction and/or installation process. If errors occur we want to be confident that they are due to: 1) errors in source that we can deterministically retrieve; or 2) differences between the environment on which the problem was discovered and the environment on which the product was tested and/or created. Furthermore, we want, as much as possible to be confident that if the problem is not reproducible from our source (i.e., case #1), then it is due to a misunderstood feature of the target environment, or a problem on that environment that is not part of our product.[1]

Another reason that we appear to de-emphasize system correctness, is that we firmly believe that correctness is only possible if our listed objectives are met. Furthermore, meeting our objectives will promote and facilitate achieving system correctness.

But what is "system correctness," and what other software qualities must be promoted?

In the first chapter of his book *Object-oriented Software Construction* [Mey88a], Bertrand Meyer describes ten basic **external** quality factors, where "external" is defined as being detectable by users of the product. These factors are: correctness, robustness, extendibility, reusability, compatibility, efficiency, portability, verifiability, integrity, and ease of use.

This list represents a good criteria for software quality, and we claim that software hygiene comes from a concern for software quality. Consequently we consider some of these criteria from the viewpoint of software hygiene.

Extendibility Any software discipline must handle with ease the modification of existing source and the addition or removal of source. The software process used to develop, test, construct, deliver and maintain the source must be capable of producing the product as represented by the selected version of the source, no matter what modifications or changes have been made since its last production.

Reusability The software process must promote and facilitate the sharing (i.e., reusability) of software components.

Compatibility Our focus is on medium scale software projects. Typically such projects involve a great deal of integration, but their size cannot guarantee the return on investment required to employ large scale integration systems (e.g., databases, integration teams). Therefore, the development and construction must facilitate integration testing in a straight-forward and universal way, without requiring full-scale construction and testing before source can be added to the true product baseline. In other words, we want to use a cost effective way of doing integration testing.

Portability Portability is a very difficult quality to achieve. The adjective "portable" is often abused. "There is no such thing as portable software; there's only software that has been ported" [She89a]. Our experience is that writing truly portable software is impossible. All one can hope for is that the strategy one employs to cope with system differences can be quickly adapted to cope with and recover from the subtle differences that inevitably cause the system to fail.

Verifiability We are trying to ensure the validity of any verification (i.e., testing) done before the source is added to the baseline, or before the product is released. As we have said before, we want to ensure that is delivered is what was tested. However, like Meyer we want to ensure that the product can be tested once installed and that the tests are meaningful and reproducible.

To sum up our objectives, we use the term "rolling release engineering". This is not to imply "release du jour", rather we mean that at any time, the product as it would be produced from the source, can be produced, is close to releasable quality, and, for the most part, tested.

Certainly there are many ways to achieve these objectives: Stenning concentrates on project management, with his major emphasis on project objectives and the software process. Meyer emphasizes the use of an object-oriented language (Eiffel) to avoid the pitfalls of conventional approaches to software construction. While not dismissing those issues, we are concentrating almost exclusively on activities directly relating to the maintenance and processing of source.

3. Software Maintenance – The Field of Conflict

A justification for our emphasis may be derived from the following observations from [Mey88a].[2]

> "Often, discussions of software and software quality consider only the development phase. But the real picture is wider. The hidden part, the side of the profession which is not usually highlighted in programming courses, is maintenance. It is widely estimated that 70% of the

1 It is often amusing when we claim that the problem with our product is due to a bug on their (i.e., the users') system. This is usually met with justifiable scepticism, but accepted (albeit begrudgingly) when we demonstrate that the problem does not exist on the other N (where $N > 3$) machines on which the system was constructed from identical sources using identical processes.

2 The authors wish to express their appreciation to Dr. Meyer for his permission to use these extracts.

Reason for Change	% of cost
Changes in User Requirements	41.8
Changes in Data Formats	17.4
Emergency fixes	12.4
Routine Debugging	9.0
Hardware Changes	6.2
Documentation	5.5
Efficiency Improvements	4.0
Other	3.4

Table 1: *Breakdown of Maintenance Costs [Lientz 1979]*

cost of software is devoted to maintenance. No discussion of software quality can be satisfactory if it neglects this aspect."

The 70% figure is based on traditional data processing projects, an area very foreign to the UNIX community. It seems likely that the percentage is far higher for the UNIX community; however, no studies have been done. Given the lack of any studies, we are forced to use studies from the D.P. developers. In a survey of 487 installations [Mey88a], *after* [Lie79a], maintenance costs were broken down as shown in Table 1.

Table 1 indicates that changes constitute approximately two-thirds of the cost of maintenance. Therefore, changes represent 46% (i.e., 66% of 70%) of the cost of software. Consequently, any improvement in the process of managing changes to a product is therefore addressing close to half of the cost of software.

4. Impediments to Software Hygiene

Impediments to software hygiene can be roughly divided into three categories: those imposed by the system suppliers, by the programmers, and by the users and/or management. We had originally planned to use this section to catalogue and illustrate the multitude of sins committed by these three groups to justify some of our principles. It is very tempting to use this opportunity to subject some of the suppliers to the same kind of pain and agony that they inflict on us. But to do so would far exceed the time and space allotted to this paper, and possibly involve talking to lawyers,[3] so, we resist temptation, and limit ourselves to a short list of requests to these three groups.

To the suppliers:

- Please ensure that your documentation is read by someone other than the author before it is distributed. This might catch some of the more blatant problems, such as the directive, "run the following four commands" preceding a list of five.
- Please read your documentation in the form that you distribute it, before you distribute it. For example, far too often 0's appear instead of "\n".
- Please don't distribute untested documentation for key elements. Many times it is cheaper for us to determine how your system is used through experimentation, than to recover from your outright mistruths. For example, don't tell us *ranlib* is required if the very opposite is true (e.g., *ranlib* does not work).
- Please ensure that **your** compilers are able to compile **your** header files.
- Please ensure that the key system development tools (*cc*, *ld*, *as*, *ar*) fail properly – report the problem and abort – when they run out of resources (e.g., disk, memory). Dumping core is not failing properly.
- Please ensure that warnings are either valid or individually suppressible. Our product construction runs processes in over 3,000 directories across six environments and produces at least 30,000 lines of diagnostics. It is extremely irritating to have to check each warning to ensure it's not just another one of **your** bugs.
- Please ensure that compiling **your** code does not raise warnings. For example, if **your** compiler reports unused variables or unreachable code, ensure that **your** *yacc* does not generate such situations.

3 Or, worse yet, listening to them.

- Please ensure that **your** semantics are consistent within **your** own environment. We don't really care if you conform to one of the standards, but we care a great deal about consistent behaviour within a single environment.
- Please ensure that **your** *lint* libraries describe **your** libraries. We suggest you run *lint* against the sources for your libraries and their lint libraries. This is also a way for you to test *lint* itself, so that the first time we run it, we don't get a core dump.[4]
- Please ensure that **your** *lint* is checking the same interpretation of the language definition that **your** compiler compiles.
- Please fix your C preprocessor. Whereas most of the above are requested of only some of you, this one is demanded of **all** of you, because you all have serious bugs in your *cpp*. While you are at it, why don't you document it?

Let us assure the reader that despite appearances, the above is a very small subset of the problems. It has been limited to problems that are shared by multiple suppliers, are particularly offensive, or are serious impediments to software hygiene.

To the programmer, please realize that:

- Your responsibility is to deliver source to someone else in such a way that they can build and use the product. The fact that "it works for you" is irrelevant.
- If someone else has a problem with your source, then you have a problem. He or she should help to uncover the cause of the problem – it may be due to the environment or methods – but ultimately, the responsibility for rectifying it is yours, even though the original problem is not.
- You are a member of a team, and that calls for a certain minimum amount of co-operation. Certain of the coding practices you were allowed in your previous environment might be considered unsafe, unhygienic, nonsensible, or otherwise unacceptable. Please show some consideration for your fellow team-members, and refrain from such practices.
- We (the QA group) feel a lot more confident about a series of small changes than one big one. It may seem that one big change involves less work, but that is rarely reality.
- Management's prime responsibility is to help you do your job and to help you solve your problems, but they cannot do that if you do not tell them about problems.

We may view the programmer as an impediment, but we actually believe that the majority intend to meet their responsibilities to the best of their ability. However, many are not suitably trained when hired. For most programmers, their first real experience working in a group is after they leave university. At far too many universities, Software Engineering courses are optional, very poorly taught,[5] or not offered.

Finally, to the users/managers:

- Managers will please re-read the last request to the programmers.
- Users/managers will please realize that software hygiene is a high priority and requires a suitable investment. It is not something that can be performed by a junior programmer. Furthermore, short-cuts demanded to meet short-term needs will result in long-term failures.
- Users/managers will please realize that a change in the requirements and/or priorities, requires a change in the schedules (e.g., deadlines may have to be set back accordingly).
- Users/managers will please realize that unplanned events that require any effort on the part of the developers (such as demonstrations to potential clients or visiting firemen) are effectively changes to the priorities (see previous request).
- Users/managers will please realize that programmers are (with a few exceptions) human, and as such, need appreciation and recognition for their efforts.

4 Assuming that you act on your bug reports.

5 In one introductory S.E. class, the professor announced that he had written more code than they (the students) would ever see. He did not announce that he also had a well-deserved reputation of never having delivered a working system. At the end of the term it was amusing to see his students desperately trying to integrate their systems for the first time, the night before they were due – a situation that this course should have been teaching them to avoid.

5. Our Credentials

A brief evaluation of the success of our approach is in order. Details of some of our software process are described in the appendix.

Our group currently consists of a project manager, a QA manager, 10 developers, and one (lonely) technical writer. Our primary project is the EMS system, which is a toolkit for building document imaging systems based on UNIX, X11, and Motif. Our principle objective is to deliver that system to our German partners (Sietec in Berlin and Nixdorf in Paderborn) who will then build client-specific applications. The system is primarily targeted to Targons (Nixdorf's RISC archictecture machines), Apollos, Mips, 386s running ODT, and Siemens' MX300s and WX200s.

The deliverables consist of 19 object libraries (containing approximately 400 modules), 250 header files, 80 application programs, and a testing suite of 50 programs. The total size of the deliverable system is approximately 70 Mega-bytes, consisting of 1240 files, 700 of which are documentation related.

The second major project is the D-tree, parts of which are used to build and maintain the EMS system.[6] The full D-tree consists of 14 libraries (containing 200 modules) and 250 programs. The full D-tree deliverables consist of approximately 50 Mega-bytes, consisting of 1800 files.

Combining the EMS and Dtree systems, there are 3,500 source files containing approximately 600,000 source lines, and 13 Mega-bytes spread across 300 directories. For comparison, X11R3 is also 13 Mega-bytes, in only 1500 files.

Our approach seems to work. Both the EMS and D-tree systems are maintained simultaneously on six environments from identical source, modulo a single construction parameterization file, yet the volume of changes to that source is huge (40,000 deltas in the last 14 months). Construction failures due to source errors are few, despite the fact that programmers are not required to test on all six environments. We have ported the EMS system to a variety of machines, with few problems, usually in hours. The problems that we have encountered are usually due to problems in the host environment (non-working *rename*(2), buggy C compilers). Most recently we ported the D-tree and EMS systems from source to a new machine, a new operating system, and a new interpretation of the C language definition in about 15 hours (despite a painfully slow disk). This exercise revealed no problems in our system other than a number of unused static declarations (the compiler aborted when a static function was declared but not defined) and the *#define* of the C keyword "far". However, we did find a number of problems in the target systems: a *cpp* bug (a frequently encountered problem); libraries built with header files that were not the ones on the system; missing libraries; missing header files; disagreement between the documentation and the implementation of libc routines; system header file *#defines* without the associated support routines; and a bug in their Motif.

However, we believe that the best indication that the system is effective is that our programmers like it and value it highly. Management does not need to pressure new programmers to adapt to the approach – their peers do that for us. This might not be important to some readers, but we strongly believe that a prerequisite to a successful project is a software process that the programmers feel helps them do their job, while, at the same time, it meets management's needs.

6. Stenning's Principles and Suggestions

This section lists Stenning's Principles and Suggestions of Project Hygiene. Any accompanying explanatory text, other than extracted quotations, is to redirect (if necessary) his principle to our area of concern, software hygiene. We have limited the extracts to the minimum amount necessary to convey their intent and the reader should read Stenning's original paper, as in places we may have distorted or misrepresented his meaning.

Principle 1

Everybody involved in the project should know the objectives of what he or she is doing.

> "Or, more bluntly, everybody should know what he or she is trying to do."
>
> "This is the fundamental principle of project hygiene, and should not need stating. However, it is surprising how frequently this fundamental principle is violated, if not totally then partially."

This principle applies directly to software hygiene, without modification. In his explanatory text, Stenning refers to problems with "fuzzy" requirements or inadequate specifications, but then goes on to state

6 The D-tree system will be distributed at the EUUG Nice conference.

> "...The problems arise when the requirements are ill-defined but people proceed as if they were well-defined."

So too in software maintenance and development. Even if the descriptions of software components are clearly defined, misunderstandings of how, where, and when the components are to be used can lead to failures in the software construction and testing.

Principle 2

Achievement of the overall project objectives should follow immediately from achievement of all individual objectives.

> "This is the necessary complement to principle 1."

In the discussion of this principle, Stenning refers to problems of integration and the phenomenon of seemingly correct individual components failing to combine to produce a working system:

> "...[E]veryone seems to know what they are doing, everyone seems to be doing a good job, and yet the project as a whole has major problems."

This problem should be familiar to any software developer and indeed to any engineering discipline, as illustrated by the recent failures within the 1.5 billion dollar Hubble space telescope, which turned out to be caused by a software error that could have been caught by correct testing and integration procedures which were not done for budgetary reasons (sort of mega-buck wise, giga-buck foolish).

In the arena of software hygiene we want to ensure that: if a developer follows the proper procedures for the development and testing of his or her source prior to "publication", then the integration of the changed source into the baseline system will be flawless. Unfortunately, sometimes the cost of full integration testing is prohibitive (the fully installed EMS system, which is a relatively small project, requires 130 Mb of disk on a MIPS). In such cases, the programmer must convey the threat of potential problems "upwards" to ensure that in meeting their objectives they do not jeopardize those of the project.

Principle 3

Both individual objectives and overall project objectives should be realistic.

> "People tend to be over-optimistic about what they are able to do and, most especially, about how quickly they are to do it.
>
> "There is no single factor more damaging to project hygiene than unrealistic targets."

The relevance of this principle to software hygiene should be obvious. Where Stenning uses "People tend to be", substitute "Programmers are invariably", but they compensate for this characteristic with the remarkable ability to under-estimate the impact that their changes might have on the rest of the project.

One must attempt to offset the programmer's natural enthusiasm for their product with a gentle comment such as, "We'll get to **that** later. At this time, let's just make sure **this** works, real [sic] soon."

Principle 4

There should be a known method for addressing each individual object.

> "Employing appropriate methods is an essential aspect of project hygiene – reliance on personal inspiration or hidden magic is unhygienic by definition.
>
> "When a new problem is encountered, we may well not know the method by which that problem should best be tackled. But under these circumstances we should know how we will explore the problem and determine the method by which the problem should be addressed."

There is an unfortunate tendency for programmers to be satisfied with a solution that works in the here and now, rather than looking for the solution will work or can be made to work everywhere. This is particularly true when addressing problems of portability. Often discrepancies are circumvented by using "quick and dirty" work-a-rounds, rather than looking for a solution that ensures that the problem is dealt with everywhere and will not "break" any previous release of the system.

Principle 5

Changes should be controlled, visible and of known scope.

No further elucidation is required, as Stenning is directly addressing the problems of software maintenance.

Principle 6

Both people and products should be insulated from the effects of changes that are not (currently) of relevance.

It is essential that mechanisms exist that can allow individuals within a project team to postpone changes until they are prepared to make the necessary adaptation. Unfortunately, full and unlimited support of this principle will often be too expensive to implement within a medium-scale project. However, in such projects, it is usually possible to delay changes to the baseline product, while not prohibiting their development and testing. With such an approach, everyone would use the baseline software by default, but have the option to explicitly incorporate changes not yet added to that baseline.

Suggestion 1

Focus on the process as a whole, rather than on the (final) product

> "Hygiene is not an attribute of an end product, but rather is an attribute of the process by which the product is produced. To improve project hygiene, we must focus on this process."

This is also true with software hygiene, which is a subset of project hygiene. If a hygienic process is used to control and change the baseline source, the production of the proper product follows as a natural result.

Later in this suggestion, Stenning states:

> "However, it should be recognised that the source code provides a very poor basis for such a process co-ordination."

In this document we emphasize software hygiene via a properly constituted process through which the baseline source is managed, modified, tested and used to produce the deliverables. This does not contradict the above statement. We strongly agree with it. The baseline source is nothing more than a modifiable representation of the product that can be transformed into the product using a deterministic procedure. As such, it is "a very poor basis for process co-ordination", but a well defined process for source management has many of the characteristics and facilities that would be required to properly support the "process co-ordination" of the medium scale project. Given that most UNIX developments have no strategy for the management of any project information other than source, it's a small step in the right direction.

Suggestion 2

Invest more effort in "higher level" descriptions.

> "Far too many projects still rush into code. Any time spent on higher level descriptions of the system – requirements, specifications, designs – is spent grudgingly and sparingly."

We enthusiastically adopt this suggestion, although our interpretation of "description" has to be clarified. Stenning is trying to bridge the enormous distance between

> "... [S]ome vague notion of what we want and actual running code",

whereas we are looking for better ways to represent the processes that transform the source into the product. By "better" we mean more reliable, more responsive, more flexible, more portable, more robust, more understandable, and considerably more succinct. *make*, *imake*, and even *make4*, just don't make it. They are, to what is required, what assembly code is to *miranda*.[7] This is not intended to denigrate Stu Feldman – he showed us the way, but *make* is fifteen years old and:

> "*Make* is most useful for medium-sized[8] programming projects; it does not solve the problems of maintaining multiple source versions or describing huge programs." [Fel79a]

To describe our solution (*qef*, the D-tree, and associated procedures) is far beyond the scope of this paper and is better done in [Til90a]. However, we discuss the essential characteristics of a software construction control system in a later section.

Suggestion 3

Co-ordinate activities as well as products

7 David Turner's (University of Kent, Canterbury) functional programming language.

8 Historical perspective is required here to appreciate what is meant by "medium-sized": At the time the article was written, UNIX could be run effectively, and not that unpleasantly, on a 250K non-separate I&D space pdp11/40 with two 5 Meg rk05s. Pure bliss was an 11/70 with a full meg, even if one had to share it with five other developers.

> "As has already been discussed, there has been a traditional tendency in the software industry to focus more on products than on process, and particularly on the program code."

In some ways, we must plead "non omnia possumus omnes''.[9]

Our emphasis has been directed towards those aspects of the software process that concentrate on the baseline source. Furthermore, such disciplines as Vic suggests are foreign to the prototypical UNIX developer whose initial design documents usually begin with

```
main(argc, argv)
```

However, we do agree, and to that end suggest that the baseline source system provides for the organization, management, and protection of **ALL** the information associated with a project.

Suggestion 4

Adopt a strict policy on project phases

> "Unfortunately, it is not always possible at the outset of a project to judge the realism of the objectives, simply because there is insufficient information on which to base such a judgement.
>
> "Under these circumstances, where the full scope and difficulty of the project are unknown, hygiene can be promoted by sub-dividing the project into self-contained phases – so self-contained that each phase is, in effect, a separate project."

In nearly all software projects involving the co-operative efforts of even as few as two programmers, well-defined, short-term, realistic goals and deadlines must be set, monitored, and met. The size and complexity of the individual phases must be determined keeping Stenning's 3rd Principle firmly in mind. There are programmers who can be safely assigned phases that run for months, but they are vastly out numbered by those who won't discover, or in some extreme cases, even believe, that they have a problem meeting a deadline until that deadline has past.[10]

Suggestion 5

Provide more semantic information to the configuration management and build systems.

It's at this suggestion that Vic drops into one of his long-term fantasies and the subject of many spirited (both figuratively and literally) Stenning/Tilbrook debates. It is also an area of great importance to software hygiene, being at the very heart of the product consistency issue.

> "As mentioned in section 2, *(the section of his paper discussing his fifth principle)* the co-ordination of continuous and concurrent changes is one of the most difficult aspects of project hygiene. While many mechanisms and tools have been developed to address various aspects of this problem, and many of these have been extremely valuable, there are still no complete solutions available."
>
> "Of necessity, in order to provide general and accessible solutions to urgent problems, most of the available change control mechanisms are based upon very simple models. The fact that something has changed may be measured, and the direct impact of that change assessed by the use of date/time stamps. Concurrent changes to individual modules are restricted by the use of check-out/check-in mechanisms. And so on." [11]
>
> "Clearly such mechanisms perform a useful function, but equally clearly they address only part of the problem *(He's up in the saddle again)*. Assessing changes on the basis of data/time stamps can become extremely inconvenient when one wishes to make a limited change to some previous release of the system. Check-out mechanisms alone do not address the issues of maintaining a consistent set of modules when a change may impact several others."

Actually, this paragraph should be repeated, with emphasis. Not only are the uses of time stamps or version numbers inconvenient, there are many situations in which they are far from sufficient. In the UNIX world, it is often dismaying how many people believe that running *make* ensures system consistency.

9 (L.) we cannot all do everything – Virgil.

10 Both the keynote speakers for this conference belong firmly in the second classification, if not the third.

11 Vic is exhibiting remarkable, although somewhat uncharacteristic, restraint. In person, this sentence would be repeated at least half a dozen times.

> "Over the past several years, there has been considerable progress in the fields of change management – in the context of such languages as Mesa, Modula, and Ada – from exploiting the syntactic structure of inter-component interfaces *(He's at the post).*"
>
> "There is considerable scope for further improvement by extending the checking to encompass not just syntax, but also semantics *(He's off! Unfortunately, his hobby horse is actually a somewhat pungent dead goat.)*"

Vic's motives and aims are admirable, but his insistence upon semantic checking over syntactic checking to discover process dependencies is suspect.

The primary objective of the build system must be to ensure that the product, as constructed, is that that would be produced by fully processing the selected source. Obviously, this can be accomplished by reconstructing the entire product, (provided there are no circular dependencies[12]). However, to fully reconstruct the system every time its source is changed is, just as obviously, undesirable. Consequently the use of mechanisms to eliminate redundant processing (i.e., would not produce any significant change in the produced objects) can be extremely valuable. Such mechanisms usually use some form of prerequisite dependency relationship and a model of consistency based on time/date stamp or version number.

Vic is proposing the use of semantic analysis to derive those dependency relationships and perhaps to replace the version or time stamp consistency model.

Our argument is that syntactic analysis is sufficient and, when one considers the relative costs and complexities of semantic analysis, is preferable. We base our position on the following observations drawn from our use of syntactic analysis and extrapolations of the relative cost and complexity of semantic analysis.

Every time our construction system is executed, those inter-component dependencies that can be derived syntactically[13] at run-time, are used to determine whether or not an object is "consistent". We have worked very hard to ensure that this syntactic analysis is affordable and accurate, yet it is still extremely costly. A new implementation is being done to use a lazy algorithm (which will also improve its accuracy) and to eliminate those file system or data base queries done by the dependency generator that are also done by the consistency checker.

We realize that it is unfair of us to cite experience with X11/Motif, but, it is those packages that present the most problems.[14] We have eight such programs, excluding those we use to test X11/Motif itself. When we apply our dependency tracker against our 36 source files that use Motif or X11 interfaces, we find that the average number of header files is 50.[15] However, many of the "#includes" are required to provide declarations that are unused in the majority of compilations. Therefore, we can assume that for our typical Motif source file, the number of relevant dependencies derived semantically would be substantially lower than the number derived syntactically. An accurate measurement of the reduction is difficult, but we estimate that it would be about twenty to thirty percent. It should be noted that, if proper coding practices are followed (e.g., for every interface used within a source file, its associated header files are included) semantic analysis should not yield any new dependencies.

Through monitoring we have discovered that large majority of reconstructions are due to a change to a major component (e.g., the C source file itself), or to multiple inconsistencies (target is "out of sync" with respect to many of its header files).[16] In either case it is unlikely that semantic analysis would eliminate any redundant processes. Therefore, this reduction would be a lot lower than the reduction in dependencies, due to the typical distribution of changes.

But at what cost?

12 If a construction uses previously installed information, multiple constructions may be required to guarantee consistency, but such situations are due to errors in the build ordering.

13 The "#include" of C is only one, but probably the most familiar, example of such a dependency.

14 Motif could be considered to be a proof of Stenning's Law that nothing succeeds like the Second Rate, but, more likely, it's proof of Tilbrook's plagiarised Corollary that Stenning is an optimist.

15 This is after unfolding the transitive relationships, which eliminates duplicates and repetitions. If they are not eliminated (which is the case in the compiler), it is not uncommon to find files that invoke hundreds of different include statements, since many of the required header files are repeated many times.

16 There are pathological cases in which a single touched header file can cause the entire source tree to be recompiled, but these are usually well planned and much heralded events (sometimes done deliberately).

As we have stated, our syntactic analysis is expensive, both in time and disk space. Yet, semantic analysis will incorporate the equivalent syntactic scan plus its own analysis and will need to preserve far more information. Once a change is detected, syntactic analysis can be limited to the changed file, whereas semantic analysis will require analysis of all subsequent files as a change in a file can change the semantic interpretation of a following file. Therefore semantic analysis will cost a great deal more than syntactic analysis, yet yield a marginal reduction in redundant processes.

On another issue, our system uses a relatively simple finite state automata. We support six different languages, the most complex of which has six states and fifteen productions. The addition of support for a new language is relatively simple. The creation of the F.S.A. tables is all that is required to incorporate it into the construction process. It is obvious that the creation of packages that perform semantic analyses will be a difficult and costly task, perhaps too difficult and costly to be feasible.

Finally, the problems addressed by semantic analysis are strictly within the realm of the compilation of a programming language. A large proportion of the production process is compiling, but almost as large a proportion is not. For our projects, C, yacc, and lex source files constitute only thirty percent of our total number of files. Developing an approach that can handle semantically derived dependencies, without penalizing the rest of the construction process will be difficult.

No doubt the irascible Dr. Stenning will want to respond, perhaps citing Protel – aw yes ... Protel ... management compared it to sliced bread, and the programmers hated it.

7. Dr. Tuna's Patented Software Nostrums

Up until this point, we have concentrated on generalities, commandments equivalent to "Bathing is good" and "Exercise is beneficial to your health." Now we get more specific: Dr. Tuna's Patented Software Nostrums are carefully designed to keep your system regular, your inodes clear, your backbone straight, and your disks unslipped and free of bitrot. Following these guidelines will extend your project's credibility and endear you to people who count.

These guidelines are for the most part self-evident and self-explanatory; where they are not, some explanation is included.[17]

1. Give a damn! The first and foremost principle is that you must actually care about having a hygienic system. This means committing time and resources to the problem and convincing everyone involved that maintaining a clean system is important and a high priority.
2. There should one and only one source! This source should be used for **all** constructions for **all** target environments.
3. Remember that the product is the source, even if the deliverable is the result of processing that source.
4. Ensure that the deliverable product is that which is constructed from the provided source using a well-defined non-interactive single pass process that aborts if any phase fails. Minor pre- and post-construction steps are allowed, but they must be well documented and fool-proof.
5. Centralize all construction and installation parameters in one well documented and easily modified file. Procedural aids to validate the information are useful, but automatic procedures to deduce the information are difficult to maintain and frequently wrong.
6. Never require the expression of construction information twice. The second instance will be inevitably wrong.
7. Adhere to Stenning's Principles and try to follow the first four and a half of his Suggestions.
8. Make sure that everyone involved understands and appreciates the software process. Everyone in the software project must understand the mechanisms used to create, modify, construct, and maintain the product. Furthermore they must understand the objectives of the software process and why it is important to them.
9. Test, test, test, and then test again.
10. Avoid simultaneous introduction of dramatic changes. When introducing a dramatic change, always start with a known recoverable healthy system.

17 Further explanation can be purchased at the rate of one beer per nostrum.

11. Create and frequently use a file system integrity package that checks for spurious files, missing files, obsolete files, and so on. This should check the source tree, object tree, and the installed tree. The source file system check should be done daily. One of the by-products should be mail (or news) to programmers informing them of their outstanding edits. The object tree and installed tree checks should be part of the construction process.
12. Work hard to separate the baseline, object, developers' source, and destination trees.
13. Adopt a standard source directory architecture and constantly test and tune that architecture. Construction ordering is defined in terms of directories. Watch and test for circular dependencies.
14. Limit local construction dependencies to sibling directories or fully installed sub-systems.
15. Do not restrict the methods programmers use to do their own development in their own tree. Rather, restrict the way they publish their results.
16. Test your construction frequently on as many machines as possible to catch compilation errors as soon as possible.
17. Always build the full system. You never really know the entire scope of the changes.
18. On a regular basis, do a bare metal construction – a complete reconstruction of the system on a machine that has been purged of any evidence of the system's previous existence. If the bare-metal build fails, fix the source, purge any remnants of the failed build and do it again. We do this at least monthly, and as the last test before shipping any source.
19. Ensure that the product is frequently tested against your own well-understood standards.
20. Ensure the deliverable is trivially relocatable through a change or specification of a single directory. Furthermore, with the exception of non-sharable resources, multiple versions of the deliverable should be executable in parallel on the same platform. Switching between versions should be trivial and complete (i.e., no danger of cross version talk).
21. Any faith in the standards effort is misplaced. Don't believe in them and you won't be disappointed. They are naive at best, misguiding at worst.
22. Do not insist on any standard that cannot be tested.
23. Adopt a standard for file naming and adhere to it. Overloaded suffixes are forbidden.
24. There is no paragraphing standard worth the time and energy required to impose it, other than be consistent (but reasonable).
25. Ensure that the construction system supports modification of all paths and automatic reconstruction if a change is significant.
26. Restrict baseline production installations to a small group, preferably the group that approves changes to the baseline.
27. A change to the baseline source means total re-installation ASAP and all personnel must recognize that fixing construction problems in the baseline is the highest priority.
28. Record your bugs, and make someone responsible for ensuring that they are fixed.
29. Use *lint* – when it works, it really helps. We use six different compilers, but *lint* still finds bugs that all six missed.
30. Design a test program strategy and provide facilities to quickly build programs that conform to that strategy.
31. Have others test the entire procedure of source-to-installed-client-site.
32. After any significant problems (especially after disasters), perform a post mortem. The unexamined failure is not worth having.
33. Isolate all machine dependent sections in a well known location or using a well known and consistent mechanism.
34. Centralize environment dependent aspects in single location.
35. Never use */usr/include* directly from a C source file. Always redirect through your own provided header file. This will allow you to circumvent their idiocies.
36. Never use "#include "file"". Always use "#include <file>".

37. –D to be considered harmful. If required, ensure that they are applied universally and consistently and are documented.
38. Don't overload source files by compiling them in different ways. Remember: one source file to one type of object file.
39. Develop or adopt a system whereby documentation can be incorporated within the source code files.
40. Develop and use boiler plate for often-written pieces of text, including headers and documentation.
41. Design the system to be ported, from the very start.
42. Avoid compiler-dependent tricks.
43. Avoid special case constructions. If they are useful, make them universal.
44. Avoid differing implementations to support optional semantics.
45. Use function prototypes everywhere and ensure that everywhere a function is used, its prototype is included.
46. NULL is a four letter word, the meaning of which is unclear, and whose use is dangerous.
47. Minimize your dependence upon suppliers. Expect little from them, and you will not be disappointed. Resist the urge to fix their bugs. It's unlikely your clients will applied the same fixes.
48. Use an installation procedure that validates that **cp** did not change the size of the installed file (a not uncommon event on some file systems).
49. Use an installation procedure that refuses to install an empty file without an explicit override. This catches yet another common file system vice.
50. Ensure that the undamaged installation of every program can be trivially checked. One way to do this is to insist every program supports a flag that explains its flags and arguments (e.g., the D-tree –x flag). Then just run every program with that flag.
51. Who cares what your programmers wear, so long as they don't shock the secretaries?

Caveat: This list is by no means complete! Each time we came back to this section, the list had grown again. Rather than attempt to offer all of the possible guides to correct software hygiene, we shall remind you of the most important nostrum of all...

Remember your two primary objectives: to have fun and make money. Any other objectives are counter-productive.

8. Characteristics of a Viable Construction System

Throughout this paper our emphasis has been on the conversion of source into the product via a well known and reliable mechanism. At our site we use *qef* (Quod Erat Faciendum) for every construction and as the driving process behind product installation. Naturally we believe *qef* is essential to good software hygiene; however, any construction system that has the following characteristics will do:

- **Constructions and Installation without Change to Source:** The construction system should support the initial installations of large collections of software, on a variety of machines, at a variety of sites, without requiring the modification of **any** source, using a trivial configuration mechanism (e.g., fill in the blanks) and a single command.

  Source is defined to be **all** the information that is created manually or must be provided by the supplier. This, by definition, includes **all** the information used to control construction and installation.

- **Termination on Error:** The construction system should halt whenever an error or condition arises that indicates that a construction has not been successful. The traditional approaches miss many failures due to inadequate checking by the tools or incorrect error returns on the part of the commands being executed. However, errors that can be rectified after a complete installation should not terminate the build, but should report the problem in a consistent manner so that the correcting actions may be performed manually.

- **Porting without Pain:** The construction system should be easily ported to any reasonable UNIX system and provide a totally consistent environment on any system to which it has been ported. The system should support the construction and installation of software on many different environments **without** change to the source itself. Furthermore, the system should promote portability by providing an approach whereby software can be built, tested and installed on many platforms simultaneously.

- **Ease of Distribution Upgrade or Modification:** The system should support the upgrading of a previously installed body of software by the simple addition of new source files and/or the removal or replacement of old source files, and the issuing of a single command. Furthermore, the addition or removal of source files should rarely, if ever, require the modification of the construction control files.

- **Generalized Controlling Information Specification and Use:** The system should provide mechanisms to specify and/or retrieve required control information (e.g., the target location of the installation) in or from one and only one place, or through one and only one interface. In other words, user supplied information should never have to be expressed more than once. A user should be able to feel confident that all aspects of the construction system, that need such information, should retrieve it in a manner that ensures consistency across the entire system and all uses. Furthermore, it should be possible to ensure that correct and consistent values are being used no matter what part of the system is invoked. That is to say, the information should be the same when retrieved during a full or partial construction, or the invocation of some subset of the construction system.

  An extension, or almost a prerequisite, of this objective is that the system should support the trivial addition of new pieces of construction control information. For example, if the system needs to support *whatsit* compilation, there should be a simple mechanism to specify the name of the *whatsit* processor and its normal flags through a unique and universally accessible interface.

  Finally, if the default value of a piece of control information needs to be changed or overridden within some subset of a source distribution, it must be ensured that the modification is effective over the entire subset.

- **Required User Supplied Information Reduced to Minimum Possible:** The system has to provide mechanisms to express construction specifications as succinctly as possible.

  For example, in a directory of source that is compiled and archived into an object library, it should be sufficient to state: "build and install a library called X". From this statement alone, the system should be able to generate all the required information that will perform that task.

  The mechanism used to convert simple specifications into the full construction procedures must be applicable to as wide a range of applications as possible. It should be relatively simple to add new procedures. Furthermore, it must be possible to state possible variations to a procedure easily and tersely. Such variations should not require the total restatement of the construction rules.

- **Phased Construction and Distribution Subsets:** The construction system must support partial or phased constructions such that they are absolutely equivalent to the same constructions taking place as part of a full construction. That is, the user should be able to select parts or phases of the construction to be done and be assured that the behaviour is equivalent to those same phases being done as part of a full construction.

  Furthermore, there should be few (if any) modifications required if the source to be processed is a subset of the full distribution. For example, one should not have to modify the construction control information when parts of the full distribution are excluded for economic, licensing, or other reasons.

- **Consistency Between Incremental and Full Constructions:** The system must guarantee the consistency of objects produced by an incremental construction with those that would be produced by a full reconstruction. Incremental reconstructions are defined to be those constructions that skip some construction steps if the object to be produced is determined to be consistent with its component files. The construction system should ensure that an object is reconstructed if any aspect of its construction has changed that might result in any significant change to the object itself.

- **Support for Testing and Development:** Finally, the system should facilitate testing and modification without endangering the production versions, whilst ensuring the almost trivial addition of those changes or extensions when completed. Experimentation with the source by a programmer should be economic and easy to initiate. The system should ensure that the behaviour of a test system in a non-production environment is truly reflective of the behaviour of that system in the production environment.

9. Conclusions

And thus ends our initial foray into software hygiene. We have barely scratched the surface, but the reader should now have an appreciation of the importance of a proper system construction and installation system, and the difficulties involved.

We also hope that the reader is aware that software hygiene *can* work, can be achieved, can be maintained. It can be done using the principles included in this paper. We have concentrated upon generalities, and upon techniques and objectives that can be *immediately* applied to new software projects. It is also possible (difficult, but possible) to apply these principles to existing projects.

Instead of presenting a full-grown system of software hygiene, we have tried to lay out the bare bones of a philosophy which attempts proper system construction and installation.

Anyone interested in further work in this area is recommended to think about the following topics, which have not been discussed here:

- Documentation strategies
  - styles and forms
  - administration and publishing
  - supplementary support tools
- Configuration strategies
  - natural language specific
  - {site/client} specific, options, client proprietary
  - machine/system specific
- Version strategies
  - resurrection of old releases
  - removal of files and renaming
  - documenting and controlling changes
- Testing strategies
  - Validation suites
  - Local testing, complete testing, *in situ* testing
  - Forms and contents of testing databases
- Post-release Maintenance
  - Version naming and tracking
  - Bug tracking and auditing
  - On-site trouble shooting and support
  - Upgrading and auditing of releases
- Interdependency of deliverables on other products

The objectives of applying proper software hygiene are ambitious. The task is a difficult one; any approach that attempts to solve the problems of medium-to-large scale software will itself be complicated and/or expensive. It will have to deal with a great many special cases – the ability of programmers to create new ways of complicating the lives of the software manager is unbounded.

However, we leave you with this thought from our own experience:

It's worth it.

Appendices

A. The EMS Software Process

The EMS software development process makes extensive use of the tools of the D-tree, which have been described elsewhere [Til86a, Til87a, Til89a, Til90a]. A complete explanation of the D-tree tools (such as *qef*) is impossible; however, it is the process of development that is important here.

A.1. The Source Development Process

Source which has been incorporated into the product (referred to as the baseline or "published" source) is maintained in a separate source directory. Each programmer works in his or her own source directory. The following is a brief description of the way in which the programmer develops, modifies, and "publishes" source.

To initialize his or her work space, the programmer creates a source directory. This directory is used to contain any new or modified source files. The programmer copies a prototype of the construction parameter file into this directory, edits it to set the construction-specific values (i.e., flags to compilers, destination directory) and then uses a program called *treedup* to configure the construction control file, called *TreeVars*, and to duplicate part or all of the directory hierarchy of the baseline source tree in his or her source tree. This procedure is repeated to create as many parallel object trees (trees in which the system is constructed) as are required. Usually the programmers will have a tree for their main development machine and will create object trees as required to test on other platforms.

It is important to note that the only source file used in the construction process not stored in the baseline is the *TreeVars* file. This file contains all the parameters which may differ from one construction to the next.

To construct the system from the baseline source, the programmer changes to a directory in the appropriate object tree and goes:

```
    qef
```

This constructs the product from the baseline source, directing any installations required to construct the system to the destination directory specified via the construction parameterization file. Note that all source files are accessed via the full pathname to the baseline source. The only files in the programmer's source tree will be files that he or she has created or has fetched for editing.

To add a new source file, the programmer simply creates the file in the appropriate directory of his or her source tree. When *qef* is rerun, new files are added to the source database that is used to create the directory's construction script. In the vast majority of cases, no further modification of the source is required to incorporate the new source file.

To modify existing source, the programmer moves to the appropriate directory in his or her personal source tree and fetches the file for editing using:

```
    sc edit <files> ...
```

sc was originally based on Eric Allman's *sccs* interface[18] (i.e., yet another *CS wrapper), but has been much modified to support remote source baselines. The above command fetches the named files for editing, that is, a writable copy of the file is created in the current directory, and the SCCS administration file is locked via the creation of a p-file. The existence and visibility of this p-file is an important factor in choosing SCCS over RCS. When one is administering 3500 files over 300 directories, a quick way to find all the files currently being edited is crucial. We have also modified the p-file creation routine to embed the name of the host and directory of the *fetched* file in the p-file. We also strongly believe in ensuring that files are not being modified simultaneously by multiple programmers. RCS supports this as an option that can be easily circumvented or over-looked.[19]

The programmer then edits the source as required. When *qef* is rerun in the object tree, files in the programmer's source tree versions of the file take precedence over those in the baseline.

The procedure of fetching files for editing into the programmer's own source tree and rebuilding is repeated as many times as is required to implement the current development.

Note that throughout the development, the programmer's source tree need only contain those files required that are either new or modified versions of baseline source and the object tree contains only files that are constructed. This total separation of baseline source, modified source and constructed files has many advantages. Perhaps the most popular among our programmers are: only the command "ls" in the source directory is needed to determine what files a programmer has changed; only "rm *" in the object tree is needed to remove constructed files.

18 We chose SCCS over RCS for a variety of reasons, most of which were outlined in [Til87a], although some are mentioned in this paper.

19 There is a work-station supplier that prides itself on providing CASE tools that support simultaneous modifications of source files. We have one of their products and it's the least reliable system in our office. The inference is clear.

When the programmer is satisfied that the modifications are correct and they have been tested (preferably on multiple platforms), the programmer **push**es the modifications using:

```
    sc push files ...
```

These commands "push" the named files to the **unpub** (unpublished source) directory, which is used as a quality assurance administered buffer between the programmer and the baseline source. The programmer is required to provide a description and rational for the changes to the file via a specially named file, a −y flag (a la *delta*(1)) or as input.

At regular intervals (usually daily), the QA group checks the *unpub* directory for unpublished files. Time permitting,[20] the product should be constructed from the *unpub* source (it's just another source tree) and tested before the modified source is added to the baseline. Given that the change is acceptable, the QA group "publishes" the modification. This *delta*s the changes into the file's SCCS administration file, updates the baseline g-file (the source in clear test), and adds an activity record to the delta audit trail. It is then QA's responsibility to construct the baseline product on all the machines on which there is on-going development (machines used for demos and full release testing are excluded). If any problems are encountered, they are immediately rectified. If a programmer introduced an error in the construction due to one of his or her modifications, it is of the highest priority to rectify the problem immediately and convey necessary changes to the QA group.

A.2. EMS Documentation

The EMS system's primary objective is to provide a tool-kit with which others may build client-specific applications. Consequently, documentation of the basic system and, in particular, of the 700 subroutines in the provided libraries is of utmost importance.

In our approach, documentation is treated in exactly the same manner as any other source. It is maintained and modified using the same tools and procedures. In fact the mechanism to "produce" the documentation is the same as to produce the other products (all one ever says is "qef").

To document subroutines and libraries we use database entries (referred to as *man3db* entries) embedded in the source files. The *man3db* program extracts these entries and creates a variety of products including: the traditional *nroff* man file; an indexed fast retrieval database; a software inventory; *lint* libraries; and a prototype database. One important aspect of this approach is that the documentation is *lint*able, in that the synopsis as defined in the *man3db* entry is checked against the function itself using *lint*. Most importantly, any time a programmer is modifying source, he or she is also modifying the file that contains the appropriate user documentation. Consequently, changing the documentation to agree with changes in the routine's semantics and/or interface is simply a matter of scrolling up a page in the editor.

Finally, all the documentation associated with the project is stored in the project source tree. This includes the project plan, designs, specifications, discussions of those descriptions, installation notes, testing reports, project inventories, validation suites, post installation instructions, the bug database, change audit trails, and information to resurrect previous releases.

A.3. Bug Tracking and Testing

Every source component in the EMS source tree is assigned an owner – the developer responsible for the correctness of the component.[21] Every component that is to be released, is assigned one or more testers (developers not involved in its maintenance). A tester is responsible for reviewing the code for completeness, conformance to the few coding standards that exist (see nostrums), agreement of the component's code and its documentation, the completeness of the documentation, and for the creation of the component's testing procedures and tools. Testers must review and test those components which have been assigned to them, though they may report problems wherever they find them.

Whenever a tester discovers a problem, a bug report is prepared and mailed to the QA group. The bug report contains the source module that contains the error (if known), the priority of the bug (p0 for "cosmetic," p4 for "renders system unusable"), a one-line description of the bug, a long description of the bug (including how to duplicate the problem), suggestions as to how the problem may be rectified, the

20 It never is.

21 Some organizations (e.g., UCB's CSRG) use the scheme that the last person who modified the file is responsible for it. However, this scheme discourages people fixing cosmetic problems or minor errors that do not require a complete understanding of the module.

environment in which the bug occurs, the date the bug was reported, and the reporter's (a.k.a. *the buggee*) name.

On receiving a bug report, the QA group attempts to validate it and checks if the bug has already been reported. If the bug is legitimate and new, the bug report is assigned a reference key and the owner of the source file is assigned the bug. The bug report is added to the bug database, mailed to the buggee and the module's owner (a.k.a. *the bugger*), and posted to the bug-newsgroup.

The owner is then responsible for fixing the bug, testing the fix (if appropriate), "pushing" the fix, informing QA that the fix has been pushed. QA then asks the buggee to verify that the fix is indeed correct and has not introduced new problems.

To encourage this process, the team is awarded minor bonuses for reaching plateaus of fixed bugs (most recently a beer and pizza lunch). On a daily basis, buggee/bugger counts, graphs and tables of the bugs, and the countdown to the next free lunch are produced and posted on bulletin boards.

It must be noted that performance reviews are not based on bug counts. To do so would require more rigorous evaluation of bugs than they merit or than can be afforded, and would discourage the reporting of minor problems and inconsistencies. We foster mild friendly competition, but avoid reprimands. The overall emphasis is that this exercise is a team effort whose objective is to improve the quality of the software to the entire team's benefit and satisfaction.

References

[Fel79a] S. I. Feldman, "Make – A Program for Maintaining Computer Programs," in *Unix Programmer's Manual*, Bell Laboratories, N.J. (January, 1979).

[Lie79a] B. P. Lientz and E. B. Swanson,, "Software Maintenance: A User/Management Tug of War," *Data Management*, pp. 26-30 (April 1979).

[Mey88a] Bertrand Meyer, *Object-oriented Software Construction*, Prentice-Hall (1988).

[She89a] Barry Shein, 1989.

[Ste89a] Vic Stenning, "Project Hygiene," in *Usenix Software Management Workshop*, New Orleans (1989).

[Til89a] David Tilbrook, "Under Ten Flags (not always smooth sailing)," in *Usenix Software Management Workshop*, New Orleans (1989).

[Til90a] David Tilbrook, "Quod Erat/Est/Erit Faciendum," in *The EUUG Fall 1990 Distribution Tape* (1990).

[Til86a] D. M. Tilbrook and P. R. H. Place, "Tools for the Maintenance and Installation of a Large Software Distribution," in *USENIX Atlanta Conference Proceedings* (June 1986).

[Til87a] D. M. Tilbrook and Z. Stern, "Cleaning Up UNIX Source -or- Bringing Discipline to Anarchy," in *EUUG Dublin Conference Proceedings* (September 1987).

Driving the Software Release Process with Shape

Ulrich Pralle

Technische Universität Berlin
Sekretariat FR 5-6
Franklinstr. 28/29
D-1000 Berlin 10
uli@coma.cs.tu-berlin.de

ABSTRACT

While conventional UNIX tools provide acceptable support for elementary software management functions, such as version control and basic configuration management, there is a substantial lack of support for higher level system management functions, such as release preparation and change management for complex software systems. One major reason for the development of the *Shape* toolkit was to improve this situation by providing a close integration of basic software object management and configuration control functions.

The toolkit is a set of version control commands and *Shape*, a configuration management tool, which are integrated on top of a dedicated version object base. Shape uses a rule based approach to identify and select particular version objects in the object base by means of arbitrary attributes. It records identified configurations in unique *configuration identification documents* making it easy to maintain histories of entire software systems. Tool-, variant-, and version selection definitions are specified together with object dependencies as versionless *abstract system models* in *Shapefiles.*

While it is generally easy to write Shapefiles for small (one programmer) projects, this task can be tricky for medium sized (or large) projects consisting of many subsystems developed and maintained by several programmers. In this paper we describe an easy to use yet powerful, user customizable product management system on top of the Shape toolkit. It enables individual programmers to maintain and develop subsystems of a larger project without loss of overall project integration. Administration, maintenance, and installation are considerably simplified by providing standardized functionality. Although the management system requires programmers to follow certain conventions, it provides good support for software system management.

1. Background

The Shape toolkit [Mah88a] is a collection of programs for version control and configuration management for UNIX based software engineering. It consists of a set of version control commands and Shape, a significantly enhanced *Make*-oid [Fel79a]. Shape and the version control commands are integrated on top of AFS (*Attributed File System*), a dedicated version object base [Lam88a]. The system features RCS-style version control [Tic85a] and a configuration identification and configuration build process that has full access to all revisions in the object base – other than Make which only knows about plain files.

AFS is a system to store software objects which are understood as a complex of content data and a set of associated attributes. These objects are called *attributed software objects.* AFS' built-in version control system supports the storage, identification, and retrieval of different versions of *lines of development.* A line of development consists of one changeable object (*busy version*) represented by a conventional UNIX file and a group of successive, immutable versions stored in a separate AFS archive file. AFS provides a retrieval interface that allows to uniformly access ordinary file system objects as well as versions stored in AFS archives. It also supports derived object management, i.e. it maintains a cache of multiple versions of compiled object-files.

The Shape program itself is upward compatible to Make in that it can properly handle conventional *Makefiles.* The *Shapefile* however, uses Makefile-style dependencies as versionless *abstract system model* and employs *configuration selection rules* (similar to DSEE's *configuration threads* [APO85a]) and *variant* *definitions* to dynamically bind particular version objects in AFS to the names listed in the system model. The version selection mechanism exploits AFS' ability to store any number of arbitrary attributes for objects in the object base. Shape uses user-supplied selection rules to uniquely identify (and select) certain software objects in AFS by means of attributes. A selection rule is a named sequence of *alternatives* forming a logical OR-expression. An alternative consists of a sequence of predicates expressing constraints that considered AFS objects must meet. This predicate sequence forms a logical AND-expression. A selection rule succeeds if one of its alternatives succeeds. An alternative succeeds if all of its predicates succeed.

Shape does not impose a particular variant handling scheme. It rather supports various commonly used variant handling techniques by providing a dynamic macro (re-)definition facility. This facility can for instance be used to implement the following variant handling techniques:

- equally named files in different directories (by modifying Shape's search path),
- one source file representing multiple variants that are extracted by a preprocessor using different preprocessor switches (conditional compilation).
- one source file processed by different tools or different tool versions/variants, and
- the combinations of the above.

Even structural variants can be supported by dynamically modifying the set of components of a certain configuration.

On special request, Shape records an identified configuration in a *configuration identification document* (CID) which has the form of a completely bound Shapefile, i.e. all object references are explicit. This makes it particularly easy to rebuild recorded configurations. As CID's are themselves documents, they can be placed under version control, making it easy to maintain histories of entire software systems.

2. A Management System for Shape

In 1989, when our toolkit was mature enough to be released to the public, we found ourselves in a mess of configuration management problems. During the development of the toolkit each programmer in our four person project group maintained his part of the project separately from each other. Shape-virtuosos wrote sophisticated but difficult to use Shapefiles, while others were content with their simple Shapefiles only suitable for building their specific programs. Rules for semantically equivalent activities were named differently, some Shapefiles didn't provide functions which others offered, and in general the usage varied for each Shapefile making a straightforward integration, administration and maintenance of the entire toolkit difficult. Another problem was that installation of the toolkit in different operating system environments was hard to manage, because the toolkit has to be bootstrapped with Make, i.e. a first Shape toolkit installation has to be done with Make. Additionally, redundant information was specified in Shape- and Makefiles.

We were in need of a product management system which gives individual programmers the freedom to develop and maintain their subsystems separately from their partners but doesn't restrict proper project administration tasks like system release or system distribution of the entire software system. We ended with a management system which fulfills the following objectives:

- avoid redundancy of system description information.
- support of installation of the toolkit (or parts of it) on a variety of different system platforms using Make for the initial bootstrap,
- proper control for building internal or published versions,
- automatic generation of module dependency information,
- a common system release procedure, automatically ensuring that all components are version safe and in a stable state making them eligible for release (tested and no modifications pending). All component versions of a certain release share a common, automatically generated release name.
- generation of a unique *configuration identification document* permitting to *rebuild* system releases at any time later.

Very much like standard transformation rules (implicit dependencies) which are "hard-wired" into Shape, version selection rules and variant definitions rarely need to be modified once they have been defined for a given project. So, rather than having each developer maintaining her own set of these rules and definitions, it makes sense to factor out these pieces of information and place them in a central location where they can

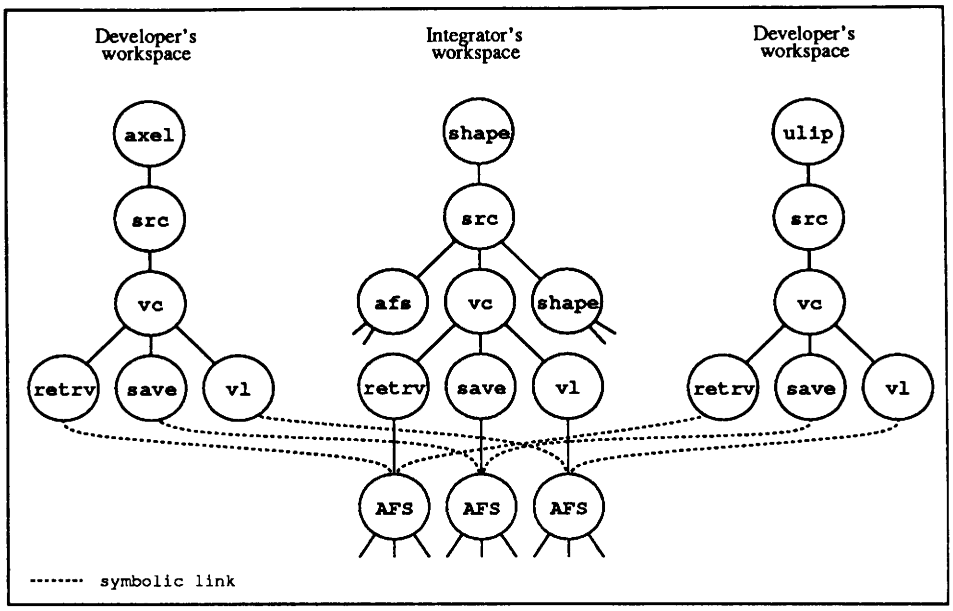


Figure 1: *Sharing AFS directories*

be accessed by all team members. In fact, many pieces of a Shape- or Makefile are candidates for sharing within a project group. Shape features an include mechanism to support this technique. As will be shown in the subsequent sections, a carefully designed set of "Shapefile building blocks" provides a highly functional software management system that adheres to project wide standards. Team programmers can easily use this system while investing only a minimum of effort.

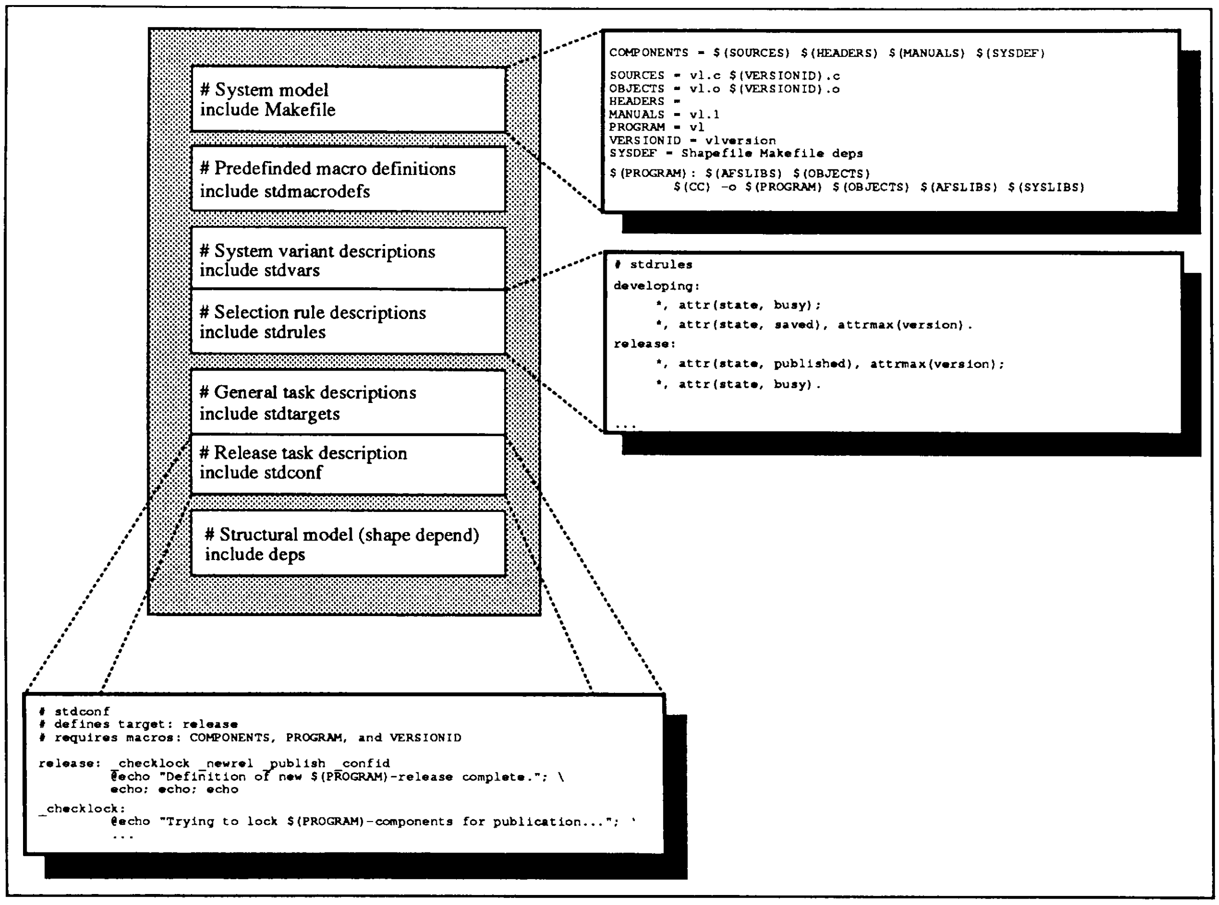
The emerging software management system (SMS) consists of templates for Make- and Shapefiles, and a set of files defining rules for version selection, variant and tool definition, release control, and system maintenance.

2.1. A Typical Project Setup

A typical small software system (for a project that uses C as implementation language) making use of the Shape toolkit consists in general of a set of source objects (like program modules, documentation, manuals, etc.), a collection of documents defining the system's structure (the system model), and task descriptions. The software objects of a particular application system are grouped together in a separate directory. Their evolution is maintained with the shape toolkit's version control system. All object versions and their attributes are stored in the object base residing in a subdirectory named AFS.

Programmer teams can work cooperatively on the same system by sharing the same AFS archive files. In our development of the Shape toolkit all AFS archives are located in a separate *integration environment* under control of a special project team member: the *project integrator* (namely the users "shape"). The integration environment is a directory tree reflecting the structure of the toolkit's subsystems. For example, the Shape toolkit's integration environment consists of directories for the subsystems *afs* (AFS library), *shape* (the program), and *vc* (version control commands) which itself is structured into the subsystems *save*, *retrv*, *vl*, etc. Each directory contains only an AFS directory holding the appropriate AFS archive files. All directories and AFS archive files are owned by the project integrator who is not involved in development. The integrator prohibits development in the integration environment but permits read- and write access on AFS directories to a certain group of developers. As depicted in Figure 1, team programmers can then share AFS directories by creating a *symbolic* AFS directory in their private workspaces pointing to the AFS directories in the integrator's workspace.[1] Team programmers develop parts of a systems in their private workspaces and publish their results with the version control commands into the public integrator's workspace. Developers using modules implemented by their partners retrieve only stable versions of these modules from the integrator's workspace.

1 For operating systems lacking symbolic links, AFS provides a pseudo symbolic link feature.


```
# System model
include Makefile

# Predefinded macro definitions
include stdmacrodefs

# System variant descriptions
include stdvars

# Selection rule descriptions
include stdrules

# General task descriptions
include stdtargets

# Release task description
include stdconf

# Structural model (shape depend)
include deps
```

```
COMPONENTS = $(SOURCES) $(HEADERS) $(MANUALS) $(SYSDEF)

SOURCES = vl.c $(VERSIONID).c
OBJECTS = vl.o $(VERSIONID).o
HEADERS =
MANUALS = vl.1
PROGRAM = vl
VERSIONID = vlversion
SYSDEF = Shapefile Makefile deps

$(PROGRAM): $(AFSLIBS) $(OBJECTS)
        $(CC) -o $(PROGRAM) $(OBJECTS) $(AFSLIBS) $(SYSLIBS)
```

```
# stdrules
developing:
        *, attr(state, busy);
        *, attr(state, saved), attrmax(version).
release:
        *, attr(state, published), attrmax(version);
        *, attr(state, busy).

...
```

```
# stdconf
# defines target: release
# requires macros: COMPONENTS, PROGRAM, and VERSIONID

release: _checklock _newrel _publish _confid
        @echo "Definition of new $(PROGRAM)-release complete."; \
        echo; echo; echo

_checklock:
        @echo "Trying to lock $(PROGRAM)-components for publication..."; \
        ...
```

Figure 2: *Typical structure of a Shapefile*

To avoid concurrent updates, the version control system provides a locking mechanism. If a user wants to change a software object, she retrieves a certain version with the command *retrv -lock* implicitly locking its development line against subsequent retrieve- or save commands. While a lock exists other developers sharing the same AFS archive can retrieve versions of this software object in read-only mode. A lock is released either explicitly by giving up the lock or implicitly by saving the busy version.

System Structure Description

The description of the system's structure is distributed over three documents: a Shapefile, a Makefile, and a file named *deps* containing a generated list of object dependencies (include dependencies only). The Shapefile as depicted in Figure 2 is the root of the system information description. It includes all information of the management system, i.e. standard target definitions, predefined macro settings, version selection-, and variant definitions. It contains application specific information for certain version selection and variant definitions and in addition it includes the Makefile.

The Makefile reflects the system model information as a collection of predefined macros and a set of target definitions and transformation rules. It provides rules for the system building- and installation task, i.e. the definition of the *main-* and *install target* as well as a general clean-up target. The Makefile must be informationally complete and self-contained in the sense that it can be successfully interpreted by different Make programs of different operating systems (e.g. 4.3BSD's, SunOS', or AT&T's Make). Especially it may not contain features which are only supported by newer Make implementations. Although the Makefile is not necessarily needed for a developing phase (because all its information can be placed in the corresponding Shapefile), it is distributed along with the software to make a "stand-alone" configuration build without the Shape toolkit possible. In contrast, the Shapefile integrates conventional and elementary configuration building facilities with higher level system management functions needed at the developing site.

The abstract system model of a given software system is described by copying a generic Shape- and Makefile, filling in a few predefined macros, and generating a list of module dependencies by calling *shape depend.*[2] The most important macros which users must supply are:

- **COMPONENTS** which contains as a list all source software objects of a certain software system. This includes source language- and interface description objects (like .c and .h files), documentation and online manuals, as well as the Shape-, Make- and "deps'-file.
- **PROGRAM** specifies the name of the software system in question. It is used by the management system as a name ingredient for various, automatically created objects (for example for the configuration identification document or for a distribution archive file generated by *shape tar*), and secondly as the name of the main target in the Makefile.
- **VERSIONID** determines the name of a so called *version identification document* (VID). The VID is an implicit component of a software system in the sense that it is an automatically created software object. It is used to derive from its name and AFS version attributes a release identification for a given configuration. The contents of a VID is a single routine written in the C programming language returning a version identification string, which can be used by the running application system to identify itself.
- **OBJECTS** contains the set of derived objects, which are in general loadable UNIX files. The mentioned Makefile template contains generic target definitions which make use of the macro OBJECTS, for example in a definition of the main target:

```
$(PROGRAM): $(OBJECTS)
        $(CC) -o $(PROGRAM) $(OBJECTS) $(SYSLIBS)
```

To enable Shape's version- and variant selection mechanism one can apply a *version selection*- and several *variant definition rules* to each user supplied transformation rule specified in the Makefile. For example, consider a typical *version-less* transformation rule of a Makefile:

```
prog: $(OBJECTS)
   $(CC) -o prog $(OBJECTS)
```

Version- and variant selection is enabled by supplying the following Shape-specific transformation rule in the Shapefile:

```
prog: $(RULE) +$(VARIANT)
```

If the first item of a dependency list determines a known selection rule, Shape enables this selection rule. Otherwise the token is treated as an ordinary dependency. Variant definitions are indicated by a leading plus sign before a variant name. Variant definitions as well as the selection rule are valid as long as the transformation is active. Several equally named transformation rules can be specified in Shape- and Makefiles. Shape interprets them in the order they are found until a complete transformation rule (a rule containing a command sequence) is detected. The other textually following, equally named transformation rules will be ignored. Therefore rules enabling version selection and variant definitions must precede their counterparts in the Makefile. It is a good practice to apply selection rules and variant definitions as macros to have a more flexible and paramterizable mechanism.

The management system provides a set of predefined selection rules which reflect the quality of the application systems' development state: the default version selection rule *developing* is used for building test systems, *stable* is used for building systems, that are known to be consistent, and, finally, *release* is used for (re-)building released systems. The variant definitions provided by the management system are variant classes for different compilers (pcc and GNU CC), different compilation "qualities" (like debugging, profiling, or optimization), and different operating system platforms (4.3BSD, SunOS, etc.).

3. The Software Release Process

The software release process is a complex activity with different characteristics depending on project policies. Establishing a release means to synchronize and verify the development stages of all components that are part of it, i.e. making sure that all components work and cooperate consistent, reviewing and evaluating the programmers' work results against specifications and documentation, rejecting unsatisfying results, etc. Most of these activities have to be carried out by humans and require a high degree of conscientiousness and responsibility. In small projects with a high degree of communication between

2 This is of course language specific.

programmers (like ours) a less rigorous release process is certainly more appropriate as in larger projects where a severe quality assessment phase must ensure that the objectives and goals of a given release are actually achieved. This, however, requires a strict framework of change requests, baseline planning and procedures to verify sets of changes in order to make sure that all objectives are met. The described product management system does not offer automatic support for these activities, but is a rather basic mechanism ensuring a minimal set of constraints for preparing and establishing a release. Nevertheless AFS provides basic means to support even higher order activities.

Every time a programmer thinks her software system is suitable for a new release, she simply types *shape release*. During the release process the modified components are promoted to a safe, internal AFS-state to prevent unintentional deletion or change. The most recently saved component versions are then marked with a unique release name to allow easy access of these versions using the release name. A unique configuration identification document for *exactly* this release is constructed and saved together with the other documents of the release.

The release procedure is performed in four steps followed by an optional build- and install step.

Step 1: Quality Assessment

As described above in this step a quality assessment phase has to validate that certain objectives for a given release are achieved. Actually this step is in our case rather basic. Currently it verifies and ensures that no modification of a component part of a release is pending or active. Because several programmers can develop simultaneously on the same software system (but on different components) in their private workspaces, a programmer might not notice the activities of her colleagues and can mistakenly activate a software release.[3] The management systems recognizes simultaneous development by means of locks that a developer implicitly holds when she had started with her modification. If there exists locks on components the management system prevents a release process and signals an appropriate message to the initiator. For convenience the programmer, who triggers the release procedure, may hold locks.

Afterwards all component versions are locked during the release procedure, to avoid further developments and concurrent release generations.

Step 2: Release Number Generation

In the second step a new release number is generated by automatically creating and saving a brand new version identification document (VID). The release name is a combination of the VID's name and a version number, for example *vlversion-3.12*. As described above the VID can be used by the application system to identify its version at runtime. For the C programming language environment a VID contains a single function which returns a version identification string. The string is build up of a version number and version state, the name of the author, and the time when the VID was created.

A newly created VID looks like this:

```
char *vlversion () {
   static char Confid[] =
          "$__version$ [$__state$] ($__date$ by $__auuid$)";
   return ConfId;
}
```

The words enclosed by dollar signs are candidates for attribute citations and trigger Shape (and the version control programs) to substitute attribute citations by their current values when the version is retrieved from AFS. The function's name is statically derived from the macro VERSIONID. *Version*, *state*, and *auuid* are AFS related attributes and represent the version number, version state, and the author's name respectively. A fully expanded VID may look like:

```
char *vlversion () {
   static char ConfId[] =
          "4.17 [published] (Wed Mar 21 16:39:40 1990 by uli@coma)";
   return ConfId;
}
```

On request the program *vl* identifies itself and the used AFS related libraries in the following manner:

3 The creation of a release should really be the system integrators' job.

```
This is vl version 4.17 [published] (Wed Mar 21 16:39:40 1990 by uli@coma).
AFS toolkit lib version 1.32 [published] (Thu Jun 14 13:52:42 1990 by uli@coma).
AFS version 1.38 [published] (Fri Jun  1 16:31:36 1990 by andy@coma).
```

Step 3: Publishing

During the third step the component revisions belonging to the release are set to state published to indicate a milestone in their development line. Furthermore all locks are released and the previously constructed release name is attached to them as an AFS attribute. This special attribute is known by the version control commands and allows easy access to all components of a certain release. For example, the command

```
vl -n vlversion-4.17
```

identifies and prints the set of components of the release labelled with the symbolic name *vlversion-4.17*:

```
Makefile[4.6]    deps[4.1]    vl.c[4.12]       vlversion.c[4.17]
Shapefile[4.6]   vl.1[4.5]    vl.cid[4.17]
```

In combination with the version retrieval command *retrv* the set of components can be trivially restored from the object base with the command sequence

```
retrv 'vl -n vlversion-4.17'
```

Step 4: Configuration Identification

In the last step of the release procedure Shape is called recursively to construct a unique configuration identification document (CID). In the recursive call Shape is forced to select only the recently published versions of each component by triggering the selection rule *release*:

```
release:
        *, attr(state, published), attrmax(version);
        *, attr(state, busy).
```

This generic selection rule states that shape should select primarily the latest saved version with state *published* of each component in question (indicated by the asterisks). If Shape cannot find a published version, it signals a warning message and tries to select a busy version. Shape terminates, if neither a published nor busy version of a certain software object exists. The last alternative of the selection rule is in fact necessary for software objects not under control of the version control system such as object code libraries and standard C header files.

The resulting CID contains all information specified in the Shapefile including the Makefile and the definitions of the product management system. It differs from the Shapefile in that all user supplied selection rules are replaced by a new selection rule which uniquely identifies the appropriate component versions. In addition, a new main target which enables the new selection rule and certain variant definitions is added. A unique selection rule looks like:

```
#% RULE-SECTION
@vl:
        vl.c,
        attr(generation,4),
        attr(revision,12),
        attr(state,published);

        vlversion.c,
        attr(generation,4),
        attr(revision,17),
        attr(state,published).
#% END-RULE-SECTION
vl: @vl +... vl.o vlversion.o
```

In this example */u/ulip/shape/vc/vl/vl.c* version 4.12 and */u/ulip/shape/vc/vl/vlversion.c* version 4.17, which both have the state *published*, have been chosen to build a configuration.

Step 5: Building

The release process is complete after the fourth step: all software components are in a stable, immutable state, a configuration has been identified, and an exact and complete system building instruction in form of a configuration identification document has been generated. In an optional build step the CID can be used to build and install the configuration.

4. Conclusion

While the described management system is rather complex, it is very easy to use. Shape- and Makefiles need rarely to be modified. In fact, they are checked into the version control system most of the time and do not need to exist as busy versions because Shape is able to retrieve all the necessary data automatically from AFS.

The easy use of the software management system considerably simplifies the daily work in that it provides standardized functionality for software development and maintenance. While writing the management system and testing our toolkit (namely shape) with other, foreign software floating around in the USENET, we became aware that it is relatively easy to switch from Make to Shape. The Makefile provided by most software systems has only to be included into a Shapefile which enables Shape's powerful version- and variant selection mechanism.

The software release process described here is certainly suitable for small projects with a high degree of communication between developers, where the development state of a system can be determined mostly informally. However, it is possible to program a different behaviour of the release process for larger projects, for example making sure that all latest revisions have undergone a quality assessment process. While Shape and the related version control commands are capable of performing the basic configuration- and version control management, the actual implementation of appropriate higher level system management functions requires a good deal of craftsmanship.

Nevertheless, there are still many edges where further work has to be done. The system currently does not support programming language environments other than the C programming environment. While designing the described software management system, it became clear that a much more systematic way to describe a particular project setup is needed. The use of object oriented techniques to define software process object properties and a corresponding command language appears very appealing. Furthermore, a more powerful system modeling language and version selection mechanism is necessary to describe complex software systems and configurations.

5. Acknowledgements

Acknowledgement is made to my colleagues Jens Brüning, Andreas Lampen, Axel Mahler, and Wolfgang Obst for their contribution to the work described in this paper. I gratefully acknowledge the CEC in the REX Project (2080) for their financial support.

6. References

[APO85a] APOLLO, *Domain Software Engineering Environment (DSEE) Reference,* Apollo Computer Inc., Chelmsford MA. (July 1985).

[Fel79a] Stuart I. Feldman, "Make – A Program for Maintaining Computer Programs," *Software - Practice and Experience* **9**(3), pp. 255-265 (March 1979).

[Lam88a] Andreas Lampen and Axel Mahler, "An Object Base for Attributed Software Objects," *Proceedings of the Fall 1988 EUUG Conference*, Lisbon, Portugal, pp. 95-106, European Unix systems User Group (October 1988).

[Mah88a] Axel Mahler and Andreas Lampen, "An Integrated Toolset for Engineering Software Configurations," *Software Engineering Notes, Vol. 13, No. 5*, Boston, Mass. **24**(2), pp. 191-200, ACM Press (November 1988).

[Tic85a] Walter F. Tichy, "RCS - A System for Version Control," *Software - Practice and Experience* **15**(7), pp. 637-654 (July 1985).

Examples

The examples consists of excerpts of the Shape- and Makefile of the program *vl* which is a part of the shape toolkit. In addition, some parts of the management system discussed in this paper are provided.

The Shapefile

```
# Shapefile for vl
#
# $Header: Shapefile[4.6] Thu Dec 21 14:54:20 1989 uli@coma published $
#

RULE=developing
SYSTEM=bsd_4_3
COMPILER=pcc
QUALITY=debug
PURITY=

$(PROGRAM): $(RULE) +cfflags +$(OTHER) +$(SYSTEM) +$(COMPILER) \
        +$(QUALITY) +$(PURITY)

install: $(RULE) +$(SYSTEM)

include Makefile

SHAPEINCPATH = $(BASE)/lib/shape
include $(SHAPEINCPATH)/afsmacros
include $(SHAPEINCPATH)/stdmacrodefs
include $(SHAPEINCPATH)/stdtargets
include $(SHAPEINCPATH)/stdrules
include $(SHAPEINCPATH)/stdvars
include $(SHAPEINCPATH)/stdconf

include deps
```

The Makefile

```
# Makefile for vl
#
# $Header: Makefile[4.6] Wed Jan 17 17:35:08 1990 uli@coma published $
#

BASE = /u/shape
CONFIG = s-bsd_4_3
INSTALL = /usr/bin/install $(INSTALLOPTIONS)
INSTALLOPTIONS = -c

INSTALLBINDIR = $(BASE)/bin
INSTALLLIBDIR = $(BASE)/lib
INSTALLINCDIR = $(BASE)/src/inc

SRCPATH = $(BASE)/src
INCPATH = $(BASE)/src/inc
CONFPATH = $(SRCPATH)/conf/$(CONFIG)
LIBPATH = $(BASE)/lib
AFSLIB = $(LIBPATH)/libafs.a
AFSTKLIB = $(LIBPATH)/libafstk.a
AFSLIBS = $(AFSTKLIB) $(AFSLIB)

CFLAGS= -I$(CONFPATH) -I$(INCPATH) -g
xflags = -DCFFLGS='"$$Flags: <$<> $(CFLAGS) $$"'
CC = cc $(xflags)

COMPONENTS = $(SOURCES) $(HEADERS) $(MANUALS) $(SYSDEFS)
SOURCES =  vl.c $(VERSIONID).c
HEADERS =
OBJECTS = vl.o $(VERSIONID).o
MANUALS = vl.1
PROGRAM = vl
VERSIONID = vlversion
SYSDEFS = Shapefile Makefile deps
```

```
$(PROGRAM): $(AFSLIBS) $(OBJECTS)
        $(CC) $(LDFLAGS) -o $(PROGRAM) $(OBJECTS) $(AFSLIBS) $(SYSLIBS); \
        rm -f vlog pattr; \
        ln $(PROGRAM) vlog; \
        ln $(PROGRAM) pattr

install: $(INSTALLBINDIR)/$(PROGRAM)

$(INSTALLBINDIR)/$(PROGRAM): $(PROGRAM)
        touch $(INSTALLBINDIR)/$(PROGRAM); \
        chmod 755 $(INSTALLBINDIR)/$(PROGRAM); \
        $(INSTALL) $(PROGRAM) $(INSTALLBINDIR); \
        /bin/rm -f $(INSTALLBINDIR)/vlog $(INSTALLBINDIR)/pattr; \
        /bin/ln $(INSTALLBINDIR)/$(PROGRAM) $(INSTALLBINDIR)/vlog; \
        /bin/ln $(INSTALLBINDIR)/$(PROGRAM) $(INSTALLBINDIR)/pattr;

clean:
        rm -f $(PROGRAM) vlog pattr $(OBJECTS)
```

The Software Release Task Description

```
# $Header: stdconf[1.12] Fri Jan 19 20:42:05 1990 shape@coma published $

release: _checklock _newrel _publish _confid
        @echo "Definition of new $(PROGRAM)-release complete."; \
        echo; echo; echo

_checklock:
        @echo "Trying to lock $(PROGRAM)-components for publication..."; \
        if vfind $(COMPONENTS) -force -locked ! -locker `whoami` -exit 1; \
        then \
                vadm -q -lock $(COMPONENTS); \
                echo "Components locked for publication."; \
        else \
                echo "Some components locked by someone else!"; \
                echo "Release attempt failed."; \
                exit 1; \
        fi

_newrel: _makevid
        @sbmt -fq -lock $(SBMTFLAGS) $(VERSIONID).c
        @echo Generating Release "`vl -# -y $(VERSIONID).c`"

_publish:
        @echo "Publishing components."; \
        sbmt -q $(SBMTFLAGS) $(COMPONENTS) \
             -n "$(VERSIONID)-`vl -# -y $(VERSIONID).c`"

_confid:
        @-echo "Generating configuration identification."; \
        if vl -q -ss+ $(PROGRAM).cid; then \
                vadm -lock -q $(VERSIONID).c; \
        fi; \
        touch $(PROGRAM).cid
        @-$(SHAPE) -n -confid $(PROGRAM) RULE=release
        @-if vadm -q -lock $(PROGRAM).cid ; \
                then \
                        sbmt -q $(SBMTFLAGS) $(PROGRAM).cid \
                        -n "$(VERSIONID)-`vl -# -y $(VERSIONID).c`"; \
                else \
                        echo "WARNING: couldn't lock $(PROGRAM).cid"; \
        fi
```

```
#$__xpoff$
_makevid:
        @-echo "Making version identification document $(VERSIONID).c." ; \
        if vl -q -ss+ $(VERSIONID).c; then \
          retrv -lock -fq $(VERSIONID).c; \
        else \
          rm -f $(VERSIONID).c; \
          echo '/*$$__copyright$$ */' > $(VERSIONID).c ; \
          echo 'char *$(VERSIONID) () {'  >> $(VERSIONID).c ; \
          echo '  static char ConfID[] =' >> $(VERSIONID).c ; \
          echo '  "$$__version [$$__state$$] ($$__stime by $$__auuid$$)";' \
               >> $(VERSIONID).c ; \
          echo '  return ConfID;' >> $(VERSIONID).c ; \
          echo '}' >> $(VERSIONID).c; \
          vadm -q -setc " * " $(VERSIONID).c; \
          vadm -q -setuda copyright=@$(COPYRIGHT).c $(VERSIONID).c; \
        fi

#$__xpon$
buildrelease:
        @-if vl -y $(PROGRAM).cid; \
        then \
                $(SHAPE) -rebuild $(PROGRAM); \
        else \
                $(SHAPE) $(PROGRAM) RULE=release; \
        fi
```

The Rule Selections

```
# $Header: stdrules[1.10] Thu Dec 21 15:12:16 1989 uli@coma save $
#% RULE-SECTION
developing:
        *, attr(state,busy),
           msg(Using busy version of $+.);
        *, attrmax(version),
           msg(Using last saved version of $+.).

stable:
        *, attr(state,saved), attrmax(version),
           msg(Using last save version of $+.);
        *, attr(state,published), attrmax(version),
           msg(Using published version of $+.);
        *, attr(state,busy),
           msg(Using busy version of $+.).

last:
        *, attrmax(version),
           msg(Using last version of $+.);
        *, attr(state,busy),
           msg(Using busy version of $+.).

symbolicname:
        *, attr(__SymbolicName__, $(SYMNAME)),
           msg(Using $+ with symbolic name $(SYMNAME)).

release:
        *, attr(state,published), attrmax(version),
           msg(Using last published version of $+.);
        *, attr(state,saved), attrmax(version),
           msg(Using saved version of $+.);
        *, attr(state,busy),
           msg(Using busy version of $+.).

#% END-RULE-SECTION
```

The Variant Section

```
# $Header: stdvars[1.6] Thu Jan 11 17:26:16 1990 shape@coma published $
#% VARIANT-SECTION

vclass system ::= (bsd_4_3, sunos_4, tpix)
vclass compiler ::= (pcc, gcc)
vclass quality ::= (debug, optimize)

# vclass system
bsd_4_3:
        CONFPATH = $(SRCPATH)/conf/s-bsd_4_3
        CFLAGS= -I$(CONFPATH) -I$(INCPATH) $(SYSLOG)
        INSTALL = /usr/bin/install $(INSTALLOPTIONS)

sunos_4:
        CONFPATH = $(SRCPATH)/conf/s-sunos_4
        CFLAGS= -I$(CONFPATH) -I$(INCPATH) $(SYSLOG)
        INSTALL = /usr/bin/install $(INSTALLOPTIONS)

tpix:
        CONFPATH = $(SRCPATH)/conf/s-tpix
        CFLAGS = -I$(CONFPATH) -I$(INCPATH) $(SYSLOG)
        SYSLIBS = -lPW
        INSTALL = /etc/install $(INSTALLOPTIONS)
        INSTALLOPTIONS = -f $(INSTALLBINDIR)

# vclass compiler
pcc:
        CC=cc
gnu:
        CC=gcc -traditional -W

# vclass quality
debug:
        CFLAGS=-g

optimize:
        CFLAGS=-O

#% END-VARIANT-SECTION
```

A Multi-Site Software Development Environment

M. O. Pierron
P. Zerlauth

Alcatel Business Systems
Software Technical Department
1, route du Dr. Albert Schweitzer
67408 Illkirch Cedex
France

ABSTRACT

Atelix has been the operational distributed software development environment used by Alcatel Business Systems, Strasbourg for software development since 1986. In 1989, a multi-site version of the environment (Atelix/multi-site) was introduced. Atelix/multi-site is an extension of Atelix. The aim of this article is to set down the features of Atelix: configuration management, project management and network transparency. It will describe the characteristics and the setting up of the Atelix/multi-site software development environment.

1. Introduction

Alcatel Business Systems is the leading European manufacturer of business communication networks. It is part of the ALCATEL group which is mainly located in Europe.

The products developed by Alcatel Business Systems (PABX, KeySystems, telephone units, fax machines, videotex etc.) are constantly increasing in sophistication, with programs growing in size and in number of features offered. Since 1988, the policy has been to use several ALCATEL units, working in collaboration, for software development.

This document presents the Atelix/multi-site software development environment, used to develop products in a multi-site context that is the joint development of a product on several sites located at some distance from each other.

To understand the mechanisms of Atelix/multi-site, the main features of the local software development environment (Atelix) will be described in the first part of this article. The second part will explain the behavior and the implementation of Atelix/multi-site.

2. Atelix: a development environment

2.1. Description

Atelix is a distributed software development environment. It is composed of a set of tools which are involved in the various phases of the life cycle for software development. Its strong point is in the fact that the set of tools for the detailed design, programming, testing and maintenance phases are all fully integrated. Atelix makes the developer's daily work much easier:

- it renders certain tasks automatic.
- it ensures the coherency of the project by making certain functions transparent for example configuration management.
- it allows project management.

Atelix belongs to the first and second generation of IPSE (Integrated Project Support Environment). It provides integration facilities for individual tools used by the project.

The developer can concentrate on his principal task – software development.

Atelix was designed and produced by Alcatel Business Systems in Strasbourg in 1985. It has been in operation since 1986. It supports the development of large programs. It has been used, for example, in a project where 60 people produced over 1 million lines of source programming.

Atelix is based on software available on the market such as Informix (relational database management system), Mosaix (configuration management system) and the UNIX platform.

The rest of this section will describe the Atelix function for the detailed design, programming, testing and maintenance phases.

2.2. Features

Atelix features (for the detailed design, programming, testing and maintenance phases) meet two major needs in terms of software development: configuration management and project management. These can be accessed using a small number of commands which are easy to use from a character-based terminal interface. In addition, the network on which Atelix is running is rendered transparent to the user.

2.2.1. Configuration management

The software developed by Alcatel Business Systems is made up of a large number of source files (from about 100 to 5000) and object modules etc.

During the integration phase, the most recent object modifications, that is to the source files, object modules, etc, must be used to ensure release coherency, and production must be fully optimized, building modified objects only. During the maintenance phase, it must be possible to take an old release and create a fully optimized new release from it. Atelix does this by dynamically controlling the software dependency tree and the various versions of the objects of which the software is composed.

Configuration management is carried out on various types of object:

- **derived objects** – these are created from other objects using tools, for example object modules or binary modules.
- **non-derived objects** – these are created by the user, using the editors, for example the source files.
- **entities** – these have the same properties as non-derived objects. These objects have the following features:
  - o they contain the data shared between the various software modules, chiefly the interfaces, constants, definitions, etc.
  - o they are included in a large number of non-derived objects.

Configuration management is transparent to the users once the objects have been declared to the configuration manager.

2.2.2. Project management

A project is developed by a team. Each member of the team develops part of the project and he is responsible for it. In order to do this, he must be able to work without disturbing the work of the other members of the team, while retaining the ability to share information. At any given moment, the team must be able to build a software release. Such action would be the responsability of an administrator normally the project leader. In order to support the management of a team, Atelix offers an organization which is based on a series of spaces (see Figure 1):

- **the development space** – this is the work space for one person or group of people, for the development and testing of part of the software. The project defines the number of development spaces it needs. It attributes the responsibility for each space to one or more persons. Each development space is isolated from the other spaces, and cannot be seen from another space.
- **the share space** – this is the data (objects) sharing space for all the members of the team. This space is unique to the project, and the team is responsible for it. Each member of the team modifies the objects for which he is responsible, using the Atelix commands.
- **the reference space** – this is the production space for the software releases using the configuration manager. There is only one such space for the project and it is controlled by an administrator.

A set of commands enables the objects to transit from one space to another. This is carried out for:

- creation of a new version of a non-derived object within the reference space from the modified object in the development space.
- making a derived object available in the share space from the object made in the development space.

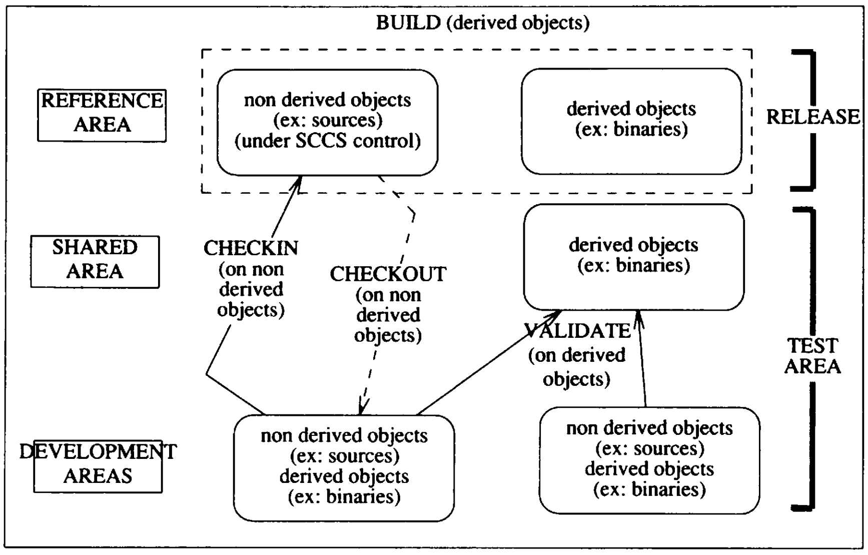

BUILD (derived objects)
REFERENCE AREA
non derived objects (ex: sources) (under SCCS control)
derived objects (ex: binaries)
RELEASE
SHARED AREA
CHECKIN (on non derived objects)
CHECKOUT (on non derived objects)
derived objects (ex: binaries)
VALIDATE (on derived objects)
TEST AREA
DEVELOPMENT AREAS
non derived objects (ex: sources) derived objects (ex: binaries)
non derived objects (ex: sources) derived objects (ex: binaries)

Figure 1: *Atelix environment*

Derived objects can be made:

- in the development space, from objects located in one of the three types of space. Objects are searched for, in a predefined order, under the control of Atelix. They are made without the configuration management system control, that is without any version management and without necessarily using the most recent version of the objects.
- in the reference space from objects located in the reference space, under the control of the configuration management system.

2.2.3. Network management

Atelix runs on a network of heterogenous machines made up mostly of Sun, Sun386i and PCs (see Figure 2). It is based on NFS and the RPC/XDR procedures. Configuration and project management are carried out on Sun. In most cases, the production tools, that is the compilers, link editors and so on, are on a Sun386i or a PC.

This network configuration is transparent to the user. Its data and processing are distributed over the network as needed.

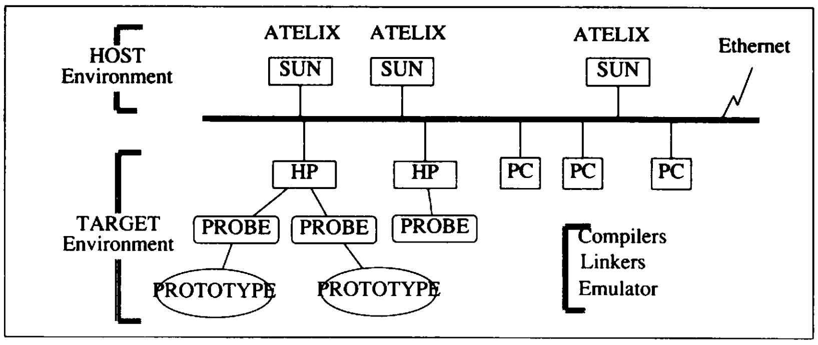


Figure 2: *Alcatel Business Systems network for software development*

3. Multi-site development environment

3.1. General points

In 1988, it was decided that several Alcatel units, from different countries and towns located far apart, should participate in the development of the same product, without centralizing the development geographically.

The software for a product is developed by a team spread over several sites. To make this possible, a multi-site version of Atelix was developed by Alcatel Business Systems in Strasbourg, in co-operation with SEL-Alcatel in Stuttgart.

Configuration and project management are spread over the various sites where the team is located. Data exchange between the sites consist of the various objects managed by Atelix. Atelix/multi-site is an extension of the "local" version of Atelix. The developer has the same set of commands at his disposal. The local behaviour of the commands is the same as with the "local" version of Atelix. Moreover, the commands ensure that software development in a multi-site context is managed in a transparent manner.

The rest of this section will describe the context of multi-site software development and the implementation of Atelix/multi-site.

3.2. Definitions

At Alcatel, products are developed at one or more sites. Within this context:

- **site** – A site is a geographical location including a set of machines linked via a local network (see Figures 2 and 3).
- **multi-site product** – A multi-site product is a software for which the team is spread over several sites.

At any one site, several multi-site products can be developed simultaneously.

3.3. Inter-site connection

All the sites taking part in the development of a multi-site product are interconnected in pairs (see Figure 3).

The link between two sites can be of several types, including X25, leased line, Ethernet, etc. Whatever the type of link used, a software layer provides the following features for the rest of the application:

- file transfer with checking and preservation of the access rights (command: transfer).
- remote execution of the commands (command: remote_execute).

This layer is based on the INR (Inter Network Routines) package supplied by SUN.

For security reasons, a site can only know the remote sites through servers which support the connection, and access to the remote sites is limited by predefined commands.

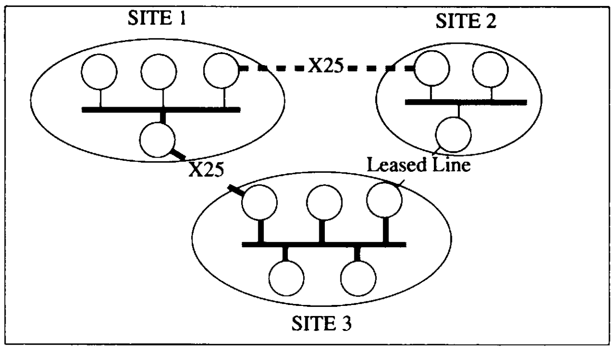


Figure 3: *Connection between sites*

Moreover, security restrictions enable the sites to remain independent. Each site controls its own local network. The network extended over the various sites is not transparent. Connection to a remote site requires use of a password. However, using the Atelix/multi-site application, the extended network is fully transparent.

3.4. Basic principles and restrictions

In consultation with the development teams, a number of principles have been defined. These concern the method of working, and frequency and the reasons for updating at the various sites.

- *1st principle - general philosophy.*

The development environment for a multi-site product must support all the features of the "local" version of Atelix: project management, configuration management and network transparency.

- *2nd principle - data management.*

The Atelix data are classified according to three types of object – the derived and non-derived objects and the entities (see paragraph 2.2.1). All data are recopied to all the participating sites.

However, each data item is controlled by one site, and can be modified at this site, local site. Indeed, in Atelix, each data is attached to a user (he alone can modify it). Data will be modified on the site on which the user works.

The data item is then updated automatically at the other, remote sites.

- *3rd principle - the frequency of updating data.*

The working method used in the teams introduces a different frequency for updating the objects on the sites according to their type:

  o **derived objects** – the team makes an average of one test version of its software per day with the latest modifications carried out on the objects. The derived objects from all the sites must be present at each site when the test version is being made. This type of object is updated daily at the remote site.

  o **non-derived objects** – the team makes a release of its software, under the control of the configuration management system in order to identify the significant stages of development. The non-derived objects on all the sites must be present when a software release is being produced.

  o **entities** – the importance of their content (interface, constants, etc.) for the software requires an identity on all the sites. All the sites must be updated immediately.

- *4th principle - condition for updating data on the remote sites.*

Atelix/multi-site is based on the project management feature of the local version of Atelix. There are three types of space at each site (see paragraph 2.2.2): the development, share and reference spaces. The development space is located on a site; it is the responsibility of one user. The share and reference spaces are copied to all the sites. The team only has one share space and one reference space, which is common to all the sites. Atelix/multi-site maintains the coherency of copying the share and reference spaces. In order to do this, the commands which modify the content of the share and reference spaces are re-executed on the other sites. They correspond to declarations made to the configuration management system, transfers from one space to the other and and the removal of objects.

Updating of the data at another site is transparent to the developer unless he makes an explicit request. An administrator is responsible for following up the results of a command. He has a system at his disposal for tracing the execution and recovering from errors.

3.5. Implementation

3.5.1. Principle of updating the sites

All the Atelix commands which modify the content of the share or references spaces at a site, will be re-executed at the remote sites in the same order. This action is named a repercussion of commands.

Re-executing the commands at the remote sites is necessary to ensure that configuration management is automatic and transparent. Merely recopying the content of the objects is not enough. Automatic and transparent management of the dependency relations is provided as in the "local" version of Atelix. The execution sequence in local is guaranteed on the distance sites in order that object versions are in agreement at all sites.

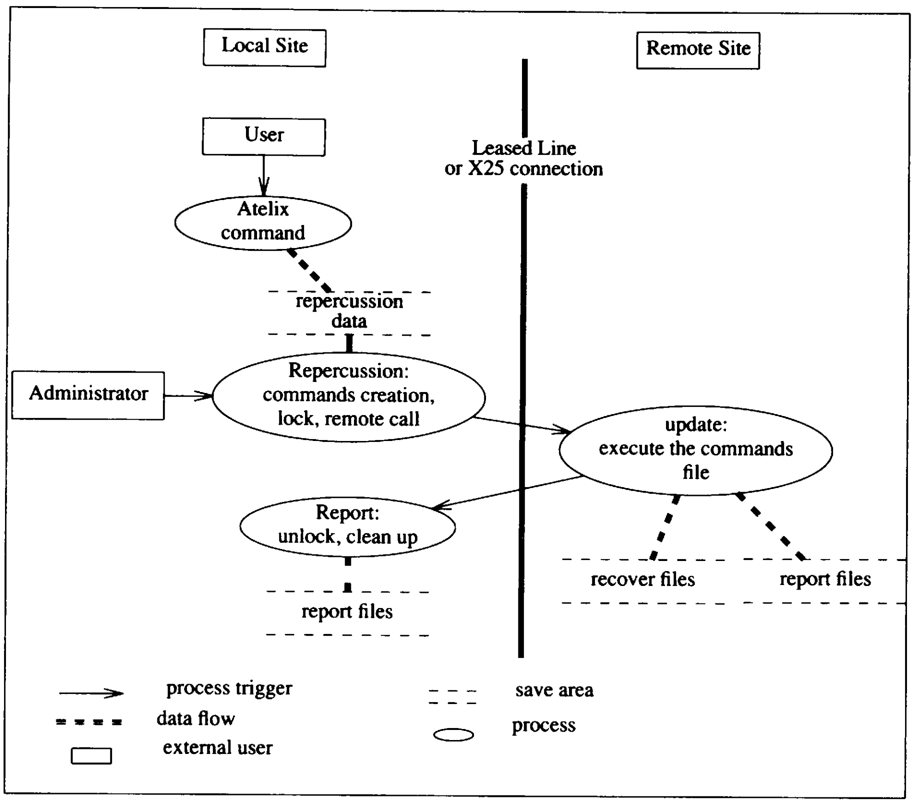


Figure 4: *Atelix/multi-site delayed repercussion*

There are two types of repercussion for the commands: immediate repercussions and delayed repercussions. An Atelix command has immediate repercussions on the remotes sites if:

- it deals with an entity.
- it concerns data transfer of development spaces towards the share space, if the user has explicitly requested it.

All the other commands have delayed repercussions, that is, they are stored in memory and then re-executed on the remote sites.

Various stages of delayed repercussion: (see Figure 4)

- The developer executes an Atelix command for an object, on the site which is responsible for the object. This site is called the local site for the object, the other sites for this multi-site product being called remote sites.
- The command is saved in a buffer area together with the information required to re-execute it on the remote sites, that is, its parameters and input files.
- The administrator sends the repercussion of all the commands stored in the buffer area to the remote sites. On the local site, the repercussion:
  - o creates a file containing the commands to be executed at the remote sites.
  - o locks the objects for which a command must be executed at the remote sites. Locking objects is necessary to ensure the execution sequence of the commands.
  - o runs the execution of the command file at all the remote sites.
- The command file, which has been prepared beforehand, is executed at all the remote sites. An execution report is created in parallel.
- Each remote site notifies the local site when execution has been completed. The execution report is transfered.
- When all the remote sites have indicated the end of execution, the objects are unlocked and the buffer area is cleared.

It is virtually the same for an immediate repercussion. The major difference is that when the command is executed, the repercussion is run immediately without any intervention on the part of the administrator.

3.5.2. The tracing, reliability and error recovery system

For a number of reasons, the repercussion system may either end with an error or may be interrupted. A mechanism for execution tracing and recovery from errors allows the cause of the problem to be found and the repercussion session to be resumed.

3.5.2.1. Trace

At each site, the administrators can access information related to their own site (repercussion of commands sent to other sites and repercussions of commands received from other sites). There are two tracing mechanisms: mail and the report files.

- **mail** – in the case of a problem, a message is sent to the administrators of the sites affected by the problem. This is also sent to the user concerned with the command, during an explicit repercussion request (immediate repercussion). Its content gives the location and time of the error and also the identity and type of object in question.
- **the execution report** – when the command file is being executed (at a remote site), an execution report file is created. It contains a trace of the execution for each command – identification of the object concerned with the command, the result, execution date etc. Moreover, it contains statistical data and global data for all the commands. It is stored on all the sites involved with the repercussion.

3.5.2.2. Reliability

To maintain the coherency of the data at all the sites, two mechanisms have been set up

- delayed re-execution of all commands having immediate repercussions. This means the execution sequence for the commands is maintained at the remote sites.
- locking objects. The objects are locked at the local site, during the repercussion time. Unlocking becomes effective if the repercussion was carried out successfully at all the remote sites.

3.5.2.3. Error recovery

The problems encountered when using Atelix/multi-site can be put into two categories:

- those relating to a set of sites – during the repercussion request, the dialogue between the sites is interrupted (link-up problem, power supply cut-off, machine shut down etc.).
- those relating to one site – during a repercussion, the execution of one or more commands (command file) may fail (problem with system, insufficient disk space, process table incomplete, incoherency of the Atelix data etc.).

When a problem occurs, the system stores all the information required to resume a repercussion. Resumption is possible after removing the cause of the problem. Parallel to this, a message is sent to the administrators of the various sites concerned and a report is created. Due to all the information stored in the report, the cause(s) of the problem(s) can be suppressed. A set of commands is available to resume the repercussion.

3.6. Roles within Atelix/multi-site

For the development of a multi-site product, there are two major types of activity – the true development of the product and the administration of this product at the various sites. We can therefore define two roles:

- **The developer:**

His role is to design and produce the multi-site product. For a developer who is accustomed to working in the Atelix environment, the multi-site appearance is transparent. However, for the derived objects he can modify the frequency of updates for the remote sites. He can make an explicit request for a command to have repercussions on the other sites immediately. By default, these objects are updated in the delayed manner (daily). In the case of objects which are updated immediately, the developer receives a message if the repercussion fails. His only role is to warn the administrator of the repercussion failure.

- **Administrator:**

There is one administrator per site and per multi-site product. His role and his power to act are limited to his own site. His major tasks are:

- o setting up the environment on his site in collaboration with the administrators on the other sites.
- o running the delayed repercussions and following them up.
- o recovering from problems. Using the execution report and the message, he determines the cause(s) of the problem. If the problem is at his own site, he deals with it, otherwise he contacts the administrator on the site concerned. When all the problems have been sorted out, he resumes the repercussion using the commands available.

4. Assessment

A first version of Atelix/multi-site was implemented in May 1989 by Alcatel Business Systems Strasbourg and SEL-Alcatel Stuttgart, within the context of developing a KeySystem.

The software team is composed of 20 people (10 people at each site). A leased 9600 baud telephone line links the two sites.

Each day, about 60 commands are re-executed on average at all the sites.

5. Conclusions

Atelix/multi-site provides a software development environment which is satisfactory today for the downstream phases of the detailed software design. The development of the multi-site system is an absolute necessity in the international context in which we operate.

The directions we are currently taking are:

- extension of the multi-site environment to the upstream phases in the life of a project (external specifications for the system, design etc.).
- provision of a software development environment which is based on standards: PCTE, CIS, CFI,....
- creation of a LAN multi-site.

6. Acknowledgements

We thank everyone who has participated in the development of the Atelix/multi-site version (especially: J.Omméja, O. Boehm, B. Fouarge, O. Mellé, F. Millotte), everyone who has read and commented on this article (especially: U. Klug) and all the team, who have had both the privilege and the patience to use the environment in its very first release.

Lynx: The Object-Oriented Inode Eater

Andy Bartlett

Imperial Software Technology Ltd
95 London Street
Reading RG5 1QA, UK
ab@ist.co.uk

Greg Reeve

White Horse Computing Ltd
11a Charlieu Avenue,
Calne, Wiltshire, SN11 0ZD, UK
greg@ist.co.uk

ABSTRACT

When development trees proliferate, source management has little to do with the logistics of the enormous amounts of disk space required to support everyone's personal development environment. Integration becomes a nightmarish mix of version skew and "first night" release nerves. When all the parts of the software are returned to the master sources, will they even clean build, let alone run as they had before the developer handed them over?

Lynx applies a chain-saw to this forest of Amazonion proportions.

Introduction

Configuration Managers want it both ways. They need interworking and co-operation. Togetherness is a key ingredient in a successful product release. But they also need rigid and strict controls. Without the controls in place, even he most trivial disaster could set the project back beyond its deadline. Lynx tackles this schizophrenic requirement at the UNIX level, complementing rather than attempting to supersede SCCS.

Lynx is a general purpose tree manager intended to facilitate any Configuration Management interface – from the informal "CM by consensus" to a fully regulated system like ISTAR-CM. Lynx by its nature is co-operative, and relies on people working together. A CM system is used to enforce co-operation, and provides the policing and monitoring that is necessary on large projects. It also enforces local roles. Who has the right to tell you to develop some software? Who is responsible for saying whether it works after the development has finished? Who is prepared to say that integrating a new component isn't going to jeopardise an upcoming delivery? Administrative applications like ISTAR-CM takes care of these issues, leaving day to day source management to Lynx.

This report is about source-sharing in an environment where users acquire sources by inheriting them. Lynx started off as a tool to unfold this extra dimension, and allow developers access to shared code. It grew into a product that made software integration a non-event.

Motivation

A post-hoc account of Lynx would go something like this:

> The development/integration cycle was out of control. Integration, testing and release were high risk activities critically dependent on developers having "followed the rules". Had they remembered to SCCS the latest version of every source file? Had they forgotten to include key files in the archive they handed over? Had more than one developer made changes to the same file? Had anyone overwritten the latest version of a file with his own personal (old) copy? Had they used some non-transferable features of their local environment?

> *We reasoned* that integration was a no-win, wasteful activity – whose existence is only justified by the untidy mess that preceded it. It was the diaper on the back end of software development. And it was an unreasonable administrative labour.
>
> *We reasoned* that integration made system testing even more impossible to achieve. Not only were modifications to be exercised and broken by the tester. He must also show whether the re-integrated product has effectively merged the development changes into the original.
>
> The target had to be an environment where everyone works co-operatively, **as if** they were all working in a single software tree, yet without the shifting baselines and chaos that would probably ensue. Integration would breed no dragons if we could reach a position where a snapshot or tape cut from development areas was guaranteed to be identical to a post-integration snapshot or release tape. If this were the case, testing (again **as if** the software had been re-integrated) could take place at any time and as often as you need to check on the state of the project. It would also be unnecessary to wait for integration before starting system testing, and the testers would no longer have the added burden of proving that the integration hadn't introduced version skews across the components.

Software advances just don't happen in a meticulous and planned way. You can't plan and schedule a visit to El Dorado like you can a business trip to New York. In fact, we stumbled into the right frame of mind, had a bit of luck and ran with it. A lucky early guess wasn't planned into a safe corner of the project, but allowed to grow and develop as we learned how to exploit it. Its the stuff that ulcers are made of.

Customer X, like all customers, wanted the pot of gold. Instead he had to make do with a CM system built around Lynx. We hope he finds it as useful as we do. Immediately after the initial release we adopted it for all our software management. That was back in March. Our luck held (we're still here), and software integration has become more of a state of mind than a traumatic activity. Something we find difficult to explain to QA.

Background

Customer X came to us with an intriguing set of requirements to support a critical internal project. Firstly the standard fare: they needed to provide comprehensive configuration management and they also wanted to use the Pmak Build Tools [Til86a]. Then the rub. They wanted transparent and effective source sharing. With a considerable number of releases planned, and a range of variants within each release, they had to have a mechanism that would not proliferate sources. And had we come across the 1984 Bell Technical Report on Build [Eri84a].

Build was a variant of make, which did not require all the sources necessary for building software to be present in a developer's local environment. It worked through an environment variable called VPATH – the Viewpath. Where shells use PATH to resolve a command, Build used the VPATH variable to resolve which tree a build component was to come from. Customer X thought that the Viewpath would provide the control they were after, but they would prefer not to expose their projects to raw make.

Our initial reaction (after acknowledging that the customer is always right) was negative in the extreme. The Viewpath sounded like a clever way of building and sharing sources. But could anyone bear to work in an environment that used it? Its all very well building against sources that aren't there, but how do you write software in such a vacuum?

In theory, well defined interfaces would allow you to take your build environment on trust. So much for theory. In practice you need access to standard header files, and your colleagues' code. You can't take it on trust. To produce a workable development scheme around the Viewpath seemed to require not just a clever make that understood the sharing mechanism, but a clever ls, grep, more and everybody else's favourite tool. Building this mechanism into a make tool is only the start of something we didn't want to get involved with. This goes to show that first impressions can often be quite wrong.

The Viewpath library

The tools/make idea was a red herring. The real problem with developing in a "source-sharing" environment is that it is alien to the traditional UNIX file system model. If you superimpose an alternative conceptual view on the standard hierarchical model, you're on your own. And what our customer needed was a different type of file system – not just the familiar two dimensional hierarchy, but with the added dimension provided by an inheritance mechanism that would provide controlled source sharing. These days sharing/re-use through inheritance is such a commonplace that there is no need to discuss it further.

Rather than building this inheritance mechanism into make, the ideal solution would be for the file system itself to manage file inheritance/sharing. Files would then "be there" for all the standard utilities, whether they were defined/redefined locally, or inherited from parent file-hierarchies. What they needed – in the unnecessary modern terminology – was an "object-oriented" file system. This was not what the project manager wanted to hear.

A viable alternative would be to encapsulate the standard kernel calls, and emulate inheritance at the user level. This involved placing a sandwich layer between applications and the kernel library calls – stat, access, opendir, readdir etc. When built with this "Viewpath" sandwich in place, software would still behave as if it was in a two-dimensional UNIX world. But the sandwich would ensure that it was a two dimensional view of a three dimensional world. And it would be translucent, if not completely transparent. The Viewpath library would unfold the extra dimension before it made the required kernel call. So for example, If a program access()'ed a file, the Viewpath sandwich would first work out what should be there – and use symbolic links to ensure that inherited files are really in place, then it would call access(2).

A practical decision was made about what files should be inherited. We didn't want generated files, objects or binaries to be inherited (unless users particularly wanted them). Nor the many temporary files that litter the filesystem. By default, the library would inherit files that are under SCCS control. Inheritance rules, and overriding would be the same as for any single-inheritance class hierarchy. We would worry about multiple inheritance if ever anyone should want to pay for it.

The Viewpath library contained all the checking, caching and linking software. The top of a local tree would be defined by a TREEPATH environment variable. Its inheritance (as in Build) through the VIEWPATH. Unless a user typed "ls -l", he could work as if the shared sources were all there, rather than symbolic links. And he would be guaranteed the latest version of every shared source, object or binary. Simple and effective, as long as it was fast enough and didn't get in the way.

Lynx: Managing Inheritance

There are two ways of using the library.

> The first is to recompile a wide range of tools using it, so that they all "see" the third dimension. We haven't yet needed to take this step.
>
> The second is to provide a few tools which allow users to unfold the extra dimension (and re-fold it later). This was to be the Lynx (because it manages all the links) program.

This gave us the groundrules for our CM system. It would have three separate layers:

> The Viewpath library – encapsulating the kernel calls, and implementing the extra (inheritance) dimension.
>
> Lynx – managing the development cycle, and providing the command-level interface to the source tree inheritance.
>
> The CM interface – required by Customer X, using Lynx for day-to-day source management, but providing the extra layer of traceability and control mandated for many large products.

Lynx was initially a mechanism for unfolding the extra filesystem dimension. Even while testing the Viewpath library it became clear that a lot more would be required. Lynx had to satisfy two classes of user: it had to provide the tree-management functions required by any CM system – creating trees, merging them back, tracking what changes have been made. It also had to provide for the developer, who would need it to check out sources, to fold and unfold, to produce snapshots, and to complement SCCS. All of Lynx's functions would be available, either informally through its shell interface – not dissimilar to SCCS – or wrapped up in a CM-system. Much of the following discussion is in terms of Lynx used as a stand-alone tool. It is equally applicable to Lynx when used as a component of a CM system.

Lynx and the developer

For developers, Lynx had to be no more difficult to use than SCCS. Most of the unresolved usability issues hinge on how much it should be used in conjunction with SCCS, and how much it should take care of the SCCS functions as well. When you check-out sources (ie obtain the "token" and a local SCCS copy of a file for modification), should it also do an "sccs edit"? In practice, a "lynx source" command is invariably followed by an "sccs edit".

Lynx is presented to users as a tool to:

- Let them set up *Lynx trees* at will. The first benefit of source inheritance is that software can be built in a child tree of the sources. This guarantees clean builds each time, allows arbitrary areas of the tree to be cleaned with "rm -rf" and ensures that if the developer can build the software, so can a tester, an integrator, or anyone else who cares to join to his tree (or use the sources once they have been merged back with the masters).
- Relieve them of the problem of having to package up their work for testing and re-integration. We had previously gone some way down this road [Til86a] by adopting a modified version of SCCS that kept an Audit trail – a log of all the most recent file updates. This could then be used as an input filelist to a tool that packaged up the appropriate versions into an archive ready for shipment back to the master sources. This mechanism was now redundant. An implication of working in a child file-hierarchy of the master sources is that your tree already re-defines the master sources. Shipment back no longer makes sense – all that is required is an automatic tree-merge.

  A developer merely has to say that a tree is "ready". The local files are automatically locked (SCCS level locking), the tree is then frozen and "in transit". This does not stop its own children being integrated back into it (in the manner of a russian doll) if convenient. Nor does it stop people joining to the tree to test it, or to do work on the next release. We'll explain all this in a moment.
- Allow them full read access to any part of the master source tree, without having to find out where the master sources are kept – whether a file is the latest version etc. They can browse the master sources by unfolding any local tree. Source that is irrelevant need never be unfolded.
- Give them confidence, that they know exactly what they are building against. If a file can be found in a local tree, it exists in the master sources. If the master sources change, it will impact the local tree immediately. There are no hidden surprises. No moving targets.
- Provide a source checkout mechanism to guarantee that no-one else was allowed to modify a source once it had been checked out. This could prove too restrictive – for example if the developer had finished with a part of the sources he is working with – or if the work had been done, but hadn't been checked back (re-integrated) into a parent tree – after all, it would not always be convenient to check back a major component. Release schedules and outside factors outside the development would come into play. This led to the need for flexible checkout and re-integration. In any inheritance hierarchy of trees, however complex or simple, there should be no predetermined order of re-integration. Nor should anyone in the hierarchy be inconvenienced with re-integrating someone else's work unless it suits them to do so.

Integration – or merging back the inherited tree structure.

In the simplest case, there would be one level of inheritance, a set of masters and a development tree. The work would be done in the development tree, and re-integrated into the masters afterwards. Life would progress in this cyclical fashion. A more likely scenario is that there are one or more master source trees (aka repositories) and hierarchies of development "trees" joining to the masters and to each other, all inheriting source directly from their parents, all guaranteed to be the only area working on the files checked-out locally. The trees may be for a variety of purposes some for testing, others for "the next release", others for experimental development. In each case, the inheritance chain grows longer because developers don't want to build against the master sources, but need to use the latest development version of a component to be sure that their work will function in the final product.

When a developer checks out sources, he is guaranteed that no-one can checkout the file upstream of him, and therefore change his baseline. He is also guaranteed that until he is finished with that file, no-one can check it out downstream of him for further development. He can explicitly *finish* using a file, or implicitly finish by making his tree frozen ready for merging. In the former case, he is still able to check-out his own source again if he later finds that he needs to – but only if someone else in the meantime hasn't checked it out from his tree.

Where a variety of inherited trees are used for different purposes, they will be ready for re-integration at different and unpredictable times. Since re-integration takes place in the manner of putting together one of those russian dolls, it doesn't matter what order you do the merging, as long as a child always goes into its immediate parent. The analogy soon breaks down. If my parent merges back into its parent, then I can continue until I am ready. When I merge back, the rule is simple. I signal my readiness to merge into the closest ancestor tree that hasn't itself been merged.

Another consequence of providing tools to manage trees owned by different people is whether or not to setuid. We live free of such restrictions. The work is done co-operatively. When I want to merge, I get my tree ready, and signal the appropriate parent that I can be integrated when it is convenient. He can choose to ignore my request (and it will become the responsibility of his parent when he merges back), or he an simply say "lynx integrate whatever-my-tree-is-called" and go for a cup of coffee while the work is folded back.

A Lynx environment is entirely co-operative. Lynx provides the ground-rules, enforces unique checkout, manages the whole cycle and takes care of all the trees in the forest. Users are expected to talk to each other – and this leads us on to the second class of Lynx user. Lynx can be used in isolation, like SCCS usually is. Or it can provide the mechanisms for a Configuration Management interface, where users are *obliged* to acknowledge each other's existence.

Lynx and SCCS

Lynx has a close relationship with SCCS. As we mentioned earlier, default Viewpath inheritance is based on files that are under SCCS control. Indeed the Lynx "Development Cycle" is an encapsulated SCCS cycle. We can follow the sequence.

Stage one is to set up a new tree. There are three ways of doing this, since a source tree can be:

a virgin tree, not inheriting any sources

a tree with an explicit inheritance Viewpath

a tree that joins on the end of another – ie the same as a class that inherits from another class.

This in theory offers wider scope for possible parentage than is usual in O-O languages, which only allow type 1 and 3. A notable use for type 2 is putting all the developments in the Viewpath to obtain a snapshot/tape of the entire project for backup/test purposes. This is equivalent to folding back all the developments into the parent tree.

Once a new tree is set up, it can *import* any sources from anywhere in the filesystem, as long as they are under SCCS control. It does this by pretending to inherit from the source areas, then unconditionally checking out the sources from them. Its done unconditionally of course, because there is no concept of unique checkout outside of a Lynx tree.

New files in the tree have to be put under SCCS control. This ensures they are inherited. For child trees to check them out, they have to be "returned for rework", which locks the SCCS file from anyone trying to edit it.

Once a child tree has joined to the repository, its user/s can unfold sources at will to reveal the third dimension. Any file, subtree or arbitrary component list can be checked out for work with the "lynx source" command. This ensures that no-one else can legally take out that file, and copies the whole SCCS history for the source file into the local area.

A great deal of time was spent worrying about what a user would get when he checked out a file. It was too great an assumption that he could manage with simply the latest version of the file. It would also have proved extremely expensive to merge a development SCCS history with the master history, and it would have forced us to abandon the goal of working "as if" integrated.

By shipping the whole baggage into the local area, integration ceases to be a merging activity. It is simply a file copy. Also any binaries built locally or in a test area will have exactly the same expanded SCCS keywords as the same files once integrated and rebuilt. The cost is that you maintain the development history within the master sources.

All Lynx actions on a file are fully logged within the comment fields of the SCCS history itself. This can be used by QA, but is also used by the system to ensure that no rogue copies of checked out files are slipped back in during integration.

Lynx and Pmak/make

This area was critical to Customer X, and to our own developers. It is covered by the requirement that there should be a seamless transition from development outside, to development within a Lynx tree. Using a high level build tool such as Pmak, this was never a serious issue, and the integrity of the source tree, and the flexibility of the inheritance management was our primary concern. If we were relying on make, the project would probably have gone straight for the Build approach, and concentrated on "sharing for construction".

Pmak has been described elsewhere. Very briefly, it takes *Build Parameters* (what header directories, where object libraries are kept, where software is to be installed etc) generates a makefile, builds the software, then discards the makefile. Wherever you are in a source tree, you inherit a build environment, which controls the construction of the makefile and thereby, your software. It also allows you to attach yourself to any other are of the filesystem, and pick up the "build environment" for that area. To ensure that the build proceeded smoothly, we merely had to attach each Lynx tree to its parent in order to inherit the appropriate libraries and header files from the tree in which they were built.

We had intended at the outset to compile Pmak against the Viewpath library, to produce a Build-like build tool. We resisted because we didn't simply want to produce yet another special set of tools with specialist knowledge in common, but instead an environment where all the developer's tools could access the inherited third dimension. We have come to realise that there is a convincing argument for making a special case of the build software itself.

Once compiled with the Viewpath sandwich, Pmak would be more than a Build clone. It would ensure that you have unfolded in your local tree, all the components that you are going to need to access directly. It is common practice to build the software you have to work on before you start changing it – just to check you've got all the bits. If you did this with a Viewpath'ed Pmak, it would unfold all the necessary source areas as it went, and you would be left after your test build with a suitably unfolded tree. Having to unfold, and *having to know what parts of the tree to unfold* is a major criticism of the current system.

Lynx Trees have roots

Lynx trees have a "below ground" and "above ground". Below ground is the tree root, that contains all the general tree information, its checkout log, and essential build information. Above ground is the "tree" itself, which contains your development directory hierarchy.

The division into root and tree serves two purposes. By keeping the tree data in the root directory, the tree does not have any extra files in it. It looks as it would in a stand alone non-Lynx tree. Also, as we mentioned earlier the inheritance structure allows you to keep your sources in one tree and build in another that is joined to it. When you need to clean out a set of intermediate, built files, you simply "rm -rf" at the tree level, without fear of damaging the tree itself.

The root directory has 3 elements:

- a directory for CM data – the checkout log, baseline information etc
- a Viewpath data file that logs the inheritance pedigree of the tree.
- a Pmak data file that ensures seamless access to object libraries, headers etc during the build

To test (and test build) software in Lynx trees, you merely have to join to that tree. What you build, (and therefore what you test) is the latest SCCS'ed version of each file. By joining, you are guaranteed a clean build, as if the development work had been re-integrated first. There is no way you can pick up non sccs'ed files or any local environmental features from the development. If you can build it and test it in an inherited tree, you can be sure that once integrated it will work the same.

Running software from Inherited Lynx Trees

This is not a problem when running finished software when all the components have been re-integrated. It is potentially serious where individual components are being developed and tested in their own Lynx Trees. It is the only area where the burden rests with the user, and where programming practices have to be considered.

The issue is that programs often need to use data files and perform various operations on their environment. Depending on how a program identifies where to pick these components up from, the Lynx strategy will stand or fall. If they rigidly assume a traditional file-system, then the whole approach has the potential to fall apart. The options are:

- A hard-coded path. This is the killer. Unless you install over the top of the standard version you are lost.
- An environment variable. This isn't so bad – but the problem is that it will expect all its data files to be positioned relative to the variable. So you would have to keep all the data files locally – whether you needed to build them or not. Nothing would get inherited.
- An access function. This is the ideal. Relative paths are evaluated through a library subroutine which may simply append the relative path to an environment variable. However it allows the

software to use another routine that first uses the Viewpath to look for the file, in case there is a more recent version that should be used.

We were obliged to standardise on the third option for all our software. Fortunately it's a problem that shows itself immediately, and rarely has serious implications for program structure.

Releases and disasters

Two words that often go together, but we'll take them separately. First releases:

Lynx includes a mechanism for generating the information you require to cut a release. While normally this would include the relative path of a source file, and its appropriate SCCS id, here we need more information. We need to know which tree each file comes from. Our distribution utility **mkdist** now takes this extended information and cuts the necessary archive however many links there are in the chain of Lynx trees.

Incidentally, both source and binary archives generated this way are identical. If you cut a distribution archive from the joined tree, and later cut another from the master sources once the changes have been re-integrated, the archives will be identical – even as far as the SCCS version ids of the individual source files – to an archive cut from the software after it has been integrated. **Integration is now a non-event.**

Now disasters. Things always go wrong. Files get deleted, filesystems get corrupted, and "the dumps" may not be geared up to rescue a project distributed over a number of Lynx trees if the inevitable ever happens. Again the solution follows directly from the inheritance structure, and the fact that none of the Lynx trees overlap. An active SCCS file is never present, thanks to unique checkout, in more than one tree. If you set up a Lynx tree that inherits explicitly (and in any order) from all the active trees, a release archive from this contains all the controlled files. Furthermore a snapshot from it contains all the information about what files are local to what development area.

To identify the inheritance structure, the tree check-out logs are all kept in a single hierarchy close to the master repository itself, and the local versions of it in the tree root are only symbolic links. The burden of backup is therefore removed from the individual developer, and even if he deleted everything by mistake only his work would be lost, and then only as long ago as the previous backup. Lynx does not compromise your back-up procedures. Its disaster recovery is only as good as these procedures, but if they are in place, recovery is quick and effective.

Using the software, some afterthoughts

We have been using Lynx and the CM since March. There are two sides to the discussion:

- Lynx as the workhorse for the CM. We have had one major release that has been entirely driven through the CM system. The only problems we had as far as the integration cycle was concerned were due to the CM catching one developer's attempt to short-circuit it very early on. The auditing and backup mechanisms proved sufficient to isolate and identify the crime quickly, prove guilt, execute the culprit and retrieve the correct files.

  An interesting exercise during this release was a brace of ports. Since the ports could (and did) entail changes to a wide variety of files, a port source release was cut, and used for the work. Once the port had been completed, a development tree was set up to incorporate the changes, the relevant sources checked-out and modified. The ability to join to this tree and do a full product test build showed up the problems with the port long before the work was re-integrated. But that, after all is what testing is about.

- Working in Lynx trees caused plenty of consternation. As we said earlier, the mechanism was translucent rather than transparent, and at first the idea of working within one's own full source tree caused a few problems. The biggest was the need to unfold the sources you wanted to see. This will soon be resolved by letting Pmak unfold as it test-builds. Other problems arose because people tried to build sub-components against locally built libraries which they had not previously needed to build. This problem too goes away with the Viewpath'ed Pmak. We had no complaints from people who had to use it regularly. The rough edges of a first release caused some dissention from very occasional users who didn't want to have to find out what was going on. We are addressing this as a serious issue.

  People didn't like SCCS files being locked against them when they tried to copy them manually rather than go via Lynx's checkout mechanism. One developer felt unnerved about working in a tree of symbolic links. "A bit like quicksand" was the comment. Our fear had been that the use of symbolic links would slow the build process up. In practice it made little difference. Perhaps the biggest complaint was when the filesystem started to run out of inodes. It still had acres of space left. So we made the CM delete dead trees rather than leave it to the user to clean up after him. So much for co-operation.

Summary

None of the ideas, and few of the mechanisms we have described are new. We have put them together to complement our Build software and SCCS to provide some basic source management facilities that work either stand-alone, or as a component of a CM system.

The source-sharing mechanisms have dramatically eased our integration/release problems, and are tolerated even by those amongst us who feel that a shell other than the Bourne shell is an intrusion into the way they work. Our development trees take up only about 6% of what they used to, and they eat inodes like there's no tomorrow.

What we don't understand is why the filesystems are as full as ever!

References

[Eri84a] V B Erickson and J F Pellegrin, "Build – A Software Construction Tool," *AT&T Bell Laboratories Technical Journal* **63**(6), pp. 1049-1059 (July 1984).

[Til86a] D M Tilbrook and P R H Place, "Tools for the Maintenance and Installation of a Large Software Distribution," *EUUG Florence Conference* (April 1986).

The Answer to All Man's Problems

Tom Christiansen

CONVEX Computer Corporation
POB 833851
3000 Waterview Parkway
Richardson, TX 75083-3851
tchrist@convex.com

ABSTRACT

The UNIX on-line manual system was designed many years ago to suit the needs of systems at the time, but despite the growth in complexity of typical systems and the need for more sophisticated software, few modifications have been made to it since then. This paper presents the results of a complete rewrite of the man system. The three principal goals were to effect substantial gains in functionality, extensibility, and speed. The secondary goal was to rewrite a basic UNIX utility in the perl programming language to observe how perl affected development time, execution time, and design decisions.

Extensions to the original man system include storing the whatis database in DBM format for quicker access, intelligent handling of entries with multiple names (via .so inclusion, links, or the NAME section), embedded tbl and eqn directives, multiple man trees, extensible section naming possibilities, user-definable section and sub-section search ordering, an indexing mechanism for long man pages, typesetting of man pages, text-previewer support for bit mapped displays, automatic validity checks on the SEE ALSO sections, support for compressed man pages to conserve disk usage, per-tree man macro definitions, and support for man pages for multiple architectures or software versions from the same host.

1. Introduction

The UNIX on-line manual system was designed many years ago to suit the needs of the systems at the time. Since then, despite the growth in complexity of typical systems and the need for more sophisticated software to support them, few modifications of major significance have been made to the program. This paper describes problems inherent in earlier versions of the *man* program, proposes solutions to these problems, and outlines one implementation of these solutions.

2. The Problem

2.1. The Monolithic Approach

One of the most serious problems with the *man* program up to and including the BSD4.2 release was that all man pages on the entire system were expected to reside under a common directory, */usr/man*. There was no notion of separate sets of man pages installed on the same machine in different subdirectories. At large installations, situations commonly arise in which this functionality is desirable. A site may wish to keep vendor-supplied man pages separate from man pages that were developed locally or acquired from some third party. An individual or group may wish to maintain their own set of man pages. Multiple versions of the same software package might be simultaneously installed on the same machine. A heterogeneous environment may want to be able to view man pages for all available architectures from any machine. Given the requirement that all man pages live in the same directory, these scenarios are difficult to impossible to support.

The *man* program distributed in the BSD4.3 release included the concept of a MANPATH, a colon-delimited list of complete man trees taken either from the user's environment or supplied on the command line. While this was a vast improvement over the previous monolithic approach, several significant problems remained. For one thing, the program still had to use the *access* (2) system call on all possible paths to find out where the man page for a particular topic existed. When the user has a MANPATH containing multiple components, the time needed for the *man* program to locate a man page is often noticeable, particularly when the target man page does not exist.

2.2. Hard-coded Section Names

Another problem with the *man* program unresolved by the BSD4.3 release was that all possible sections in which a man page could reside were hard-coded into the program. This means that while a **manp** section would be recognized, a **manq** directory would not be, and while a **man3f** directory would be recognized, a **man3x11** directory would not be.

Likewise, the possible subsections for a man page were also embedded in the source code, so a man page named something like */usr/man/man3/XmLabel.3x11* would not be found because **3x11** was not in the hard-coded list of viable subsections. Some systems install all man pages stripped of subsection components in the file name. This situation is less than optimal because it proves useful to be able to supply both a *getc* (3f) and a *getc* (3s). Distinguishing between subsections is particularly convenient with the "intro" man pages; a vendor could supply *intro* (3) *intro* (3a), *intro* (3c), *intro* (3f), *intro* (3m), *intro* (3n), *intro* (3r), *intro* (3s), and *intro* (3x) as introductory man pages for the various libraries. However, the task of running *access* (2) on all possible subsections is slow and tedious, requiring recompilation whenever a new subsection is invented.

3. References in the Filesystem

The existing man system had no elegant way to handle man pages containing more than one entry. For example, *string* (3) contains references to *strcat* (3), *strcpy* (3), amongst others. Because the *man* program looks for entries only in the file system, these extra references must be represented as files that reference the base man page. The most common practice is to have a file consisting of a single line telling *troff* to source the other man page. This file would read something like:

```
.so man3/string.3
```

Occasionally, extra references are created with a link in the file system (either a hard link or a symbolic one). Except when using hard links, this method wastes disk blocks and inodes. In any case, the directory gains more entries, slowing down accesses to files in those directories. Logic must be built into the *man* program to detect these extra references. If not, when man pages are reformatted into their cat directories, separate formatted man pages are stored on disk, wasting substantial amounts of disk space on duplicate information. On systems with numerous man pages, the directories can grow so large that all man pages for a given section cannot be listed on the command line at one time because of kernel restrictions on the total length of the arguments to *exec* (2). Because of the need to store reference information in the file system, the problem is only made worse. This often happens in section 3 after the man pages for the X library have been installed, but can occur in other sections as well.

The *makewhatis* (8) program is a Bourne shell script that generates the */usr/lib/whatis* index, and is used by *apropos* (1) and *whatis* (1) to provide one-line summaries of man pages. These programs are part of the *man* system and are often links to each other and sometimes to *man* itself. If any of the man subdirectories contain more files than the shell can successfully expand on the command line, the *makewhatis* script fails and no index is generated. When this occurs, *whatis* and *apropos* stop working. The *catman* (8) program, used to pre-format raw man pages, suffers from the same problem.

Of course, *makewhatis* wasn't working all that well, anyway. It was a wrapper around many calls to little programs that each did a small piece of the work, making it run slowly. It, too, had a hard-coded pathname for where man pages resided on disk and which sections were permitted. *Makewhatis* didn't always extract the proper information from the man page's NAME section. When it did, this information was sometimes garbled due to embedded *troff* formatting information. But even garbled information was better than none at all. Even so, these programs left some things to be desired. *Apropos* didn't understand regular expression searches, and both it and *whatis* preferred to do their own lookups using basic, unoptimized C functions like *index* (3) rather than using a general-purpose optimized string search program like *egrep* (1).

4. The Solution

4.1. A Real Database

The problem in all these cases appeared to be that the filesystem was being used as a database, and that this paradigm did not hold up well to expansion. Therefore the solution was to move this information into a database for more rapid access.

Using this database, *man* and *whatis* need no longer call *access*(2) to test all possible locations for the desired man page. To solve the other problems, *makewhatis*(8) would be recoded so it didn't rely on the shell for looking at directories.

4.2. Coding in Perl

When project was first contemplated, the perl programming language by Larry Wall was rapidly gaining popularity as an alternative to C for tasks that were either too slow when written as shell scripts, or simply exceeded the shell's somewhat limited capabilities. Since perl was optimized for parsing text, had convenient *dbm*(3x) support built in to it, and the task really didn't seem complex enough to merit a full-blown treatment in C or C++, perl was selected as the language of choice. Having all code written in perl would also help support heterogeneous environments because the resulting scripts could be copied and run on any hardware or software platform supporting perl. No recompilation would be required.

Some concern existed about choosing an interpreted language when one of the issues to address was that of speed. It was decided to do the prototype in perl and, if necessary, translate this into C should performance prove unacceptable.

The first task was to recode *makewhatis*(8) to generate the new *whatis* database using *dbm*. The *directory*(3) routines were used rather than shell globbing to circumvent the problem of large directories breaking shell wildcard expansions. Perl proved to be an appropriate choice for this type of text processing (see Figure 1).

4.3. Database Format

The database entries themselves are conveniently accessed as arrays from perl. To save space and accommodate man pages with multiple references, two kinds of database entries exist: direct and indirect. Indirect entries are simply references to direct entries. For example, indirect entries for *getc*(3s), *getchar*(3s), *fgetc*(3s), and *getw*(3s) all point to the real entry, which is *getc*(3s). Indirect entries are created for multiple entries in the NAME section, for symbolic and hard links, and for **.so** references. Using the NAME section is the preferred method; the others are supported for backwards compatibility.

Assuming that that WHATIS array has been bound to the appropriate *dbm* file, storing indirect entries is trivial:

```
    $WHATIS{'fgetc'} = 'getc.3s';
```

When a program encounters an indirect entry, such as for *fgetc*, it must make another lookup based on the return value of first lookup (stripped of its trailing extension) until it finds a direct entry. The trailing extension is kept so that an indirect reference to *gtty*(3c) doesn't accidentally pull out *stty*(1) when it really wanted *stty*(3c).

The format of a direct entry is more complicated, because it needs to encode the description to be used by *whatis*(1) as well as the section and subsection information. It can be distinguished from an indirect entry because it contains four fields delimited by control-A's (ASCII 001), which are themselves prohibited from being in any of the fields. The fields are as follows:

1 List of references that point to this man page; this is usually everything to the left of the hyphen in the NAME section.

2 Relative pathname of the file the man page is kept in; this is stored for the indirect entries.

3 Trailing component of the directory in which the man page can be found, such as **3** for **man3**.

4 Description of the man page for use by the *whatis* and *apropos* programs; basically everything to the right of the hyphen in the NAME section.

At first glance, the third field would seem redundant. It would appear that you could derive it from the character after the dot in the second field. However, to support arbitrary subdirectories like **man3f** or

```
s/\\f([PBIR]|\(..)//g;       # kill font changes
s/\\s[+-]?\d+//g;            # kill point changes
s/\\&//g;                    # and \&
s/\\\((ru|ul)/_/g;           # xlate to '_'
s/\\\((mi|hy|em)/-/g;        # xlate to '-'
s/\\\*\(..//g  &&            # no troff strings
    print STDERR "trimmed troff string macro in NAME section of $FILE\n";
s/\\//g;                     # kill all remaining backslashes
s/^\.\\"\s*//;               # kill comments
if (!/\s+-+\s+/) {
    #   ^ otherwise L-devices would be L
    print STDERR "$FILE: no separated dash in $_\n";
    $needcmdlist = 1;        # forgive their braindamage
    s/.*-//;
    $desc = $_;
} else {
    ($cmdlist, $desc) = ( $`, $' );
    $cmdlist =~ s/^\s+//;
}
```

Figure 1: ***makewhatis*** *excerpt #1*

man3x11, you must also know the name of the directory so you don't look in **man3** instead. Additionally, a long-standing tradition exists of using the **mano** section to store old man pages from arbitrary sections. Furthermore, man pages are sometimes installed in the wrong section. To support these scenarios, restrictions regarding the format of filenames used for man pages were relaxed in *man*, *makewhatis*, and *catman*, but warnings would be issued by *makewhatis* for man pages installed in directories that don't have the same suffix as the man pages.

4.4. Multiple References to the Same Topic

A problem arises from the fact that the same topic may exist in more than one section of the manual. When a lookup is performed on a topic, you want to retrieve all possible man page locations for that topic. The *whatis* program wants to display them all to the user, while the *man* program will either show all the man pages (if the **–a** flag is given) or sort what it has retrieved according to a particular section and subsection precedence, by default showing entries from section 1 before those from section 2, and so forth. Therefore, each lookup may actually return a list of direct and indirect lookups. This list is delimited by control-B's (ASCII 002), which are stripped from the data fields, should they somehow contain any. The code for storing a direct entry in the *whatis* database is featured in Figure 2.

Notice the check of the new datum's length against the value of MAXDATUM. This is because of the inherent limitations in the implementation of the *dbm*(3x) routines. This is 1k for *dbm* and 4k for *ndbm*. This restriction will be relaxed if a *dbm*-compatible set of routines is written without these size limitations. The GNU *gdbm* routines hold promise, but they were released after the writing of these programs and haven't been investigated yet. In practice, these limits are seldom if ever reached, especially when *ndbm* is used.

```
sub store_direct {
    local($cmd, $list, $page, $section, $desc) = @_; # args
    local($datum);

    $datum = join("\001", $list, $page, $section, $desc);

    if (defined $WHATIS{$cmd}) {
        if (length($WHATIS{$cmd}) + length($datum) + 1 > $MAXDATUM) {
            print STDERR "can't store $page -- would break DBM\n";
            return;
        }
        $WHATIS{$cmd} .= "\002";  # append separator
    }
    $WHATIS{$cmd} .= $datum;  # append entry
}
```

Figure 2: ***makewhatis*** *excerpt #2*

```
Idx  Subsections in ksh.1                Lines
 1   NAME                                    3
 2   SYNOPSIS                               22
 3   DESCRIPTION                            15
 4   Definitions.                           43
 5   Commands.                             338
 6   Comments.                               6
 7   Aliasing.                             107
 8   Tilde Substitution.                    47
 9   Command Substitution.                  28
10   Process Substitution.                  49
11   Parameter Substitution.               645
12   Blank Interpretation.                  15
13   File Name Generation.                  87
```

Figure 3: ***ksh*** *index excerpt*

5. Other Problems, Other Solutions

The rewrite of *makewhatis*, *catman*, and *man* to understand multiple man trees and to use a database for topic-to-pathname mapping did much to alleviate the most important problems in the existing man system, but several minor problems remained. Since this was a complete rewrite of the entire system, it seemed an appropriate time to address these as well.

5.1. Indexing Long Pages

Several of the most frequently consulted man pages on the system have grown beyond the scope of a quick reference guide, instead filling the function of a detailed user manual. Man pages of this sort include those for shells, window managers, general purpose utilities such as awk and perl, and the X11 man pages. Although these man pages are internally organized into sections and subsections that are easily visible on a hard-copy printout, the on-line man system could not recognize these internal sections. Instead, the user was forced to search through pages of output looking for the section of the man page containing the desired information.

To alleviate this time-consuming tedium, the man program was taught to parse the *nroff* source for man pages in order to build up an index of these sections and present them to the user on demand. See Figure 3 for an excerpt from the *ksh* (1) index page, displayable via the new **–i** switch.

The */usr/man/idx*/* directories serve the same function for saved indices as */usr/man/cat*/* directories do for saved formatted man pages. These are regenerated as needed according the the same criteria used to regenerate the cat pages. They can be used to index into a given man page or to list a man page's subsections. To begin at a given subsection, the user appends the desired subsection to the name of the man page on the command line, using a forward slash as a delimiter. Alternatively, the user can just supply a trailing slash on the man page name, in which case they are presented with the index listing like the one the **–i** switch provides, then prompted for the section in which they are interested. A double slash indicates an arbitrary regular expression, not a section name. This is merely a short-hand notation for first running man and then typing

```
/expr
```

from within the user's pager. See Figure 4 for example usages of the indexing features.

This indexing scheme is implemented by searching the index stored in */usr/man/idx1/ksh.1* if it exists, or generated dynamically otherwise, for the requested subsection. A numeric subsection is easily handled.

```
man -i ksh      # show sections
man ksh/        # show sections, prompt for which one

man ksh/tilde
man ksh/8       # equivalent to preceding line

man ksh/file
man ksh/generat # equivalent to preceding line
man ksh/13      # so is this

man ksh//hangup # start at this string
```

Figure 4: *Index Examples*

```
sub find_index {
    local($expr, $path) = @_;  # subroutine args
    local(@matches, @ssindex);
    @ssindex = &load_index($path);

    if ($expr > 0) {            # test for numeric section
        return $ssindex[$expr];
    } else {
        if (@matches = grep (/^$expr/i, @ssindex)) {
            return $matches[0];
        } elsif (@matches = grep (/$expr/i, @ssindex)) {
            return $matches[0];
        } else {
            return '';
        }
    }
}
```

Figure 5: *Locate Subsection by Index*

For strings, a case-insensitive pattern match is first made anchored to the front of the string, then – failing that – anywhere in the section description. This way the user doesn't need to type the full section title. The *man* program starts up the pager with a leading argument to begin at that section. Both *more*(1) and *less*(1) understand this particular notation. In the first example given above, this would be

```
less '+/^[ \t]*Tilde Substitution' /usr/man/cat1/ksh.1
```

Once again, perl proved useful for coding this algorithm concisely. The subroutine for doing this is given in Figure 5. Given an expression such as "5" or "tilde" or "file" and a pathname of the man page, *man* loads an array of subsection index titles and quickly retrieves the proper header to pass on to the pager. Perl's built-in **grep** routine for selecting from arrays those elements conforming to certain criteria made the coding easy.

5.2. Conditional Tbl and Eqn Inclusion

Several other relatively minor enhancements were made to the man system in the course of its rewrite. One of these was to include calls to *eqn*(1) and *tbl*(1) where appropriate. For instance, the X11 man pages use *tbl* directives to construct a number of tables. It was not sufficient to supply these extra filters for all man pages. Besides the slight performance degradation this would incur, a more serious problem exists: some systems have man pages that contain embedded **.TS** and **.TE** directives; however, the data between them was not *tbl* input, but rather its output. They have already been pre-processed in the unformatted versions. To do so again causes *tbl* to complain bitterly, so heuristics to check for this condition were built in to the function that determines which filters are needed.

To support tables and equations in man pages when viewed on-line, the output must be run through *col*(1) to be legible. Unfortunately, this strips the man pages of any bold font changes, which is undesirable because it is often important to distinguish between bold and italics for clarity. Therefore, before the formatted man page is fed to *col*, all text in bold (between escape sequences) is converted to character-backspace-character combinations. These combinations can be recognized by the user's pager as a character in a bold font, just as underbar-backspace-character is recognized as an italic (or underlined) one. Unfortunately, while *less* does recognize this convention, *more* does not. By storing the formatted versions with all escape-sequences removed, the user's pager can be invoked without a pipe to *ul* or *col* to fix the reverse line motion directives. This proivdes the pager with a handle on the pathname of the cat page, allowing users to back up to the start of man pages, even exceptionally long ones, without exiting the *man* program. This would not be feasible if the pager were being fed from a pipe.

5.3. Troffing and Previewing Man Pages

Now that many sites have high-quality laser printers and bit-mapped displays, it seemed desirable for *man* to understand how to direct *troff* output to these. A new option, **-t**, was added to mean that *troff* should be used instead of *nroff*. This way users can easily get pretty-printed versions of their man pages.

For workstation or X-terminal users, *man* will recognize a TROFF environment variable or command line argument to indicate an alternate program to use for typesetting. (This presumes that the program recognizes *troff* options.) This method often produces more legible output than *nroff* would, allows the user to stay in their office, and saves trees as well.

5.4. Section Ordering

The same topic can occur in more than one section of the manual, but not all users on the system want the same default section ordering that *man* uses to sort these possible pages. For instance, C programmers who want to look up the man page for *sleep*(3) or *stty*(3) find that by default, *man* gives them *sleep*(1) and *stty*(1) instead. A FORTRAN programmer may want to see *system*(3f), but instead gets *system*(3). To accommodate these needs, the *man* program will honor a MANSECT environment variable (or a **–S** command line switch) containing a list of section suffixes. If subsection or multi-character section ordering is desired, this string should be colon-delimited. The default ordering is "ln16823457po". A C programmer might set his MANSECT to be "231" instead to access subroutines and system calls before commands of the same name. A FORTRAN programmer might prefer "3f:2:3:1" to get at the FORTRAN versions of subroutines before the standard C versions. Sections absent from the MANSECT have a sorting priority lower than any that are present.

5.5. Compressed Man Pages

Because man pages are ASCII text files, they stand to benefit from being run through the *compress*(1) program. Compressing man pages typically yields disk space savings of around 60%. The start-up time for decompressing the man page when viewing is not enough to be bothersome. However, running *makewhatis* across compressed man pages takes significantly longer than running it over uncompressed ones, so some sites may wish to keep only the formatted pages compressed, not the unformatted ones.

Two different ways of indicating compressed man pages seem to exist today. One is where the man page itself has an attached **.Z** suffix, yielding pathnames like */usr/man/man1/who.1.Z*. The other way is to have the section directory contain the **.Z** suffix and have the files named normally, as in */usr/man/man1.Z/who.1*. Either strategy is supported to ease porting the program to other systems. All programs dealing with man pages have been updated to understand man pages stored in compressed form.

5.6. Automated Consistency Checking

After receiving a half-dozen or so bug reports regarding non-existent man pages referenced in SEE ALSO sections, it became apparent that the only way to verify that all bugs of this nature had really been expurgated would be to automate the process. The *cfman* program was verifies that man pages are mutually consistent in their SEE ALSO references. It also reports man pages whose **.TH** line claims the man page is in a different place than *cfman* found it. *Cfman* can locate man pages that are improperly referenced rather than merely missing. It can be run on an entire man tree, or on individual files as an aid to developers writing new man pages.

The amount of output produced by *cfman* is startling. A portion of the output of a sample run is seen in Figure 6. Some of its complaints are relatively harmless, such as *dbm* being in section **3x** rather than section **3**, because the *man* program can find entries with the subsection left off. Having inconsistent **.TH** headers is also harmless, although the printed man pages will have headers that do not reflect their

```
at.1: cron(8) really in cron(1)
binmail.1: xsend(1) missing
dbadd.1: dbm(3) really in dbm(3x)
ksh.1: exec(2) missing
ksh.1: signal(2) missing
ksh.1: ulimit(2) missing
ksh.1: rand(3) really in rand(3c)
ksh.1: profile(5) missing
ld.1: fc(1) really in fc(1f)
sccstorcs.1: thinks it's in ci(1)
uuencode.1c: atob(n) missing
yppasswd.1: mkpasswd(5) missing
fstream.3: thinks it's in fstream(3c++)
ftpd.8c: syslog(8) missing
nfmail.8: delivermail(8) missing
versatec.8: vpr(1) missing
```

Figure 6: *Sample* ***cfman*** *run*

filenames on the disk. However, entries that refer to pages that are truly absent, like *exec*(2) or *delivermail*(8), merit closer attention.

5.7. Multiple Architecture Support

As mentioned in the discussion of the need for a MANPATH, a site may for various reasons wish to maintain several complete sets of man pages on the same machine. Of course, a user could know to specify the full pathname of the alternate tree on the command line or set up their environment appropriately, but this is inconvenient. Instead, it is preferable to specify the machine type on the command line and let the system worry about pathnames.

Consider these examples:

```
man vax csh
apropos sun rpc
whatis tahoe man
```

To implement this, when presented with more than one argument, *man* (in any of its three guises) checks to see whether the first non-switch argument is a directory beneath */usr/man*. If so, it automatically adjusts its MANPATH to that subdirectory.

Not all vendors use precisely the same set of *man*(7) macros for formatting their man pages. Furthermore, it's helpful to see in the header of the man page which manual it came from. The *man* program therefore looks for a local *tmac.an* file in the root of the current man tree for alternate macro definitions. If this file exists, it will be used rather than the system defaults for passing to *nroff* or *troff* when reformatting.

6. Performance Analysis

The *man* program is one that is often used on the system, so users are sensitive to any significant degradation in response time. Because it is written in perl (an interpreted language) this was cause for concern. On a CONVEX C2, the C version runs faster when only one element is present in the MANPATH. However, when the MANPATH contains four elements, the C version bogs down considerably because of the large number of *access*(2) calls it must make.

The start-up time on the parsing of the script, now just over 1300 lines long, is around 0.6 seconds. This time can be reduced by dumping the parse tree that perl generates to disk and executing that instead. The expense of this action is disk space, as the current implementation requires that the whole perl interpreter be included in the new executable, not just the parse tree. This method yields performance superior to that of the C version, irrespective of the number of components in the user's MANPATH, except occasionally on the initial run. This is because the program needs to be loaded into memory the first time. If perl itself is installed "sticky" so it is memory resident, start-up time improves considerably. In any case, the total variance (on a CONVEX) is less than two seconds in the worst case (and often under one second), so it was deemed acceptable, particularly considering the additional functionality the perl version offers.

Nothing in the algorithms employed in the *man* program require that it be written in perl; it was just easier this way. It could be rewritten in C using *dbm*(3x) routines, although the development time would probably be much longer.

The *makewhatis* program was originally a conglomeration of man calls to various individual utilities such as *sed*, *expand*, *sort*, and others. The perl rewrite runs in less than half the time of the original, and does a much better job. There are two reasons for the speed increase. The first is the cost of the numerous *exec*(2) calls made via the shell script used by the old version of *makewhatis*. The second is that perl is optimized for text processing, which is most of what *makewhatis* is doing.

Total development time was only a few weeks, which was much shorter than originally anticipated. The short development cycle was chiefly attributable to the ease of text processing in perl, the many built-in routines for doing things that in C would have required extensive library development, and, last but not at all least, the omission of the compilation stage in the normal edit-compile-test cycle of development when working with non-interpreted languages.

7. Conclusions

The system described above has been in operation for the last six months on a large local network consisting of three dozen CONVEX machines, a token VAX, quite a few HP workstations and servers, and innumerable Sun workstations, all running different flavors of UNIX. Despite this heterogeneity, the same code runs on all systems without alterations. Few problems have been seen, and those that did arise were quickly fixed in the scripts, which could be immediately redistributed to the network. The principal project goals of improved functionality, extensibility, and execution time were adequately met, and the experience of rewriting a set of standard UNIX utilities in perl was an educational one. Man pages stand a much better chance of being internally consistent with each other. Response from the user and development community has been favorable. They have been relieved by the many bug fixes and pleasantly surprised by the new functionality. The suite of man programs will replace the old man system in the next release of CONVEX utilities.

The 2.6 MSD Software Development Environment

Ed Gould
Bradley White

MT XINU
2560 Ninth Street
Berkeley, California 94710
USA
ed@mtxinu.com

ABSTRACT

2.6 MSD is MT XINU's recently-released distribution of the Mach Operating System and other software. The distribution contains a complete BSD user environment, the X Window System, and some user-contributed software, in addition to Mach. The distribution supports three hardware families, the Vax, the Sun 3, and the IBM RT/PC.

The software development environment used to build and maintain this distribution – originally developed at Carnegie Mellon University – has a number of interesting properties. Among them are:

- Support for staged release levels
- RCS-based tools for source tree maintenance
- Release management and notification tools
- A compilation environment that includes viewpathing

This paper will detail the important features of the tools, and how we used them in producing the existing distribution. It will also describe our plans to use these same tools to concurrently maintain and support this release while developing the next one.

1. Introduction

Working with Carnegie Mellon University, MT XINU has recently released a distribution including the Mach Operating System and other software. The distribution provides the complete 4.3BSD-tahoe user environment running on a slightly enhanced Mach 2.5 kernel. In addition, the distribution includes the X Window System, Version 11 Release 4, from the Massachusetts Institute of Technology, as well as some user-contributed software. It is known as "2.6 MSD."

The requirements for this distribution included that it be available on a variety of hardware platforms. In particular, the Digital Equipment VAX family, Sun Microsystems' 68020-based Sun 3 line, and the IBM RT/PC are supported by the distribution. Binaries for all platforms are built from the same sources, with very few machine dependencies outside the kernel.

The tools we used to facilitate this work were originally developed at Carnegie Mellon University, and build upon the RCS Revision Control System [Tic82a, Orr90a]. They also include enhanced versions of `make`, `cpp` and `ld`.

2. Issues

One of the major challenges in assembling a software distribution is the maintenance and development of the software itself. Our source tree contains roughly 245 megabytes of code, divided among some large number of files. (This is a large collection of sources by some measures, and a modest one by others. We doubt that anyone would really consider it small.) Development was done by five or six programmers,

The work reported on in this paper was or is sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864, monitored by the Space and Naval Warfare Command under Contract Number N00039-87-C-0251.

working separately in a loosely coordinated manner. Each programmer had their own workstation; workstations shared access to the common source and object trees using the AFS distributed file system.

A variety of potential problems present themselves in this environment. The issue of how to maintain the freedom desired by the programmers to fix whatever problems they encounter as those problems arise, balanced against the need to control multiple modifications to the same sources, is an important one. Also, the need to isolate one programmer from experimental changes made, e.g., to the C library, by another programmer should not interfere with the experimenter's desire to maintain a consistent view of the development environment. In other words, one programmer should feel free to modify what would essentially be `/lib/libc.a` (even though that's not the real location of the library) and not have that change affect other programmers.

Further, we needed to build and manage binaries for at least three different hardware environments. A completely separate tree of object files must be maintained for each hardware platform to be supported.

3. Motivations

There were several motivations that prompted us to solve these problems. Foremost among them, probably, was heterogeneity. We wanted all of the binaries for all of the platforms to be built from exactly the same sources, so as to maximize the similarity among the systems as seen by the user. Further, we were obligated to support multiple hardware platforms, and wanted to do so in an easy, well ordered manner.

Secondarily, there are interesting questions regarding the maintenance of this collection of software from a more global perspective. In the deep, dark past, there was only one UNIX source tree. As the UNIX System proliferated, so did versions of the sources. Currently running in the United States is an experiment called the "National File System." It is interesting, at least, to consider the possibility of one national or international source tree. More important than unifying the world's sources, though, is the ongoing need for tools with which to manage ever larger collections of software. We think that these tools may be a step in that direction, following in the tradition of the Programmer's Workbench [Dol78a].

4. Solutions

The overall process with which we maintain our sources and build objects has five stages, as illustrated by Figure 1. Software is collected from a variety of sources and it is entered into the RCS tree with standard RCS tools. Then, it is extracted into a parallel source tree, from which the source portion of the distribution will be generated. Also from the source tree, the `build` program produces object files and, if requested, copies those objects into the release area, from which the binary portion of the distribution is built.

The release area contains several parallel trees of objects. For each hardware type, there are, currently, four such trees, with each tree representing one release stage. These multiple release stages, as described in more detail below, allow for, among other things, keeping the contents of the current release available while developing the next one.

4.1. RCS Tools

4.1.1. General

The Revision Control System (RCS) facilitates the management of software projects by providing support for multiple development paths, maintaining revision histories, and allowing symbolic names to be

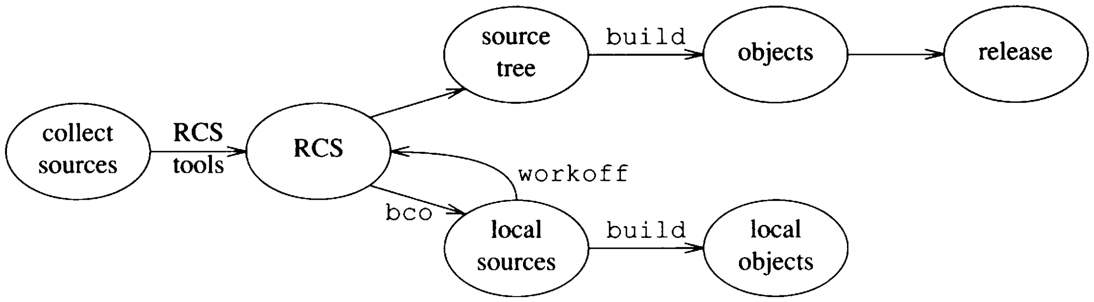


Figure 1: *Software Management Stages*

associated with particular revisions. However, it can be cumbersome to use in large, multi-programmer, multi-directory systems. The 2.6 MSD development environment provides both extensions to the standard RCS commands, and a set of "wrapper" programs which address some of the shortcomings. We describe the main features of these added facilities in the remainder of this section.

4.1.2. Extensions

The extensions to the RCS commands are mostly concerned with adding a flexible mechanism for naming revisions, and raising the status of branches. (RCS Version 4.0 has the beginnings of similar functionality with its "–b" switch.) By making extensive use of branches, separate development can easily proceed in parallel. The tools provide explicit support for merging a set of changes from a branch back onto the trunk, or for merging changes from the trunk back onto a branch (which allows continued development along a branch while new functionality can easily be picked up from the trunk).

While branching is a good way to organize development in an RCS tree, it is cumbersome to use with the standard RCS system. It is undesirable to make branches to all files in a system, or to remember exactly which files have had branches made for a particular development effort. As one step towards avoiding these problems, the 2.6 MSD RCS tools have been modified to use a set of rules, called *configuration*, when determining which revision to manipulate. (Normally, only a fixed revision number, or the last revision on the trunk is used.) Each rule in the configuration is tried in order until one matches an existing revision. Examples of configuration rules include:

An identifier	matches if the identifier is the symbolic name of a revision.
A revision number	matches if that revision exists.
A date	matches if a revision prior to that time exists.

Date rules only apply to the trunk (dates for branches can be ambiguous), and a special form refers to the last revision on the trunk. A file name may also be given with a revision number rule, in which case the rule only applies for that particular file. Two methods are supported for saving the "state" of the sources to a system. The first assigns a symbolic name to all revisions in the configuration. This name could then later be used to describe the same configuration. Alternatively, a list of file names and revision numbers can easily be generated to specify exactly which version of each component makes up the entire system.

4.1.3. Wrappers

In addition to the modifications to the RCS commands, a set of wrapper programs have also been developed (referred to as the "branch-style" RCS commands, or "b-commands" for short), which provide additional functionality aimed at simplifying the maintenance and integration of development path branches within a set of shared master sources. For example, the regular RCS commands provide no convenient way to group a set of files under development, whereas the *b-commands* maintain a special control file which contains the name of each file on the branch. Other features of the branch-style commands include:

- A new branch is automatically created upon the first check-in.
- The check-in procedure can prime the log messages with a difference listing, or with the log message from another branch.
- The check-out procedure automatically creates any necessary directory structure in the working area, and produces a writable working file.

The regular RCS commands assume that programmers have full write access to the master revision files. The branch-style variants also include support for executing an authentication cover program when write access is required to the RCS or source trees. This can be used to implement more fine-grained access control.

4.2. Compilation

Once the source trees are in place, the requirement to compile those sources into usable objects comes to the forefront. Our compilation environment tries to do two things. First, it provides a mechanism, based on viewpathing, that allows each programmer to have a local copy of only those source files that are actually being modified, retrieving unmodified sources from the main source area. Second, it allows programmers to work independently, without having to worry about interfering with, or being disturbed by, another programmer's work.

```
% workon /etc/newfs
[ BCSBBASE /afs/mtxinu.com/project/dist/member/ed/src ]
[ BCSSBASE /afs/mtxinu.com/mach/src ]
[ BCSVBASE /afs/mtxinu.com/mach/rcs ]
[ RCS_AUTHCOVER /usr/local/sdm/lib/authcover (owner mxarch) ]
Branch  [ed_newfs]
[ branch ed_newfs ]
[ running bcs to create .BCSconfig-ed_newfs ]
[ BRANCH: ed_newfs, VBASE: /afs/mtxinu.com/mach/rcs ]
[ creating ./.BCSconfig-ed_newfs ]
> <90/06/29,01:05

[ target etc/newfs ]
./etc/newfs: created directory
[ current directory is etc/newfs ]
[ running bcs to create .BCSpath-ed_newfs ]
[ BRANCH: ed_newfs, VBASE: /afs/mtxinu.com/mach/rcs ]
[ creating ./.BCSpath-ed_newfs ]
> ./etc/newfs
% bco newfs.c
[ BRANCH: ed_newfs, VBASE: /afs/mtxinu.com/mach/rcs ]
[ ./etc/newfs/newfs.c ]
/afs/mtxinu.com/mach/rcs/etc/newfs/newfs.c,v  -->
./etc/newfs/newfs.c
revision 2.3
done
[ creating ./.BCSset-ed_newfs ]
> ./etc/newfs/newfs.c
% vi newfs.c
% build
[ /afs/mtxinu.com/project/dist/member/ed/src (as latest) ]
mkdir ../../../obj/vax/etc/newfs
cd ../../../obj/vax/etc/newfs
cc -c    -O  ../../../../src/etc/newfs/newfs.c
cc -c    -O  /afs/mtxinu.com/mach/src/etc/newfs/mkfs.c
cc    -o newfs.out newfs.o mkfs.o
mv newfs.out newfs
%
```

Figure 2: *Modifying the* `newfs` *Program*

4.2.1. Locating Sources

Providing an environment that accomplishes these two goals is done largely by a method known as "viewpathing." Inspired by the "PATH" mechanism used by the UNIX shell [Bou78a] to locate binary executables, a viewpath (embodied in the "VPATH" environment variable) specifies a list of directories in which to find source files.

Viewpathing may, perhaps, be best understood by an example. Consider the `/etc/newfs` program. The sources to *newfs* reside in the `/afs/mtxinu.com/mach/src/etc/newfs` directory, and consist of three files: `Makefile`, `mkfs.c`, and `newfs.c`. When a programmer wants to modify the program, a local work environment is created, using the `workon` command. `Workon` sets up both a local directory in which the programmer will make modifications and the proper environment and support files for use by the *b-commands* described above.

The programmer proceeds to check out a source file using `bco`, which places the working copy into the current directory, modify it, and then use `build` to compile a new version. Effectively, what `build` does in this example is to set the `VPATH` environment to `:/afs/mtxinu.com/mach/src/etc/newfs` (as with `PATH`, the null string before the first colon denotes the current directory) and then invoke `make`. `Make` uses `VPATH` to locate the source files to be compiled, and passes expanded path names to `cc`. Files in directories listed earlier in `VPATH` will be found in preference to those in directories listed later, producing the desired results. A more detailed description of the modifications to standard tools appears below.

The actual mechanism used is, in fact, more like the 3-D file system [Kor89a], allowing for a list of trees of source files, and also allowing, by adding one more layer of directories, for the resulting object files to reside in yet another local directory. This additional layer is used to facilitate building objects for multiple hardware types from the same sources. Figure 2 illustrates a simple edit and compile session. Note that `build` first creates the object directory in which the local objects will be built and then invokes `make`. (The invocation of `make` does not appear explicitly in the session. It begins with the line reading "cd ../../../obj/vax/etc/newfs.")

Once the modifications are complete and the programmer is satisfied that they have been adequately tested, the `workoff` program is used to merge the changes back into the trunk of the RCS source tree.

4.2.2. Modified Tools

This viewpathing mechanism is implemented by, and thus confined to, a relatively small set of tools. In particular, to implement the mechanisms just described, only `make` and `cpp` (the C preprocessor) have been changed. `Make` now knows how to find the source files, and passes expanded path names on to whatever commands are specified within the `makefile`. `Cpp` must also know about viewpathing, so that it may properly locate `#include` files. `Cpp` receives viewpath information via the `CPATH` environment variable, which is also set up by `make`.

One other tool, `ld`, has been modified as well. A search path for libraries, called `LPATH`, has been added. The `build` program sets up `LPATH` so that libraries will be found only in the release areas, not from the currently-installed libraries on the workstation. Flags to `build` specify which release stage is currently in use. In some sense, this behavior mimics what could be obtained by doing a `chroot` into some pre-established development environment.

4.3. Release Management

After correct sources have been obtained, and they have been compiled in the appropriate manner, there remains the problem of "installing" the resulting files in a place where they will undergo testing and eventually find their way onto a distribution tape. We refer to this stage of the software life-cycle as *release management*.

The 2.6 MSD tools support a *staged release process* in which there are multiple distribution hierarchies with released files migrating on demand from one stage to the next. Typically these stages indicate a degree of confidence in the released software, e.g., alpha, beta, and final release stages. (It is these release hierarchies that also provide the appropriate environment for the compilation process.)

The release management tool is also responsible for setting the appropriate owner and group identifiers, and protection modes, as well as certain post-processing phases like stripping binaries. The value of attributes like owner, group, and mode, can be specified in a separate database, or in the corresponding `makefile`. Another database keeps track of who released what and when, and guards against versions of individual components being lost in the shuffle between stages.

As a component moves from one release stage to the next, a notification message is generated and mailed to a given address. This mechanism is typically used to keep the entire project team informed about changes to the software base, and seems to strike a good balance between being inundated with too much detail, and not knowing what is being changed underneath you. The initial notification message is primed with the RCS log generated during the check-in phase.

If a change needs to be backed out, it can be done so simply and effectively by reversing the order of the release stages. In our environment the release hierarchies reside on a single machine (for each supported architecture) known as the repository. Individual workstations then upgrade their local file systems using the SUP system [Sha89a].

5. Evaluation

We have found that, while these tools and mechanisms take some getting used to, they generally perform as expected, giving the desired results. We have encountered no problems of one programmer interfering with another's work, even when experiments have proven fruitless. (Of course, there are no safeguards against someone releasing a broken version of a program. Testing is still an important part of any development cycle.) In particular, using these tools has convinced us that a generalized viewpathing mechanism is very useful for software development.

Towards that end, we would like to see the 3-D file system, or something essentially similar to it, incorporated into the operating system. In the absence of such a development, we would like to have some more tools available. In particular, given that sources reside in at least two directories, and objects in a third, we find that we use the directory stack mechanism quite a lot. Some way to alleviate directory switching would be handy, although it is not easy to imagine just how to accomplish this. The current implementation has placed a premium on changing the fewest existing tools.

6. Futures

The multiple release stages will be of primary use in the process of building the second distribution. We want to keep the entire first distribution available, so that we may investigate problems as they arise. It may also be useful to have an intermediate stage between the first and second releases that represents fixes to the first release, but no enhancements.

7. Conclusions

We have been using a set of tools that provide for management of sources, a useful yet protected compilation environment that easily supports compiling binaries for multiple hardware platforms from one set of sources and feeds into a multi-stage release mechanism. Producing the 2.6 MSD distribution without these tools would have been a nearly insurmountable task. The balance of flexibility for the programmers against the constraints necessary to keep them from interfering with one another has been struck without significant inconvenience.

Further development in this area seems warranted. In particular, we would like to see a mechanism like the 3-D file system incorporated into the operating system kernel.

8. Acknowledgements

Initial development of these tools took place at Carnegie Mellon University. In particular, we acknowledge the large contributions of Mike Accetta, Glenn Marcy (now at the Open Software Foundation), and Doug Orr (now at Chorus systémes). Many of the additions to RCS were influenced by the design of Apollo's DSEE.

References

[Bou78a] S. R. Bourne, "The UNIX Shell," *Bell System Technical Journal*, Murray Hill, NJ **57**(6, part 2), p. 1931, Bell Laboratories (July-August, 1978).

[Dol78a] T. A. Dolotta, R. C. Haight, and J. R. Mashey, "The Programmer's Workbench," *Bell System Technical Journal*, Murray Hill, NJ **57**(6, part 2), p. 2177, Bell Laboratories (July-August, 1978).

[Kor89a] David G. Korn and Eduardo Krell, "The 3-D File System," pp. 147 in *Baltimore Conference Proceedings*, USENIX Association, Berkeley, CA (June, 1989).

[Orr90a] Douglas Orr, "RCS Cookbook," 2.6 MSD Programmer's Supplementary Documents, MT XINU, Berkeley, CA (June, 1990).

[Sha89a] Steven A. Shafer and Mary R. Thompson, "SUP: The Software Upgrade Protocol," Unpublished Manual, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA (September, 1989).

[Tic82a] Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE (September, 1982).

ATK + 8859 = Multi-Lingual Text and Mail

A Study in Expanding the Andrew Toolkit

Thomas Neuendorffer

Information Technology Center
Carnegie Mellon University
Pittsburgh, Pa. 15213
tpn+@andrew.cmu.edu

ABSTRACT

The Andrew Toolkit and Andrew Message system have been available on the X distribution tape since X11 release 2, providing multi-font text with embedded objects (drawings, rasters, animations, etc.) that can be edited, printed or mailed between users. The recent addition of the ISO 8859 fonts to the X-distribution has prompted the ITC to add some additional support to allow multi-lingual text and mail that takes advantage of the expanded character set.

This paper provides an overview of what facilities are available, and how they can be customized to fit the needs of a particular site. The goal of this paper is to illustrate some of the features of ATK that allow it to be adapted to specific needs or purposes. There is also a technical overview of what was involved in adding the ISO support to ATK while maintaining backward compatibility.

1. Introduction

The true test of a system is its ability to adapt to external needs and opportunities that arise after the system was conceived. UNIX has certainly passed this test in the way it has adapted to systems and computer architectures far beyond the PDP-11's on which it was first implemented. For a user interface toolkit, the challenges are different. Several recent papers [Hay90a, Wil90a, Cat90a] have described the use of Andrew as the basis for major programming efforts. The goals of this paper are to present the expansion methods of ATK that allow significant customization and application development without significant programming effort, and to illustrate the line of thinking that goes into producing an ATK application. In the process, we hope to demonstrate how the problem of accented characters can be solved relatively easily given a system with reasonable power and flexibility.

1.1. The problem/opportunity

While graphical workstations have been available for several years now, the writers of software for these machines have been surprisingly slow in taking advantage of their capabilities. This is particularly true when it comes to text oriented systems, accented characters and electronic mail. There are relatively few systems that provide on-screen editing of accented characters. Those that do exist typically do not deal with electronic mail, or the problems of exporting these capabilities to other applications. Nor has there been much work in the area of toolkits that support the creation of applications that handle accented characters. There is an obvious need to deal with this problem on a system-wide basis. A user should be able to read "Flygande bäckasiner söka hwila på mjuka tufvor" in electronic mail and copy that into a text being edited or paste it in a spreadsheet.

In 1987, the International Organization for Standardization (ISO) published a set of 8-bit single byte encoded fonts [ISO87a] to support the Latin alphabet used by many western countries. The fonts included with X-Window system [Sch86a] release of January 1990, adhere to part one of this standard. With these basic resources in place, the Andrew toolkit was well situated to attack this problem.

1.2. Issues

The key issues in dealing with the ISO fonts in ATK were.

1. Screen display
2. Internal representation
3. External representation, in a form compatible with electronic mail.
4. Printing.
5. Ease and flexibility of user entry.

Of these issues, the first three are basic support issues that needed to be dealt with in the lower levels of the system, while the last two call for good basic support within the system to allow their development as extensions. Our goal has been to see that a good general capability in these areas is provided, while, at the same time, the system remains open for local conventions, modifications and customization, including graphical interfaces.

2. Background: The Andrew Toolkit

The Andrew toolkit (ATK) [Pal89a] provides a object-oriented environment wherein objects can be dynamically-loaded. In addition to containing multiple fonts, text can contain embedded raster images, spreadsheets, drawing editors, equations, simple animations, etc.. These embedded objects could themselves contain other objects, including text. These resulting multi-media documents can be edited, printed, or sent in mail messages via the Andrew Mail System [Bor88a]. AMS takes advantage of the fact that the ATK external file format (*datastream*) is defined to be short line seven-bit ASCII, so that it can safely send ATK files around in electronic mail, regardless of the fact that they may contain esoteric objects, like animations or electronic pianos. Because AMS uses the same text object that the ATK editor does, any modifications to that text object to deal with ISO fonts will automatically add that functionality to both the editor and the messages program, as well as the many other applications that use the text object. In addition, since the cut-buffer also uses datastream format, it will also be possible to cut and paste ISO characters between applications, just as it is currently possible to cut and paste styled text.

In order to understand ATK programming, it is important to grasp the concepts of the two main hierarchies within the system, the view hierarchy and the class hierarchy.

2.1. The Class Hierarchy

The class hierarchy should be familiar to those with experience in object oriented programming. Whenever a programmer codes a new object, he/she writes it to be a sub-class of another object, whose methods (functions) it will inherit and can override. Two children of the top-level observable class are the *view* class and the *dataobject* class. The *dataobject* class is designed to be the parent of those objects that will store and read data from a file. One dataobject is the *text* class, that will store text strings and style information to a file. When one is editing text with the ATK editor *ez*, it is the *text* object that stores and saves that information.

Most *dataobjects* have a corresponding *view* object. The *view* is responsible for being able to display the dataobjects information on the screen, and provides the interaction with the menu and keystroke packages that allows the user to edit the *dataobject's* information. The *textview* object provides this capability for *text*. The combination of a *dataobject* and its *view* is called an *inset*. See Figure 1.

2.2. The View Hierarchy

The view hierarchy is more unique to ATK. ATK provides a general mechanism that allows insets to be embedded in one another. A text inset could contain a spreadsheet inset that could itself contain a raster, a drawing, or even another text inset. The standard ATK editor *ez* does not, in fact, know anything about editing text. Text just happened to be one of the insets that it can embed in its *frame*, which is a view that manages the message line, as well as the underlying views being edited. *Ez* is just as capable of viewing and editing a spreadsheet or a raster image as it is text. Figure 2 illustrates the editor editing text that contains an animation inset, along with a diagram showing the view hierarchy. Requests for services, menu postings and keystroke requests are passed up the tree, and events (mouse hits, redraw requests, etc.) are passed down. The parent can override a child's requests, or add their own on in addition. Events are passed to the parents who have the option of passing them on to their children.

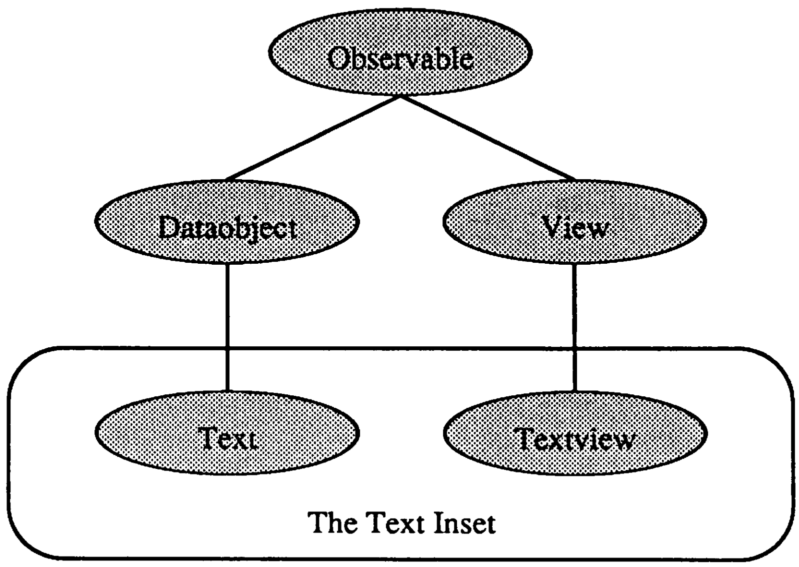
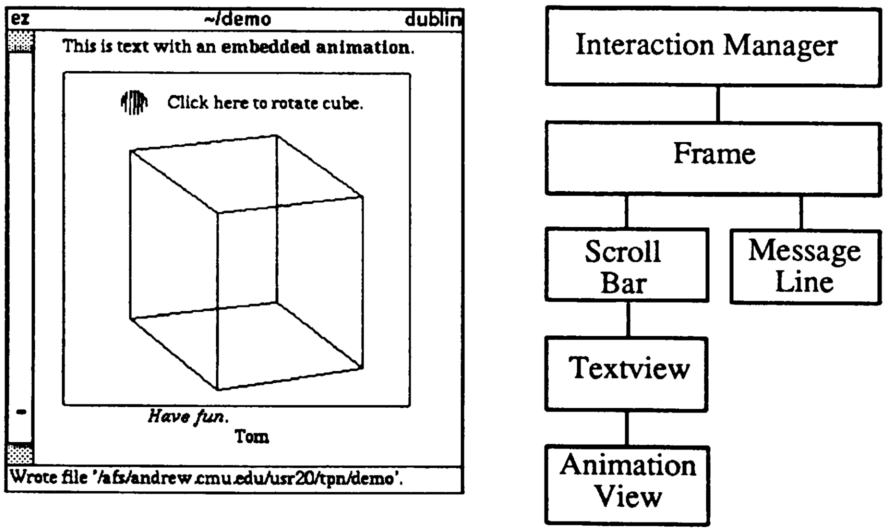


Figure 1: *The class hierarchy*



This is text with an embedded animation.

Click here to rotate cube.

Have fun.
Tom

Wrote file '/afs/andrew.cmu.edu/usr20/tpn/demo'.

Figure 2: *The view hierarchy*

For our purposes, this is particularly important in dealing with key sequences and menu choices. The view at the top of the view tree handling all of this is the inset manager or *im*. When a view receives the input focus, the *im* is the object that sees that each request to map a procedure with one or more keystrokes or menu entries is handled.

2.3. Customizing the Interface

During this process of adding the procedures requested by the view, the *im* also checks to see if the user has added or overridden any procedures in his init file for that viewtype. If so, the user specified procedures are added. In addition, many objects check for attributes in the user's preference file that will customize a specific interface. The system manager can also install global init and preference files. These can allow for significant site customization without any changes to the source.

3. Approaches to supporting ISO characters in ATK

3.1. Creating an ISO inset

The typical approach to extending ATK is to write a new inset. To apply that approach here would mean treating ISO characters the same way that we treat drawing, raster images and other objects that can be inserted into text. Whenever an ISO character would be needed, there would be an interface that would insert character sized inset containing that one character into the text. Printing could also be supported, since insets generally provide their own print methods. This approach has the advantage of being completely backward compatible, requiring no interface changes to the text object. It would also allow these characters to be inserted in insets other than text. The disadvantage comes in the fact that insets are somewhat heavyweight approach to inserting single characters in a document. It would also make certain common actions, such as searching a text for a string containing on or more ISO characters, a much more difficult (though not impossible) routine to implement. The ability to add insets is a powerful mechanism unique to ATK, however since ISO characters, for most operations, should be treated the same as other text characters, the use of an ISO inset was rejected.

3.2. Subclassing text

Another approach would be to create a sub-class of text with the additional functionality of being able to deal with ISO characters. This is the approach taken by the specialized editors that have been created to edit C, Modula 3, and Lisp code. It is also the approach taken by various other objects, like one that displays a list of choices and provides feedback to an application when a user clicks the mouse on a choice. Again, while this is an excellent approach for some applications, it would not be appropriate for dealing with ISO characters. For one thing, the use of ISO characters should be ubiquitous. All of the sub-classes of text, like the message line or the list object mentioned above, should be able to deal with ISO characters. ATK does not currently provide the capability for subclasses to masquerade as their parent class as far as other sub-classes are concerned. Add to this the fact that when text was written, certain decisions were made regarding what methods needed to be available for overriding by the subclass. Certain things (such as displaying an individual character on the screen) are going to have to be handled by the parent class, for efficiency if no other reason.

There is ongoing work at Kieo University, Japan [Kit90a] that is using this approach to create a Japanese text inset. Given the completely different model that is needed to support Japanese characters, this is a reasonable approach.

3.3. Expanding the text object via new proctable entries

ATK provides the capability for programmers to write modules to expand the capabilities of existing objects. These modules provide new functions that can be invoked by the user via key-strokes or menu commands. Because of its flexibility, and the fact that it allowed dealing with the existing text object, it was decided to use this approach, combined with backward compatible modifications or bug fixes to the text inset itself where necessary and/or appropriate.

3.4. Using Ness

Ness, an object expansion language that is shipped with ATK, includes the capability of defining new user level procedures that modify text. Though it could have been used to implement some of the desired routines, C based extensions had the advantage that they could deal with parts of the system that Ness doesn't provide interfaces for, the buffer help facility for example. Still, much customization work is possible with Ness. See [Han90a] for more information.

4. Providing the basic support

4.1. Screen display

This in fact was already dealt with. Since X had always allowed for more than 128 characters per font, the ATK graphics layer was already prepared to deal with them, including a call to indicate if a given character was in a given font. Using the X font aliasing feature, it was relatively simple to replace the standard ATK fonts with their ISO equivalent. A few minor bugs regarding the different handling of chars and unsigned chars had to be dealt with.

4.2. Internal representation

This also presented little problem. The ATK text object was not using the 8th bit of its character string for any other purpose, except for the placeholder character that was placed in the location of an embedded inset. And, even in this case, it was only used as a hint that a view may be held by a surrounding style object. So internally, no changes were made to store the 8-bit characters. Again, some changes had to be made for modules like the search routine, but these were relatively minor.

4.3. External representation

During the initial work, it was sufficient to store the resulting files in 8-bit form, however this would not be reasonable for shipped code. The ATK datastream is defined to be 7-bit ASCII, with no long lines. This was done primarily to allow it to be sent as electronic mail with standard mailers. All ATK objects are expected to save themselves according to these guidelines. Now of course there are many ways to save 8 bits in seven bit form, however we needed a way that would be backward compatible with older versions of ATK. The form chosen is a variation on the way styled text is stored. An "a" with the high bit set will store as

```
    \^{a}
```

Compare this to the way that a bold "**a**" would store.

```
    \bold{a}
```

This format has the advantage that older unpatched versions of ATK will just treat high bit characters as an unknown style with the name "^". Thus if a new text is read and written out again by an old version of ATK, the high-bit information will preserved, even though the display will be incorrect.

4.4. Printing

ATK generates troff code for high-quality multi-font printing. The decision to use troff was made at a time when it was the only commonly available system under UNIX capable of printing high quality text on laser printers. Though troff is not distributed with ATK, it is generally available on most UNIX systems. ATK's use of troff for printing has, from the beginning, been both a blessing and a curse. It has been treated as the assembly language for creating printed documents and for the most part, has worked quite well. Where it falls apart is where we don't really have enough information to do what needs to be done, the primary example of this being the proper handling of tabs. Through the years though, we have been able to adapt it to provide additional facilities, such as the printing of tables of contents, indices, and footnotes, as well as being able to deal with embedded images, either through insets that generate troff drawing, or insets that generate postscript that is placed in the troff stream and passed through to the printer. While there has been talk of getting ATK to generate postscript directly for printing, a large amount of work would be required to write code to take care of all of the things we are currently using troff to do.

Since raw troff does not handle accented characters, we needed to look into producing macros to deal with them. In the process, we decided to modify the text-to-troff conversion object *txttroff* to look for a user-defined procedure for translating high-order bit characters to usable troff strings. This allowed us to experiment with various options, while allowing for routines that dealt with local printing conventions.

4.5. Regarding the basic support

At this point the lowest level of support is in place. One could argue that the above changes fall into the category of bug fixes to the underlying system, and that the bugs were historical artifacts of the fact that ATK was developed on a window system that assumed a 7 bit character set. What is left is system expansion work that can be done without changes to the underlying system.

5. The first application, Swedish.c

In development efforts of this type it is often useful to attack a more specific problem as part of the process of finding a general solution. We began to concentrate efforts on a specific target audience, a group at Chalmers University of Technology in Sweden.

The Chalmers group had been working with ATK for several years, however they used the standard Swedish conventions when they needed accented text. For example, a left open bracket "}" would be used to represent an "å". By adapting their printing programs to convert back to the accented form, they were able to get properly printed documents, but this solution could obviously be improved upon. There should be no reason to deal with these sorts of limitations in a graphical workstation environment.

In response to their needs, the module Swedish.c was written. It provides dynamically loadable routines for each of the 6 most common Swedish ISO characters into text, and procedures for searching through a section of text and replacing all of the "special" characters with their ISO equivalents. Since text has now been modified to store the ISO characters in 7-bit format, two users, both using the ATK messages program to send and receive mail, could now exchange native language electronic mail. Of course not everyone they correspond to will be using ATK, so the conversion routines can be used to easily convert to and from the local conventions.

Users are able to call these routines by adding the following lines to their .atkinit file

```
addkey swedish-o-diaeresis-small \eS textview
addkey swedish-o-diaeresis-cap \eT textview
addkey swedish-a-diaeresis-small \eU textview
addkey swedish-a-diaeresis-cap \eV textview
addkey swedish-a-overcircle-small \eW textview
addkey swedish-a-overcircle-cap \eP textview
addmenu swedish-convert-to-ATK ''Swedish,Use Swedish characters'' textview
addmenu swedish-convert-from-ATK ''Swedish,Use brackets for Swedish'' textview
```

The *addkey* requests will call the given procedure from a textview when the given key-sequence is typed. In this case, an ESCAPE W (or F5 on an IBM keyboard) will insert an å into the text being edited, but the user could modify the file for any desired key or key-sequence. Similarly, the *addmenu* requests will add a line to a new or existing card that, when chosen, will call the named procedure. Here, a "Use Swedish characters" will be added to the card titled "Swedish" to call the convert-to-ATK routine. These routines can also be added for all users at a site by adding them to the global .atkinit file. An additional proctable was added to deal with printing that would take advantage of the hook added to *texttroff*. This function needed only to do the same conversions that convert from ATK did, except that the backslash character had to be escaped.

```
\ \
```

This relatively simple module (included in its entirety in Appendix 1) did much to make both ATK and X far more useful to the people at Chalmers. They have expanded it to included a few more ISO characters and expect to begin introducing the messages program to their students in the fall.

6. Attacking the general problem

With the success of Swedish.c, the next step was to attack the problems encountered in a generally applicable way.

6.1. Several solutions to character input

If a site has a keyboard set up to send the proper ISO-codes for the ISO characters on its keys, then character entry should be straight forward. However not all sites have these luxuries. In order to support a variety of interaction styles, several solutions to the general problem of character input were implemented.

6.1.1. Character modifying procedures

The first set of methods simply added an accent to the last character typed. Each procedure was linked to a specific accent and could be bound to a menu or keystroke. When invoked, the procedure would look at the last character entered, do a table lookup, and replace it with the appropriate ISO character.

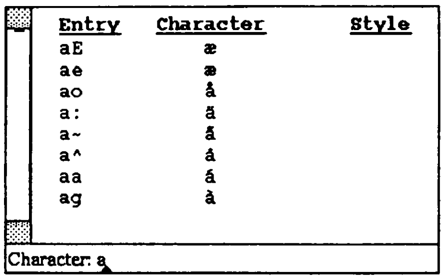
Another set of procedures allowed the accent character to be entered first. This was possible because the key binding package allows for overrides to be set so that a programmer can cause keys to be interpreted by a special function before possibly being interpreted by the normal methods. So an override was set such that the next key typed would enter the code for the typed character with an appropriate accent.

6.1.2. Compose character, version 1

The problem with these solutions was that since the mapping of character to accent was done by the general keymap facility, there was no central knowledge of accent-specific mappings that could conveniently be set or looked up to provide the user help. The next step was to provide a translation table and a compose character function.

For flexibility, it was decided use a user-replaceable table to specify the translations between character combinations, a high-order-bit character to input, and the troff string for printing. The location of this table is specifiable in the users preference file. This again allows for customization by both sites and individuals. An example table entry that would specify the translation of an "a" follow by a "^" to an "â" (ISO character 226) looks like:

```
a^ 226 a\*^
```



Entry	Character	Style
aE	æ	
ae	æ	
ao	å	
a:	ä	
a~	ã	
a^	â	
aa	á	
ag	à	

Character: a

Figure 3: *Completion Help for the letter 'a'*

This entry also specifies "a*^" as the string to be sent to troff for printing this character (see Printing below).

It was now possible to provide a compose character function that provided online, interactive help. The string input facilities provided by the editor package includes the ability to display a help buffer when a "?" is typed and to fill in part of an input string in response to a space character. Since the strings to be entered were all ~3 characters, providing completion was perhaps somewhat redundant. The help facility, however, allows the user to grow familiar with the character set provided by compchar quickly and easily as they are not forced to look up the character in a help file or other documentation. Figure 3 shows the help buffer for completing the letter a.

6.1.3. Compose character, version 2

The primary failure of this help mechanism occurs in Andrew messages because while it does use some of the facilities provided by the editor package, other facilities, including the help buffer, are disabled since it does not use the buffer package. In addition, using the string input facilities had the drawback of requiring a newline to terminate the name of the character desired. This rapidly became an annoyance even in testing so a way to avoid this was clearly needed. A second compchar function was implemented using the key overrides to allow the input of a character to be terminated automatically upon the completion of the character. Help could then be provided in the message line as each character was typed. So with this new approach, when one types the "compose character" key, followed by an "a", the message line would display the following message prompting for the completion

Possible completions: æ-E, æ-e, å-o, ä-:, ã-~,â-^,á-a,à-g

When the completion character was typed, the character would be immediately entered in the document.

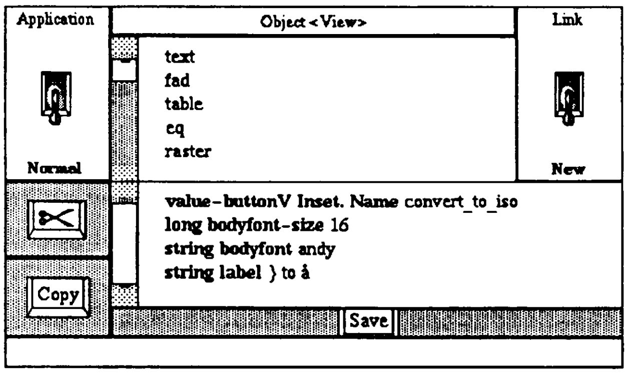
6.2. Printing

After implementing the local solution above, we started looking at more general solutions to printing accented characters with troff. It turns out that two of the standard macro packages for troff, mm and ms, provided troff macros for printing the accented characters and we were able to adapt these macros for use with ATK. For initial testing, we used the expansion method added to the text-to-troff translation object for Swedish to implement this, along with another preference option that allowed us to modify the troff macros that ATK applies to the troff it generates. Once this proved a general enough solution, we installed it as the part of the default printing scheme, leaving in place the capability for user override.

7. Adding a graphical interface

7.1. A Scenario

Let us return to Sweden for the moment and consider the following scenario. A site administrator has a large number of users (both experienced and novice) using AMS for sending and receiving mail. Not all of the users have Swedish keyboards and he/she would like to provide simple graphic interface that would allow them to take advantage of these newly available non-ASCII characters. One option would be to provide a "soft" on-screen keyboard that would display the commonly used characters, and allow the users to enter them into a mail message by clicking the mouse on the desired character. It would also be useful to provide buttons to do the transition to and from the local conventions. On most systems, this would be a fairly difficult project, requiring a lot of programming and major modifications to the underlying mail



text
fad
table
eq
raster

value-buttonV Inset. Name convert_to_iso
long bodyfont-size 16
string bodyfont andy
string label } to å

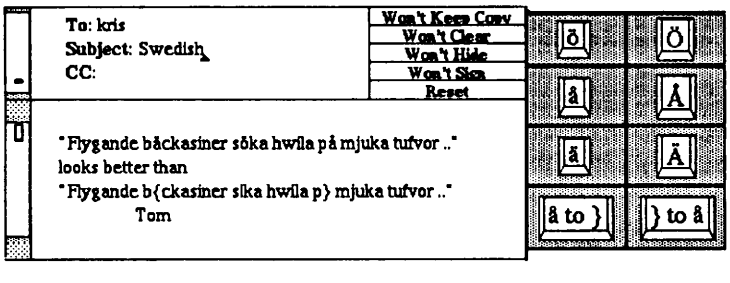
Figure 4: *The Arbcon controller object for Adew*

program, assuming that it would be even doable. Andrew provides support to allow this kind on interface to be added with under 100 lines of user written code.

7.2. Online creation of the interface

The Andrew Development Environment Workbench (Adew) described in [Neu89a] and [Bee90a] allows the programmer to quickly prototype the application on the screen by pasting together a collection of insets. It then provides support for turing the prototype into a finished application. The sample Adew programs have been mostly stand-alone application like a calculator, or interfaces to external devices, like the mailable electronic piano mentioned above. With a little additional knowledge of the system however, Adew can be used for graphical extensions to new or existing insets. So to deal with the scenario, let's start with creating the soft keyboard. Adew provides a set of buttons, switches, sliders and the like that can be pasted into a prototype. This prototype can be based on text, spreadsheet, of any other inset that allows embedded objects. For this application, we can use the tiling *lset* object to quickly create eight panels to hold the buttons. The Arbcon (Figure 4) can then be used to create references to new button objects that are simply pasted into place. Each of these buttons has various attributes that can be set via the Arbcon, including a string to place in the button. Since we have already modified text to allow input of ISO strings, we can now use to compose character function to input the desired characters to be displayed. The resulting "soft keyboard", next to a sendmessage window can be seen in Figure 5.

We now have an easily modifiable visual prototype on the screen. This prototype is stored as a editable file, not unlike any other file in the system. The next step is to think about how to implement the application. Adew provides a program called createcon that will take one of these prototypes and writes for it a dynamically loadable controller object. This object, when associated with the prototype, will contain pointers to all of the dataobjects and views in the application, and callback routines for all of the buttons, sliders and switches. Normally each button will be given a different call-back, based on the logical name of the object, provided by the prototyper as one of the object attributes. We will use this to set up the bottom two translation buttons, that need only to call the existing procedures in swedish.c. Another mechanism allows the same call-back to be assigned to all objects with the same logical name prior to a postfixed number (example: name_23). In this case, the value of the number will be provided as an argument to the call-back. This provides us with a simple way to implement the remaining keys, by giving each button the same name, followed by the number of the ISO character to insert. When a button is pressed, the call-back procedure will insert the proper character into the sendmessage text. The last



To: kris
Subject: Swedish
CC:

"Flygande bäckasiner söka hwila på mjuka tufvor .."
looks better than
"Flygande b{ckasiner s|ka hwila p} mjuka tufvor .."
Tom

Figure 5: *A sendmessage window with Swedish soft keyboard*

problem is one of integration. How does one create the button panel and controller in such a way that they have a pointer to the sendmessage object.

7.3. Tying it all together

Normally the controller is automatically tied to the prototype by means of a control button, that has, as one of its attributes, the controller's name and a function in the controller to call to initialize it. This is also how the createcon program normally knows what prototype object to create. In this case, we want to use a different initialization method that can be called from *sendmessages* to create the interface.

The first step is to create the controller object by running createcon with the -c flag to indicate the name of the class we wish to create.

```
    createcon -c msbcon buttons.lset
```

The result will be a new msbcon.c file, a new msbcon.ch file (to define the new object), an imakefile, and a shellscript (makemsbcon) that will build the Makefile and the application. What we need to add is a proctable entry, not unlike we did for swedish.c above, that can be called from *sendmessage* to create a new window containing the buttons, and a new instance of the controller object. Since this procedure will have pointers to all of the necessary objects, it can be responsible for providing the controller object with the needed pointer to the sendmessage object.

The accompanying code (Appendix 2) shows all of the user written code needed to write the application in this scenario. Without comments, it is well under 100 lines of user-written C. While some pieces could have been more straight-forward, we are continually looking for examples like this where new general functionalities should be added to the lower level's of the system.

It could be noted that this is a fairly specialized application, dealing only with sendmessage and the Swedish situation. The philosophy is that graphical interfaces should be easily enough to implement so that they can be specialized for specific applications, or even custom made for specific users. In this case, since Adew separates the prototype from the application itself, one could easily write a controller that bound the button to another application. Another application that provided a graphical interface for adding accents to existing characters would not be difficult.

8. Conclusion

ATK offers a variety of expansion methods, each one appropriate for a certain set of problems. Because these sets overlap, there is often more than one approach to a given challenge. While this can lead to confusion at times, it also lends a large degree of flexibility to the programmer who, with a variety of approaches available, can often choose either the most direct path to a specific problem, or a more general solution to a set of challenges. This ability to support the implementation of fast prototypes, complex long term solutions, and most everything in between is a major strength of ATK. In addition, the modular approach of ATK often means that a solution applied to one application will work on others with no additional implementation effort.

ATK is not the system to end all systems, and not all problems are as easy to deal with as the ones described above. However, it is hoped that some of solutions that ATK has pioneered will make their way into commercial systems. In the meantime, ATK provides a viable alternative for many users who may be dissatisfied with the limitations of the current alternatives.

9. Acknowledgments

Thanks to Rob Ryan who did much of the work of implementing the general solutions described above. The work described in this paper is available as part of the patches to ATK currently being distributed by the ITC.

Thanks also to...

Christer Bernerus of Chalmers University of Technology for his interaction with regard to swedish.c.

Bill Cattey of Project Athena at MIT for some useful discussions and his work on the troff macros.

Mark Pawliger for his information on foreign language keyboards.

Andy Palay for review and advice.

And all the other implementors, supporters and users of ATK. Thanks folks.

References

[Bee90a] Martin D. Beer, "Developing Document Management Systems Using the ANDREW Toolkit.," in *Proceedings of UKUUG Summer 1990 Conference*, London (July 1990).

[Bor88a] Nathanial Borenstein, Craig Everhart, Jonathan Rosenberg, and Adam Stoller, "A Multi-media Message System for Andrew," in *Proceedings from the USENIX Winter 1988 Conference* (February 1988).

[Cat90a] William Cattey, "The Evolution of Turnin. A Classroom Oriented File Exchange Service," pp. 171-181 in *Proceedings of the USENIX Summer 1990 Conference* , Anaheim CA (June, 1990).

[Han90a] W. J. Hansen,, "Enhancing documents with embedded programs: How Ness extends insets in the Andrew ToolKit," in *IEEE Computer Society International Conference on Computer Languages* (1990).

[Hay90a] Charles C. Hayden, John. C. Mitchell, Jishnu Mukerji, and Frederick A. Schmidt, "The Software Development Assistant," *AT&T Technical Journal* (March/April 1990).

[ISO87a] Technical committee ISO/TC97, *Information Processing – 8-bit single byte coded graphic character sets,* International Organization for Standardization (ISO), Switzerland (1987).

[Kit90a] Kaz Kitagawa, *Private communication,* Kieo University, Japan (1990).

[Neu89a] Thomas Neuendorffer, "ADEW: The Andrew Development Environment Workbench: An Overview," in *Proceedinngs of the 3rd Annual X Window conference*, Boston (1989).

[Pal89a] Andrew J. Palay, Wilfred J. Hansen, Mark Sherman, Maria G. Wadlow, Thomas P. Neuendorffer, Zalman Stern, Miles Bader, and Thom Peters, "The Andrew Toolkit – An Overview," in *Proceedings of the EUUG Spring 1989 Conference* (1989).

[Sch86a] R.W. Scheifler and J. Gettys, "The X Windows System," *ACM Transactions on Graphics* **5**(2), pp. 79-109 (April 1986).

[Wil90a] Nick Williams and William Cattey, "The Educational On-Line System," in *Proceedings of the EUUG Spring 1990 Conference*, Munich (April 1990).

Appendix 1 – Swedish.c

```
/*------------- SWEDISH.C -- ----------*/
#include <class.h>
#include <andrewos.h>
#include <proctbl.ih>
#include <swedish.eh>
#include <textv.ih>
#include <text.ih>
int (*textview_SelfInsertCmd)();
/* routines for entering characters */
void swedish_odsinsert(tv) struct textview *tv;
{   textview_SelfInsertCmd(tv,246);}
void swedish_odcinsert(tv) struct textview *tv;
{   textview_SelfInsertCmd(tv,214);}
void swedish_adsinsert(tv) struct textview *tv;
{   textview_SelfInsertCmd(tv,228);}
void swedish_adcinsert(tv) struct textview *tv;
{   textview_SelfInsertCmd(tv,196);}
void swedish_aosinsert(tv) struct textview *tv;
{   textview_SelfInsertCmd(tv,229);}
void swedish_aocinsert(tv) struct textview *tv;
{   textview_SelfInsertCmd(tv,197);}
```

```
/* return the position of the document to act upon */
static getSandL(t,tv,start,len)
struct text *t;
struct textview *tv;
long *start,*len;
{
    if((*len = textview_GetDotLength(tv)) >  0)
        *start = textview_GetDotPosition(tv) ;
    else {  /* process whole document */
        *start = 0;
        *len = text_GetLength(t);
    }
}
/* Conversion table */
static unsigned char conv[] = {
    '\\',   (unsigned char) 214,
    '{',    (unsigned char) 228,
    '}',    (unsigned char) 229,
    '[',    (unsigned char) 196,
    ']',    (unsigned char) 197,
    '|' ,   (unsigned char) 246,
    0,0 };
/* convert to use iso characters */
void swedish_converttoATK(tv)
struct textview *tv;
{
    struct text *t;
    long start, len;
    register unsigned char c, *cp;
    long rep = 0;
    t = (struct text *) textview_GetDataObject(tv);
    getSandL(t,tv,&start,&len);
    for(;len; len--,start++){
        c = text_GetChar(t,start);
        for(cp = conv; *cp != 0; cp++){
            if(*cp++ == c){
                text_AlwaysReplaceCharacters(t,start,1,cp,1);
                rep++;
            }
        }
    }
    if(rep != 0)text_NotifyObservers(t,0);
}
/* convert to use local conventions */
void swedish_convertfromATK(tv)
struct textview *tv;
{
    struct text *t;
    long start, len;
    register unsigned char c, *cp;
    long rep = 0;
    t = (struct text *) textview_GetDataObject(tv);
    getSandL(t,tv,&start,&len);
    for(;len; len--,start++){
        c = text_GetChar(t,start);
        for(cp = conv + 1; *cp != 0; cp++){
            if(*cp++ == c){
                text_AlwaysReplaceCharacters(t,start,1,cp - 2,1);
                rep++;
            }
        }
    }
    if(rep != 0 )text_NotifyObservers(t,0);
}
```

```
/* provide conversions for printing */
char * swedish_converttotroff(i)
unsigned char i;
{
    static char buf[6];
    register unsigned char *cp;
    if(i == 214){/* special case backslash, since it has to be quoted */
        *buf = '\\'; buf[1] = '\\'; buf[2] = '\0';
        return buf;
    }
    buf[1] = '\0';
    for(cp = conv + 3; *cp != 0; cp++){
        if(*cp++ == i){
            *buf = *(cp - 2);
            return buf;
        }
    }
    sprintf(buf,"\\\\%3.3o",i);
    return buf;
}
boolean swedish__InitializeClass(ClassID)
struct classheader *ClassID;
{
    struct classinfo *textviewtype = class_Load("textview");
    struct proctable_Entry *pr;
    int res;
    /* import the textview routine for entering characters */
    if((pr = proctable_Lookup("textview-self-insert")) != NULL
        && proctable_Defined(pr) ){
        textview_SelfInsertCmd = proctable_GetFunction(pr) ;
    }
    /* export routines */
    proctable_DefineProc("swedish-o-diaeresis-small",swedish_odsinsert,
                          textviewtype,NULL,"insert small diaeresis o");
    proctable_DefineProc("swedish-o-diaeresis-cap",swedish_odcinsert,
                          textviewtype,NULL,"insert cap diaeresis o");
    proctable_DefineProc("swedish-a-diaeresis-small",swedish_adsinsert,
                          textviewtype,NULL,"insert small diaeresis a");
    proctable_DefineProc("swedish-a-diaeresis-cap",swedish_adcinsert,
                          textviewtype,NULL,"insert cap diaeresis a");
    proctable_DefineProc("swedish-a-overcircle-small",swedish_aosinsert,
                          textviewtype,NULL,"insert small overcircle a");
    proctable_DefineProc("swedish-a-overcircle-cap",swedish_aocinsert,
                          textviewtype,NULL,"insert cap overcircle a");
    proctable_DefineProc("swedish-convert-to-ATK",swedish_converttoATK,
                          textviewtype,NULL,"Convert to Iso characters");
    proctable_DefineProc("swedish-convert-from-ATK", swedish_convertfromATK,
                          textviewtype,NULL,"Convert from Iso characters");
    proctable_DefineProc("swedish-convert-to-troff", swedish_converttotroff,
                          &swedish_classinfo,NULL,"return troff string");

    return TRUE;
}
```

Appendix 2 – isomsg.c

```
/*----------- isomsg.c -----------*/
........ program generate code deleted ............
#define BUTTONFILE environ_AndrewDir("/lib/arbiters/mb.lset")
int (*cfromiso)();
int (*ctoiso)();
static struct isomsg *FindSelf();
static struct view *NewWindow(filename,bflags)
char *filename;
int bflags;
```

```
{
    struct frame *newFrame;
    struct im *window;
    struct buffer *buffer;
    /* create a new buffer on the given file */
    if((buffer = buffer_GetBufferOnFile(filename,buffer_ReadOnly
                                        | buffer_MustExist)) == NULL){
        char buf[1300];
        sprintf(buf,"ERROR: can't create %s",filename);
        message_DisplayString(NULL,0,buf);
        return NULL;
    }
    /* create a frame and window  to hold the buffer */
    if ((newFrame = frame_New()) != NULL)
        if ((window = im_Create(NULL)) != NULL) {
        /* insert the frame in the window,
           enable the menu commands,
           add the message handler and put the frame in the buffer */

            im_SetView(window, newFrame);
            frame_SetCommandEnable(newFrame,TRUE);
            frame_PostDefaultHandler(newFrame, "message",
                                     frame_WantHandler(newFrame, "message"));
            frame_SetBuffer(newFrame, buffer, TRUE);
        }
        else{ /* window creation failed , destroy the frame */
            frame_Destroy(newFrame);
            newFrame = NULL;
        }
    /* return the created frame */
    return (struct view *) newFrame;
}
isomsg_Start(v,dat)
struct view *v;
long dat;
{
    struct isomsg *self;
    struct sendmessage *mes;
    if(!class_IsTypeByName(class_GetTypeName(v),"sendmessage")){
        /* Error, must be called from a sendmessage object */
        message_DisplayString(NULL,0,"Must be called from sendmessages");
        return;
    }
    /* keep a pointer to the passed in sendmessage view */
    mes = (struct sendmessage *) v;
    /* create a new 150x200 window containing the ADEW created buttons */
    im_SetGeometrySpec("150x200");
    v = NewWindow(BUTTONFILE,0);
    if(v == NULL) return;/* window creation failed */
    /* call FindSelf to associate a new isomsg controller object
       with the buttons in the frame */
    if((self = FindSelf(v)) == NULL){
        message_DisplayString(NULL,0,"Can't init source file");
        return;
    }
    /* store a pointer to the sendmessage view in the isomsg object*/
    self->mess = mes;
    /* if the sendmessage object is destroyed, we need to be informed,
       so that we can delete our pointer to it */
    sendmessage_AddObserver(mes,self);
}
```

```
........ program generate code deleted ............
static void convert_to_isoCallBack(self,val,r1,r2)
struct isomsg *self;
struct value *val;
long r1,r2;
{
if(r2 == value_OBJECTDESTROYED) {
        if(self->convert_to_iso == val) self->convert_to_iso = NULL;
}
{
/* user code begins here for convert_to_isoCallBack */
if(ctoiso != NULL && self->mess != NULL && self->mess->BodyTextview != NULL){
    ctoiso(self->mess->BodyTextview);
}
/* user code ends here for convert_to_isoCallBack */
}
}
static void convert_from_isoCallBack(self,val,r1,r2)
struct isomsg *self;
struct value *val;
long r1,r2;
{
if(r2 == value_OBJECTDESTROYED) {
        if(self->convert_from_iso == val) self->convert_from_iso = NULL;
}
{
/* user code begins here for convert_from_isoCallBack */
if(cfromiso != NULL && self->mess != NULL && self->mess->BodyTextview != NULL){
    cfromiso(self->mess->BodyTextview);
}
/* user code ends here for convert_from_isoCallBack */
}
}
static void cvtCallBack(self,val,r1,r2)
struct isomsg *self;
struct value *val;
long r1,r2;
........ program generate code deleted ............
{
/* user code begins here for cvtCallBack */
struct textview *tv;
unsigned char st[1];
struct text *t;
if(self->mess){
    /* get pointers to the text and the textview that the
       sendmessageobject uses to store the body of the message. */
    if((tv = self->mess->BodyTextview) == NULL) return;
    if((t = self->mess->BodyText) == NULL) return;
    /* since we stored the numeric value of the iso character as
       the second half of the logical inset name,
              that number is passed as r1.*/
    *st = (unsigned char) r1;
    /* insert that string as the dot postion, notify the text's observers
       that it has changed, and advance the dot position */
    text_InsertCharacters(t,textview_GetDotPosition(tv),st,1);
    text_NotifyObservers(t,0);
    textview_SetDotPosition(tv,textview_GetDotPosition(tv) + 1);
}
/* user code ends here for cvtCallBack */
........ program generate code deleted ............
boolean isomsg__InitializeClass(ClassID)
struct classheader *ClassID;
{
struct classinfo *viewtype = class_Load("view");
firstisomsg = NULL;
proctable_DefineProc("isomsg-go",isomsg_go, viewtype,NULL,"isomsg go");
```

```
/* user code begins here for InitializeClass */
{
    struct proctable_Entry *pr;
    /* load the swedish class */
    class_Load("swedish");
    /* export the start routine so that it can be called from sendmessage */
    proctable_DefineProc("isomsg-start",isomsg_Start, viewtype,NULL,"isomsg star
    cfromiso = ctoiso = NULL;
    /* import the swedish conversion routines */
    if((pr = proctable_Lookup("swedish-convert-from-ATK")) != NULL
        && proctable_Defined(pr) ){
        cfromiso = proctable_GetFunction(pr) ;
    }
    if((pr = proctable_Lookup("swedish-convert-to-ATK")) != NULL
        && proctable_Defined(pr) ){
        ctoiso = proctable_GetFunction(pr) ;
    }
}
/* user code ends here for InitializeClass */
return TRUE;
}
........ program generate code deleted ............
```

NFS Servers for Filesystems on Optical Disks

Hans Peter Klünder

iXOS Software GmbH
jan@ixos.uucp

ABSTRACT

Optical disks are becoming more wide spread in diverse applications. Pressed CDROM-, writeable WORM- and rewriteable MO-disks are being used to replace paper, microfiches and magnetic tapes. This has given rise to the problem of simple and efficient access to the optical disks.

Our solution is based on the NFS protocol. An NFS Server maintains filesystems on the disks and presents them to clients in a network, using the NFS protocol and thus hiding the hardware and its peculiarities. The client kernel hides the communication with the Server, and applications enjoy the comfort of filesystems without perceiving that file accesses are satisfied by a user-level Server.

This paper presents the NFS servers we developed for each of the three types of disks.

Network File System

The NFS protocol defines the communication between a fileserver and clients. The fileserver presents a filesystem to then network, and clients in the network access the filesystem.

Remote Procedure Calls, roughly corresponding to system calls, define simple actions like create and read. A client that wants to access the filesystem sends an RPC message to the server; the server satisfies the request and responds. The RPCs contain no information about the physical structure of the filesystem; satisfying abstract RPCs on a real filesystem is the server's task. Thus, peculiarities of the filesystem are hidden from the client.

Writing applications would be painful if they had to send RPCs in order to access a Network File System. Therefore, RPCs are implemented in the client's kernel. The server's filesystem can be mounted by the client and when system calls refer to the filesystem the kernel satisfies them by sending corresponding RPCs to the server and waiting for the answer. Thus, communication with the server is hidden from the application, and the server's filesystem appeares like any ordinary filesystem.

However, what looks like a filesystem to the user is a fiction invented by the server and believed by the client kernel. Usually, the server presents a filesystem that corresponds to a standard filesystem on a magnetic disk that the server accesses directly. Equally well a server may maintain a special filesystem that properly fits on a particular hardware, presenting it to the clients in a simple and pleasant shape. That's the way we use NFS: The Server maintains optical disks and whatever may be on them, presenting a simple and uniform UNIX-like filesystem to the clients.

The Server and its Filesystem

For each of the three types of optical disks, we developed an NFS Server. The Server is a user program that manages devices, i.e. drives and jukeboxes, and filesystems on the sides of the disks, presenting them to the clients via RPC interface according to the NFS protocol. More precisely, the Server offers three RPC services: The NFS service and the mount service as specified in the NFS protocol, and an additional administration service. The administration service accepts RPCs from a special administration program, thus enabling an administrator to control the Server. The mount service enables client computers to mount the Server's filesystem, and the NFS service actually satisfies the NFS requests, thus defining the user's view on the filesystem.

A Filesystem on a side of an optical disk is called a volume. Each CDROM has one volume; WORM- and MO-disks have two sides each of which may contain a volume. The Server manages devices, disks and volumes and combines the accessable volumes to a single filesystem by adding an artificial root directory that contains the root directories of the volumes. The volumes have names, and these names are the names of their roots in the Server's root. Thus, if the Server's filesystem is mounted on a client computer, a user, viewing the Server's root, will find various subdirectories, each of them representing the root of a volume. An application accesses a volume through the corresponding subdirectory in the Server's root.

The Server's filesystem does not indicate the location of a volume in a device. When a disk is moved from one jukebox to another, the Server's filesystem is unchanged, and a client doesn't notice the movement. When the Server get's an RPC that requests access to a volume, the Server first locates the volume and perhaps moves it into a drive of a jukebox and then performs the required action on the volume.

Performing an RPC on a volume depends on the type of disk; this is where the Servers differ.

Compact Disk Read Only Memory

Filesystems on CDROM were standardized by the High Sierra Format and the similar ISO 9660. Both describe hierarchical filesystems that are pressed on the CDROM and can't be modified. The Server interprets these filesystems and satisfies reading RPCs; writing RPCs are rejected.

Write Once Read Many

On WORM disks, each block can be written once and cannot be modified. We need a UNIX-like filesystem that can be modified at any time and anywhere. To realize this, we divide each volume into a control region and a data region, each of them starting at one end of the volume and growing until they collide and the volume is full. The data region contains the content of the regular files, always contiguous. The control region contains a sequence of control blocks, each of them describing a modification of the filesystem, e.g. creation or deletion of a directory or a file. In other words, the control region is a complete history of the filesystem structure, including references to the data region for regular files.

To achive contiguous files, the Server internally buffers files after creation, performing write requests actually on magnetic disk. The difficulty is to find a suitable moment for flushing a file to WORM; NFS is stateless, and the Server is not informed when an application closes a file. Fortunately, many commands that create files change their attributes before closing them. Therefore, the Server flushes, as an important side effect, a file to WORM when the user requests changes of file attributes. Another possibility to flush files on WORM is an adminstration RPC that instructs the Server to flush all files that are not modified since a specified amount of time; using this RPC, a simple demon may periodically instruct the Server to flush forgotten files. Once a file is flushed, writing on the file is expensive: When a write is requested on a file that is already on WORM, the Server copies the entire file into a new buffer on magnetic disk and then treats it as mentioned above; thus, changing one byte duplicates the whole file on disk. However, this way of buffering and flushing has been proved to work well, since applications that want to change one byte of a file normally don't choose WORM disks for their work. Typically, WORM disks are utilized in two ways: Developers use WORM disks for storing large source trees and public domain packages from their piles of magnetic tapes, thus having online, for example, the 379 cooking recipes from the EUUG conference tape of Dublin 1987; they read tapes to WORM using tar and cpio, and both do exactly what the Server expects. Customers use WORM disks for archiving large amounts of data; the data must never be changed, and for that reason they choose WORM disks.

Though the entire information about a volume's filesystem structure can be found in the control region, it could be hard for the Server to find some particular information that is stored in a few of all these control blocks. For example, an RPC called "readdir" requests the content of a directory, but entries in a directory can be made at any time, and their control blocks are spread over the control region. The Server can't search for entries of the directory in the whole control region, meanwhile sending the user to drink a cup of tea or two. Therefore, the Server maintains an additional database on magnetic disk containing the filesystem structures of the volumes. The database is updated whenever a control block is written, thus always representing the current state of the volumes. Using the database, the Server satisfies RPCs like "readdir" without physically accessing a disk at all; a simple SQL statement does the job. This is a huge performance advantage when the disks are stored in a jukebox: Clients may browse the volumes, and the Server can satisfy their requests without moving disks in the jukebox, unless they read the files or modify the filesystem.

Magneto Optical

On rewritable Magneto Optical disks, the Server realizes standard filesystems. MO-disks, like WORM, are often used in jukeboxes; therefore, the Server maintains a database similar to the database for WORM filesystems.

The Server's process model

Conventional NFS servers bind RPCs to processes: A server gets an RPC, does some work and replies. The usual way of serving many parallel clients is to start several processes, each of them working independent and handling complete RPCs. On fast devices like magnetic disk this works fine: Most RPCs are handled quickly, and nearly always at least one server process is idle and accepts RPCs. On slow devices like optical disks and jukeboxes, this conventional model fails: The net users learn how to mount the server; soon there will be more clients than server processes and all these clients access disks in the same jukebox. Clients do such things. Most of the server processes will sleep because they wait for the jukebox. They can't accept new RPCs, even if an RPC could be satisfied quickly from cache or from database. A crowd of RPCs accumulate while most processes wait for a single slow resource and a crowd of users complains that the server doesn't work.

Therefore, we bind processes to resources and RPCs to data structures. The Server is a group of specialized processes communicating via pipes and shared memory. The shared memory contains several data structures, among them caches and an RPC-book with a number of RPC-slots, each of them capable to keep the entire information associated with an RPC received by the Server.

One process receives RPCs and writes them into the RPC-book. Very often, the RPC-process can satisfy an RPC immediately, using caches and buffers and thus avoiding access to optical disk or database. Otherwise, it sends the RPC to a device-process that manages a drive or a jukebox or to a db-process that accesses the Server's database. More precisely, the RPC-process writes a pointer to the RPC-slot into an appropriate pipe and then is ready to receive the next RPC; each process has a pipe for receiving such pointers and examines the pipe when idling. The process that got the pointer may finish the RPC and reply to the client. Otherwise, the process sends it along to another process; for example, when an RPC requests modifications on a volume, the proper device-process writes to optical disk and then a db-process updates the database.

Thus, if there are many clients accessing the same jukebox, most of them wait while their RPCs are queued in the pipe the device process reads from. But these RPCs don't block other processes; in particular, the RPC-process is nearly always ready for accepting RPCs. Thus, RPCs that access caches or database only or a volume in another jukebox are served immediately. Of course, this process model can't speed up slow devices. But it ensures that if a resource is free the responsible process doesn't wait for anything else and clients can use the resource without delay.

Kernel versus Server

UNIX is often called an open system, and the word open may be interpreted to indicate that the kernel is always open for extensions with everything one can think about. Instead of writing a user-level Server, it's possible to implement an additional filesystem in the kernel using the kernel's file system switch. This forces the well-known trend of kernel inflation, and finally UNIX will burst never to be resurrected.

Of course, there are much smarter operating systems, providing generic services by a small nucleus, and surely it will be fun to work with them. However, some old-fashioned customers still insist on using UNIX. Therefore, we try to keep it alive and write Servers.

Potential Pitfalls of a Distributed Audit Mechanism

Chii-Ren Tsai

VDG Inc.
4901 Derussey Parkway
Chevy Chase
Maryland 20815

Virgil D. Gligor

Electrical Engineering Department
University of Maryland
College Park, MD 20742

Matthew S. Hecht

IBM Corporation
182/3A24, 800 N. Frederick Ave.
Gaithersburg, MD 20789.
{crtsai,gligor}@eng.umd.edu
uunet!pyrdc!ibmsid!{crtsai,gligor,hecht}

ABSTRACT

We identify several generic problems of distributed audit, such as cyclic generation of audit records, user ID mapping, and a consumer-producer problem for audit trails, that are not addressed by previous work. To study these problems, we built an experimental prototype of a distributed audit mechanism with centralized control based on the AIX 2.2.1 audit subsystem [IBM88a, Hec88a], TCP/IP [Tsa89a], Network File System (NFS) [San85a], and Distributed Services (DS) [Sau87a]. The prototype includes a central *auditor* role that can invoke and revoke auditing for remote hosts, can locate the *audit trail server* where audit trails are dumped, can perform audit system management, such as dynamically adding or deleting audit events on a per-user, per-group, or per-system basis, and can query the audit status of remote hosts. We highlight the AIX audit subsystem, present details of the implemented prototype, propose solutions to the identified problems, and discuss several open issues.

1. Introduction

How do you perform audit and integrate audit trails for a cluster of LAN-connected systems, each of which contains an audit subsystem? What are the problems you may encounter when you invoke distributed audit?

The purpose of this work is to study several generic problems of distributed audit by enhancing the functionality of the existing AIX audit subsystem to support audit in a distributed domain. Several features are added atop the local audit subsystem to allow selection of both local and distributed audit. Several simple mechanisms are implemented for synchronizing system clocks, for configuring secondary audit-trail servers when the primary audit-trail server is not available, and for preventing both the audit-trail filesystem and the local audit-trail filesystem from overflowing their allocated space.

The only other example of relevant work in this area is auditing for SunOS MLS distributed system [Sib88a]. That work focuses on the overall design for the SunOS audit subsystem and the method to collect audit records at audit-trail servers based on NFS in a distributed environment. In our work, we use NFS or DS to collect audit records at an audit trail server. In addition, we implement an auditor role for centralized control and discuss several generic problems of distributed audit, such as user ID mapping, consumer and producer problems of the audit trail server, and the cyclic generation of audit records, which are not addressed in [Sib88a].

In Section 2, we summarize the design of the AIX audit subsystem previously described in [Gli87a, Hec88a]. Section 3 presents the design of a central auditor based on the existing AIX audit subsystem and utilizing either NFS or DS. Section 4 discusses generic problems of distributed audit and their solutions. Section 5 states several open issues.

2. The AIX Audit Subsystem

AIX 2.2.1 provides NFS, DS, and an audit subsystem. The AIX audit subsystem consists of an audit daemon, **auditbin**, six audit commands, **audit, auditapp, auditstream, auditpr, auditselect** and **auditwrite**, and five system calls: **audit, auditbin, auditevents, auditlog** and **auditproc**. In addition, the AIX audit subsystem supports two mechanisms for collecting compressed audit records: *stream collection* and *bin collection*. For stream collection, audit records are written into a circular buffer inside the kernel. The audit records are retrieved from the special device **/dev/audit** and are usually pipelined to a printer or a monitor for interactive output. For bin collection, two bin files, **/audit/bin1** and **/audit/bin2**, serve as the buffers for storing audit records. Whenever a bin file size reaches a predefined threshold, the file is dumped onto the system's audit trail and the other bin file starts collecting audit records. After the dumping process is finished, the first bin file is again available for use. (This mechanism is called *buffer-swapping*.) Bin collection is slower than stream collection since audit records are stored in a regular file instead of a kernel buffer. However, the threshold of the bin file, which is specified in **/etc/security/config**, is configurable and can be made much larger than the kernel buffer. Therefore, audit bin files can also be used as temporary audit trails if necessary.

Audit events can be selected on a per-user basis. All registered audit events are specified in the file **/etc/security/audit/events**. A set of audit events can be defined as an audit class to simplify system management for audit event selection. The AIX audit subsystem is designed to be easily extensible for newly added trusted applications that are installed as setuid-root programs. The event names and the audit-record formats of the new events are added to the audit event file **/etc/security/audit/events**.

Whenever auditing is turned on and bin collection is specified, the daemon process auditbin is forked to direct audit records to one of the audit bins (i.e., **/audit/bin1** or **/audit/bin2**) and to execute the commands listed in the file **/etc/security/audit/cmds**. Usually, these commands are defined to append audit bins onto an audit trail. If stream collection is started, the command **auditstream** can be used to create a channel for reading of the audit records.

For audit trail analysis, **auditselect** and **auditpr** provide the capability to select and print audit records by: the audit event, command, user login ID, process real user ID, process effective user ID, process ID, parent process ID, and result of the operation or time stamp.

3. Distributed Audit with Centralized Control

In a distributed environment, it is generally tedious and time-consuming to turn on/off auditing for each system individually. Consequently, it would be more efficient to provide a central auditor who controls and manipulates all the audit subsystems. Even though it supports two collection modes and audit records can be dumped to a remote audit trail server if the appropriate configuration is made in a distributed environment, AIX 2.2.1 audit subsystem needs to be extended with a central auditor. To allow an audit subsystem to be used in both a stand-alone system and a distributed system, we use the client-server model to design distributed audit with centralized control.

3.1. The Client-Server Model

As shown in Figure 1, each system has an audit daemon (server), which may take instructions from either the local auditor for local auditing or a remote auditor for distributed audit. This concept can also be applied to distributed audit for heterogeneous systems.

3.2. The Central Auditor Role

The auditor has the following responsibilities:

1. turn on/off remote auditing,
2. query the auditing status of each system,
3. configure audit profiles and audit trail servers, and
4. analyze audit trails.

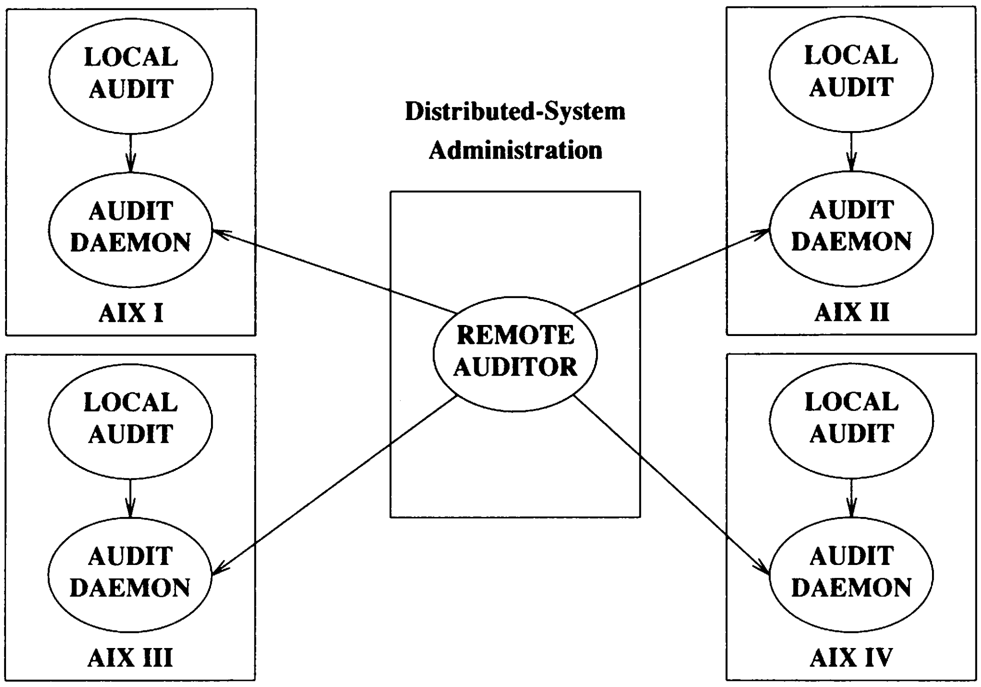


Figure 1: *Client-Server Model for Distributed Auditing*

An audit subsystem needs several profiles for configuration and system administration, such as *audit event profile, system audit profile* and *user audit profile.* In the AIX audit subsystem, the audit event profile, system audit profile, and user audit profile are specified in **/etc/security/audit/events, /etc/security/config,** and **/etc/security/passwd**, respectively. The audit-event profile contains all the defined audit events in the system, and can be extended with new events as new applications are added to the system. The system audit profile includes the audit system configuration and management information, such as the auditing mode, audit trails, and audit classes (which are sets of audit events). The user audit profile, which can be changed by the auditor from time to time, specifies audit events or classes for users.

The central auditor directs each system to transfer its audit records to the primary audit trail server or to subservers if the primary server fails. The central auditor may utilize either the primary server for audit-record analysis. (The performance of audit record analysis can be increased if the auditor uses the audit trail server.)

3.3. The Audit Protocol

To support the auditing role, each audit daemon must comply with a protocol to execute certain instructions on behalf of the auditor. The protocol message flows are shown in Figure 2.

First, identification and authentication must be performed after a connection is established between the auditor and an audit daemon. Since cryptographic protocols for identification and authentication such as Kerberos [Ste88a], had not been supported at the time of this work the audit daemon can only authenticate the auditor with the auditor's network address, namely IP address and TCP port number. Therefore, the auditor must use a well-known port for the connection with the audit daemon and the audit daemon expects all the auditor command messages from specific hosts. After authentication succeeds, the auditor may turn on/off remote auditing or query the auditing status or perform audit system management, such as configuring audit-trail subservers or specifying audit classes for users. If the auditor wants to turn on auditing, the command "on" is sent to the audit daemon. Then, the audit daemon must synchronize its system clock with the auditor's system. This step is extremely important for distributed audit since clock discrepancies among systems makes audit-trail analysis difficult.

In addition to the command "on", commands "off", "query" and "config" represent instructions for turning off auditing, querying auditing status, and configuring the user audit profile and audit trail subservers, respectively.

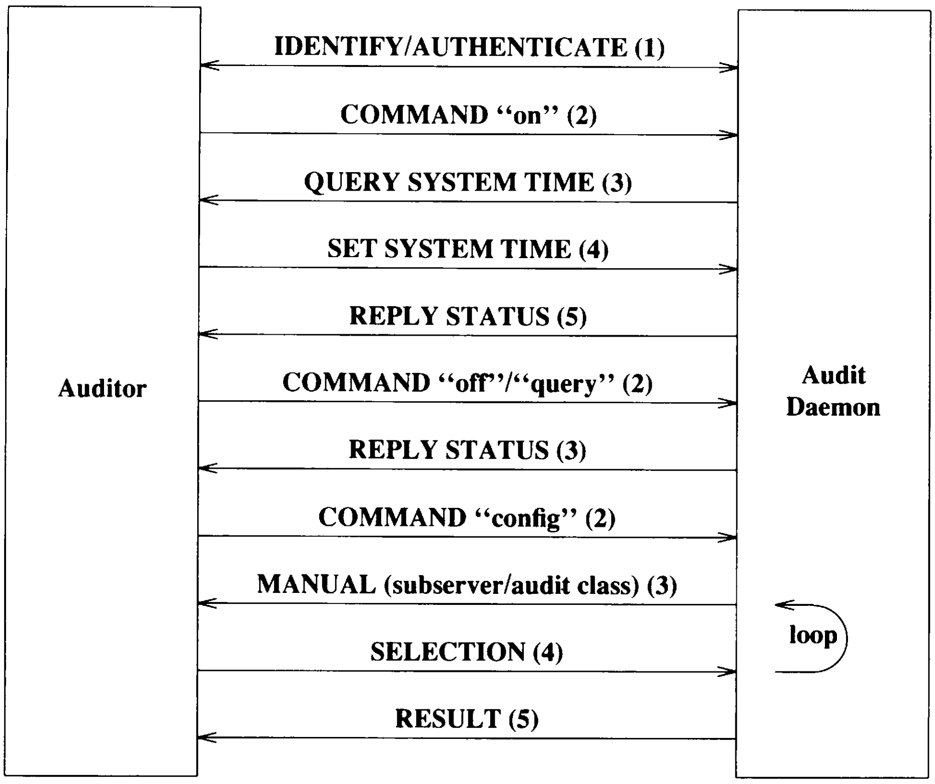


Figure 2: *Audit Protocol*

3.4. Implementation

The distributed audit prototype is based on TCP/IP in cooperation with NFS or DS. Since the auditor is configurable, each audit daemon can authenticate the auditor by utilizing its TCP port number. Thus, we reserve two well-known ports for the auditor and audit daemon, respectively. Alternatively, if the audit daemon of the auditor's host is disabled permanently, a single well-known port is sufficient for both the auditor and audit daemon. To synchronize the clock with the auditor's system, we use the DDN protocol RFC 868 [Age85a] to read the time from the auditor's system by sending an empty datagram to the well-known port 37 at the auditor's system. Then, we set the local system time to be the same as the auditor's system time. In a local area network, this way of synchronization is acceptable since transmission delay between the two hosts is insignificant.

Much of the audit command implementation is supported with a manual-driven interface. Audit system management data are transmitted to the auditor by audit daemons through TCP connections. Since audit system management may not be done frequently, it is not necessary to cache remote user profiles in the central auditor. This saves memory space and eliminates the need for frequent consistency checks for the audit profiles in the central auditor.

In addition to the audit protocol, we also implemented a daemon, **monitor**, to prevent exhaustion of the local audit system **/audit** and of the audit-trail filesystem in the audit trail server. We also implemented a command, **dstrail**, to handle the failure and the recovery of the primary audit trail server. Before the auditing of a system is turned on, the auditor can interactively set upper limits or high-water-marks for the sizes of the local audit-trail filesystem **/audit**, of the audit-trail filesystem in the audit-trail server, and of the system's audit trail. The ability to limit the audit trail size allows the auditor to dynamically allocate space for each audit trail and to prevent the fastest growing audit trail from exhausting the audit trail filesystem. The daemon **monitor** is forked by the audit daemon whenever auditing is turned on, and periodically polls filesystem sizes and checks them against their upper limits. If any one of those parameters exceeds its upper limit, auditing is turned off and **monitor** immediately exits to the auditor. Manual intervention by the auditor becomes necessary.

The command **dstrail** in the command file **/etc/security/audit/cmds** is executed whenever audit records need to be appended to an audit-trail file. The audit trail of each system is saved in a specific file on the audit-trail filesystem of the audit-trail server. If the audit-trail filesystem of the primary server is not mounted on a local directory, **dstrail** tries to mount it using NFS or DS. Whenever the mount-over action or the communication link fails, **dstrail** switches the mount of the audit-trail filesystem to the highest priority subserver. To maintain data consistency, we assume that all audit trails have to be collected in the primary server whenever possible. Consequently, **dstrail** redirects the audit trail stored in the subserver back to the primary server whenever the primary server recovers. Therefore, it is unnecessary that the subservers keep a large repository for an audit-trail filesystem. However, for reliability reasons, it may be important to have another large audit-trail filesystem in one of the subservers in case the primary server's failure cannot be recovered in time.

3.5. Audit Events

In this prototype, we extend audit event-selection from a per-user basis to a per-user, per-group, or per-system basis. In the audit event file **/etc/security/audit/events**, 171 audit events are defined. These audit events can be partitioned into *kernel-level events* and *user-level events*. Kernel-level events consist of 110 system-call events and 6 TCP/IP kernel events. It is helpful to understand the correlation between user-level events and kernel-level events before audit classes are defined. On one hand, kernel-level events provide certain audit records, which cannot be collected from the auditing of user-level events. On the other hand, it is unnecessary to turn on all the kernel-level events because additional audit records, especially those generated for certain system-call events, may increase the complexity and decrease the speed of audit-trail analysis, and may occupy significant amount of disk space in the audit trail filesystem. Therefore, defining audit classes in terms of security-relevant system-call events [Cug89a] and kernel-level network events [Tsa89a] makes auditing more efficient.

3.6. Audit Records and Audit Trail Analysis

Audit records contain *common* audit fields and *event-specific* audit fields. Common audit-record fields include the (1) event name, (2) command name, (3) user's login ID, (4) process real user ID, (5) process effective user ID, (6) process ID, (7) parent process ID, (8) result of the action, and (9) time stamp. Event-specific audit records for system-call events consist of input arguments and returned values; TCP/IP network events also include network address and port numbers.

The user login ID can be used for accountability in a single system. However, it would be helpful to have an extra ID that can be used for accountability in a distributed environment, such as the *audit ID* in [Sib88a] or the Universal user ID (UUID) in [Apo87a], which is invariant for each user in a distributed system. Providing an extra auditable ID significantly simplifies audit-trail analysis. However, because a special audit ID or a universal UID would require AIX kernel changes, which were unwilling to make for this project, user audit ID is represented by the user's login ID, process ID, parent process ID and time stamp, network addresses and port numbers. This audit ID can be used to trace a user's behavior in a distributed environment.

To illustrate the complexity of distributed audit when universal user identities are not provided, consider the example of use behavior shown in Figure 3. A user U_1 logs in to system A at time t_0 and then logs in to system B as user U_2 at time t_1. U_2 logs in to system C as user U_3. At time t_3, U_3 logs in to System B as user U_4 and then logs out of system B at time t_4. U_3 logs out of system C at time t_5. At time t_6, U_2 logs out of system B. U_1 continues his/her activities in system A and then logs out at time t_7. To trace the user U_1's activities in the environment, we first analyze the audit records for U_1 between t_0 and t_1 and then analyze the audit records for U_1 on other systems between t_1 and t_6 before analyzing the audit records for U_1 after t_6.

The key audit requirement of the above example is to discover that U_1 logs in to other systems under the identity of other local users. After the login user's ID is identified, we can easily select the audit records for that user out of that system's audit trail. We have to use IP Internet addresses, TCP port numbers, and time stamps to locate the first audit record of a connection on another system and then to extract the user's login ID from that record. Similarly, subsequent login user's IDs can be discovered. Therefore, we can select the audit records of a user's activities on any system during a login period by using the login user's IDs obtained from above. If the audit records happen to be collected from two different login sessions, for example, one from console login and the other from the remote login, we have to distinguish audit records for different login sessions. At this point, we can take advantage of the process ID and the parent process ID of each audit record to make the distinction. Thus, a user's network activities can be traced precisely.

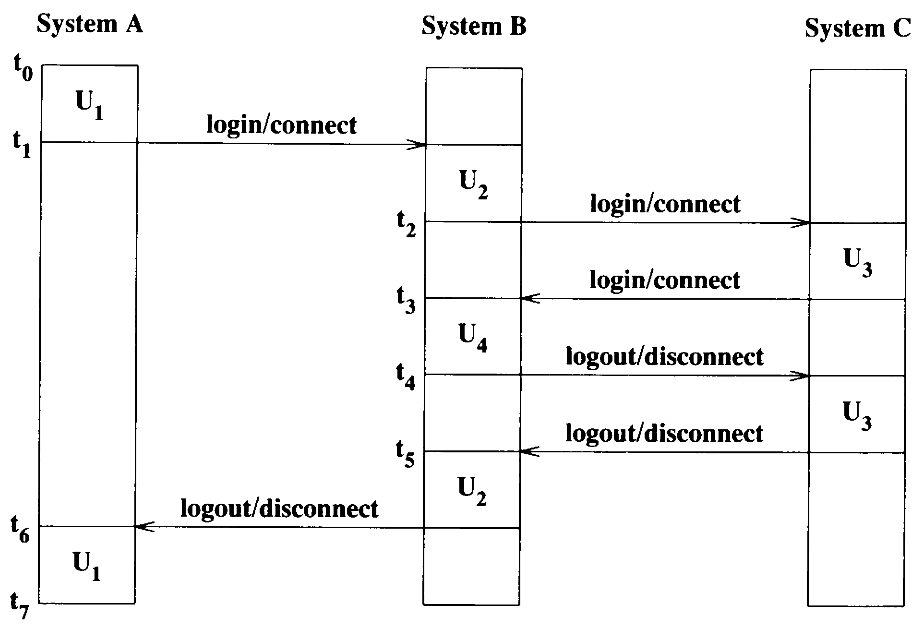


Figure 3: *A User Network Behavior*

4. Generic Problems for Distributed Audit and Solutions

We have identified several problems of distributed audit such as the consumer-producer problem for audit trails, cyclic generation of audit records, and user ID translation for audit-trail analysis. In this section, we illustrate the consumer-producer problem and the cyclic generation of audit records by describing audit scenarios and experiments. Also, we present a user ID mapping problem for audit-trail analysis.

4.1. The Consumer-Producer Overflow Problem and the Cyclic Generation of Audit Records

To illustrate the consumer-producer problem for audit trails, we first conducted experiments of distributed audit based on the prototype described in the previous section using NFS or DS. The underlying protocols for NFS and DS, which can coexist in the network, are TCP/IP and the IBM LU 6.2 protocol, respectively. The environment of the experiments is shown in Figure 4, where the audit-trail server with 1.01 gigabytes of disk memory also served as the gateway between the token ring and the Ethernet.

4.1.1. Preliminaries of a Distributed Audit Experiment

We configure a filesystem, **/netaudit**, with 100 Mbytes at the audit-trail server as the central pool for audit trails. As mentioned, the AIX audit subsystem provides two mechanisms to collect audit records, namely, bin collection and stream collection. It is easier to identify the consumer-producer problem by using bin collection. Therefore, we configure a filesystem, **/audit**, with 7.5 Mbytes in each system as the filesystem for two local audit-bin files, namely **/audit/bin1** and **/audit/bin2**, which are used as the buffers to store compressed audit records. Also, each system mounts **/netaudit** from the audit-trail server over a local directory, namely **/audit/netaudit**, by using either NFS or DS (both NFS and DS can mount a remote directory or file over a local file system). The audit trail of each system will be stored as a file named with its host name in **/audit/netaudit**. The filesystem **/netaudit** must be owned and protected by the superuser root of the audit-trail server, so that audit trails cannot be accessed by users without the root privileges of the audit-trail server. However, NFS has a superuser restriction since by default an NFS client's superuser cannot have superuser privileges on an NFS server (unless the superuser restriction is removed). The superuser restriction can be removed if access to the root of an NFS server is not tightly controlled, or if the NFS servers and clients are administered by a single person. In practice, the superuser restriction is usually removed whenever NFS is used. In contrast, DS supports the capability of user ID translation, so that a DS client's superuser can have the superuser privileges on **/netaudit** only if the superuser ID of each system is translated into the superuser ID of the audit-trail server. (Note that a DS mount cannot always succeed. For instance, if the audit-trail server is not in the same network as the client, the DS mount will fail because DS cannot pass through a gateway.) Therefore, NFS and DS are used to mount the audit-trail filesystem **/netaudit**.

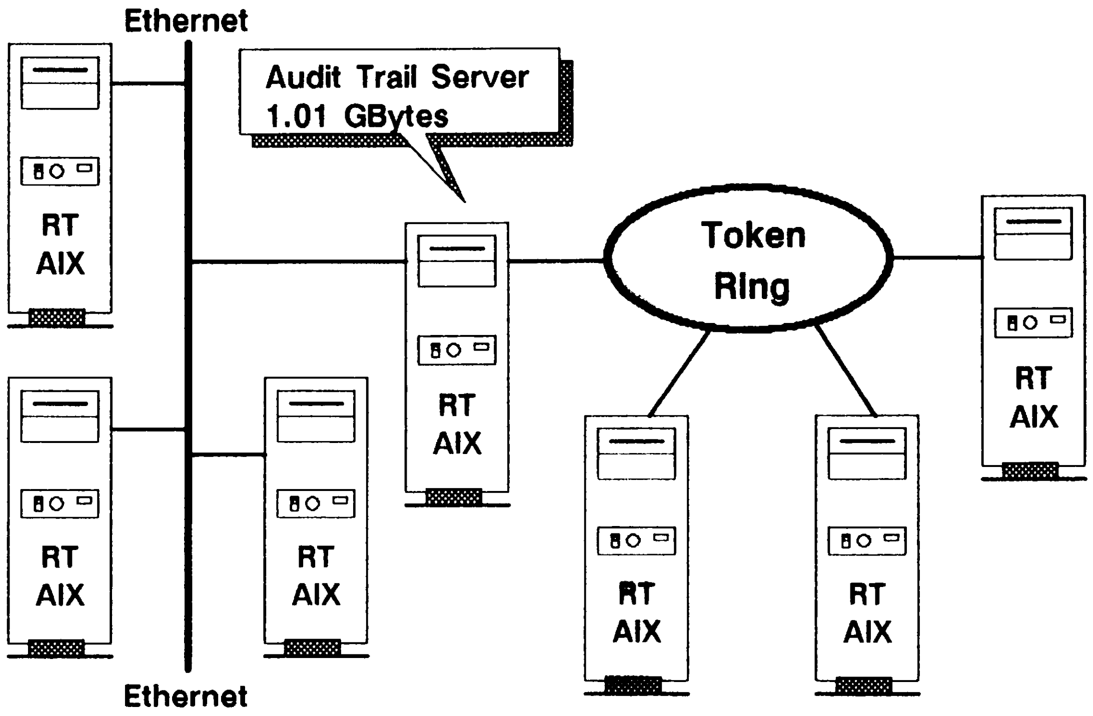


Figure 4: *Network Configuration*

We configure one of the RTs as a *single-system image* (SSI) of the audit-trail server. A single-system image is the configuration of several systems to appear to be one system. The SSI administrator (server) and the SSI clients share user files and directories. Those files and directories include *system administration files,* such as the user password file, user group files, and user configuration files, *users' home directories* and *sharable application programs,* namely programs in **/usr/bin, /usr/lib, /usr/pub, /usr/lpp, /usr/include, /usr/database, /usr/dos/bin, /usr/datamgt, /usr/games,** and **/usr/man**.

4.1.2. Experiments and Results

We specify all auditable events for all users including the root on each system. Also, if an audit bin filesystem, **/audit**, reaches a predefined upper limit or high-water-mark, the auditing of that system will be automatically turned off. When the auditing of each system is turned on, the first bin file, either **/audit/bin1** or **/audit/bin2**, starts collecting audit records. After the first bin file reaches a predefined threshold, namely 20 Kbytes in these experiments, the second bin file will start collecting audit records while the data in the first bin file are dumped to the system's audit trail in **/audit/netaudit**. If the second bin file reaches the threshold and the first bin file is still being dumped onto the audit trail, the second bin file will continue collecting audit records until the first bin file is available. Repeated occurrences of this phenomenon may occasionally exhaust the filesystem of the audit bin files.

In the series of experiments, only the auditing of the single-system image host is occasionally turned off because audit records are generated faster than those records can be transferred to the audit trail, thereby causing the **/audit** filesystem to run out of space. Also, the single-system image host mounts various directories and files from the server using DS or NFS. Because NFS servers are stateless, the SSI client generates many more audit records than the SSI administrator and other non-SSI hosts. A set of recorded data and its chart are shown in Table 1 and Figure 5, respectively, to illustrate this situation.

In Table 1, the sequence number represents the event sequence; Audit Bin denotes accumulated audit records in one of the audit bins; and Consumption Time represents the time required to transfer audit bins to the audit trail. For instance, it takes 14 seconds to transfer 40926 bytes to the audit trail and, in the meantime, the system produces 184032 bytes of audit records for Sequence 0. The next sequence of Table 1 (Sequence 1) shows that it takes 15 seconds to dump 184032 bytes of audit bin to the audit trail and the system generates 490720 bytes of audit record during this period. At Sequence 30 of Table 1, the auditing is turned off because the consumption rate is not high enough and the **/audit** filesystem runs out of space.

Seq. #	Audit Bin (bytes)	Consumption Time (sec)	Seq. #	Audit Bin (bytes)	Consumption Time (sec)
0	40926	14	16	572469	47
1	184032	15	17	736102	29
2	490720	42	18	613444	39
3	1042739	64	19	1083664	52
4	1226652	70	20	1104034	89
5	1267565	66	21	1880927	92
6	797441	51	22	1247191	66
7	1022216	58	23	654362	43
8	1165397	54	24	879107	67
9	1144906	64	25	1635555	108
10	879101	36	26	1247048	68
11	306653	25	27	838260	73
12	552030	39	28	879180	57
13	920116	50	29	1206280	82
14	1226664	62	30	1533285	auditing off
15	1472050	83			

Table 1: *Audit Bin and its Consumption Time*

Seq. #	Audit Bin Production Rate (bytes/sec)	Audit Trail Consumption Rate (bytes/sec)	Seq. #	Audit Bin Production Rate (bytes/sec)	Audit Trail Consumption Rate (bytes/sec)
0	1637	3273	16	6090	15660
1	7668	1705	17	14155	11009
2	19628	7361	18	14266	17118
3	21280	10014	19	18683	10576
4	16140	13720	20	16727	16419
5	14739	14263	21	17578	10318
6	10094	16045	22	12108	18261
7	15726	12268	23	8842	16853
8	16648	14603	24	15157	11282
9	17889	18209	25	19945	10720
10	11879	15471	26	10221	13406
11	7301	20930	27	11176	16627
12	16236	9019	28	10989	10478
13	16729	10036	29	17232	12559
14	20790	15595	30	16311	12832
15	18633	15527			

Table 2: *Production Rate and Consumption Rate*

Audit-record production and consumption rates are computed and plotted in Table 2 and Figure 5, respectively. Black bars represent production rates, and dotted bars denote consumption rates. Figure 5 shows that the production rates are much higher than the consumption rates during the last few sequences before auditing is turned off.

4.1.3. Discussion

The producer-consumer overflow problem of the audit subsystem can be characterized by a simple model, which consists of production and consumption rates and buffer availability. In the model, P_r and C_r represent the production rate and the consumption rate, respectively. The number of buffers required to resolve the problem can be characterized by the following conditions.

- $P_r << C_r$One buffer is sufficient.
- $P_r \approx C_r$Two buffers are sufficient for buffer swapping.
- $P_r >> C_r$Multiple buffers and flow control are necessary.

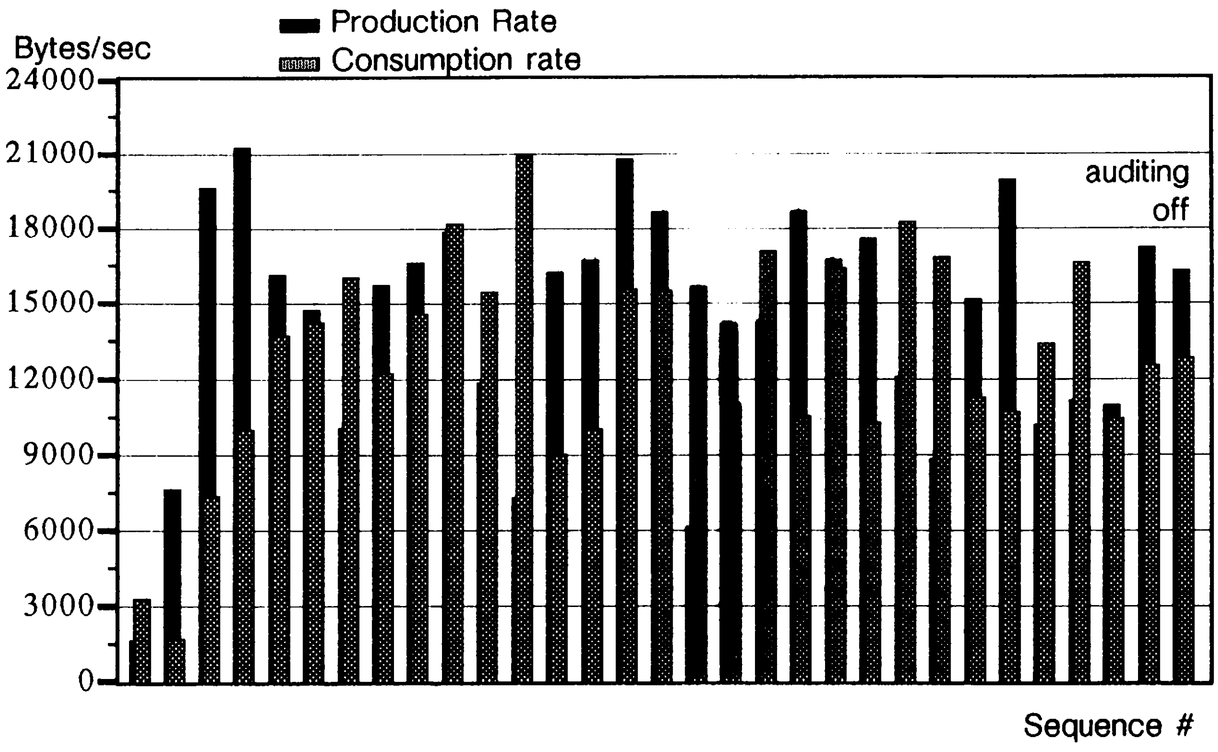


Figure 5: *Production Rate and Consumption Rate vs. Sequence Number*

The AIX audit subsystem uses buffer swapping to collect audit records in bin collection. It is assumed that the production rate is more or less the same as the consumption rate. Generally, buffer swapping can satisfy most of practical situations. However, if the production rate is continuously greater than the consumption rate for a long period of time, flow control is needed to tolerate a consistently high-load transient state.

The production rate is related to the number of audit events, the load of the system, and the access times to buffers. The consumption rate is affected by the network traffic and the access time to the audit trail if a remote device is used. Since the parameters affecting the consumption rate cannot be tuned, solutions to the overflow problem must be providing by controlling the parameters which may affect the production rate.

Audit Event Selection

In the audit experiments, we turned on all the audit events. We performed an experiment to estimate the additional audit records generated by kernel-level audit events. In the experiment, we collected audit records for a user who logs in to a system and then executes a shell-script program. The shell-script program and the results are shown in Table 3.

As shown in Table 3b, 158953 bytes of audit records are generated for all events, and 4669 bytes of audit records are produced when user-level audit events are turned on for the user. However, as shown in Table 3c, 90% of the 4669 bytes are also generated by kernel-level events associated with the superuser, *root*. Consequently, audit records generated by kernel-level events occupy more than 99% of the records if all audit events are turned on for users. As shown in Table 3d, less than 50 percent of the audit records are generated by security-relevant events, which are marked with stars. Therefore, if only security-relevant events are turned on for users, the number of audit records is significantly reduced and the producer-consumer overflow problem may be avoided.

Avoidance of Cyclic Generation of Audit Records

Two groups of events are *cyclicly dependent* if a group of events activates another group and, vice versa, the latter group subsequently activates the former group. Whenever cyclic dependencies of audit events exist, they may start cyclic generation of audit records. In Figure 6, for example, the auditor logs in to one

1.	**mkdir** testdir	7.	**rmdir** testdir
2.	**cd** testdir	8.	**ps -ef**
3.	**cp** ../tfile tfile	9.	**df**
4.	**cat** tfile	10.	**who**
5.	**rm** tfile	11.	**date**
6.	**cd** ..	12.	**logout**

Table 3a: *Shell script Program*

Audit Event Selection	Audit Records	Size of Records (Bytes)	Number of Records	Generation Period (seconds)
I. User-Level Events		4669	43	9
II. All Audit Events		158953	1872	9

Table 3b: *Audit Record Sizes*

Kernel-Level Audit Event	Number of Records
kconnection	14
kimport_data	12
kexport_data	7
ksocket_creat	2
auditproc	2
stcb	1
ioctl	1
exece	1

Table 3c: *Event Occurrences for I*

of the systems, x, and then remotely logs into the audit-trail server to print the audit trail of this system (x). If we turn on kernel-level network events in system x, audit records of kernel-level network events are constantly generated because audit-record generation and the printing of audit records are cyclicly dependent on each other. That is, system x receives outputs from the audit-trail server that implicitly

Kernel-Level Audit Event	No. of Records	Kernel-Level Audit Event	No. of Records
close	460	sgetpid	22
read*	296	kexport_data*	17
ossig	225	ksocket_creat	17
seek	198	exece	14
write*	128	staffs	14
open*	60	alarm	13
ioctl	59	rexit	11
ulimit	58	wait	11
gtime	50	fcntl*	7
kconnection	39	lock	7
sbreak	33	sigblock	7
saccess*	24	stat	7
kimport_data*	23	lstat	6
fork	22	dup	6

Table 3d: *Event Occurrences for II*

Table 3: *An Audit Event Selection Experiment*

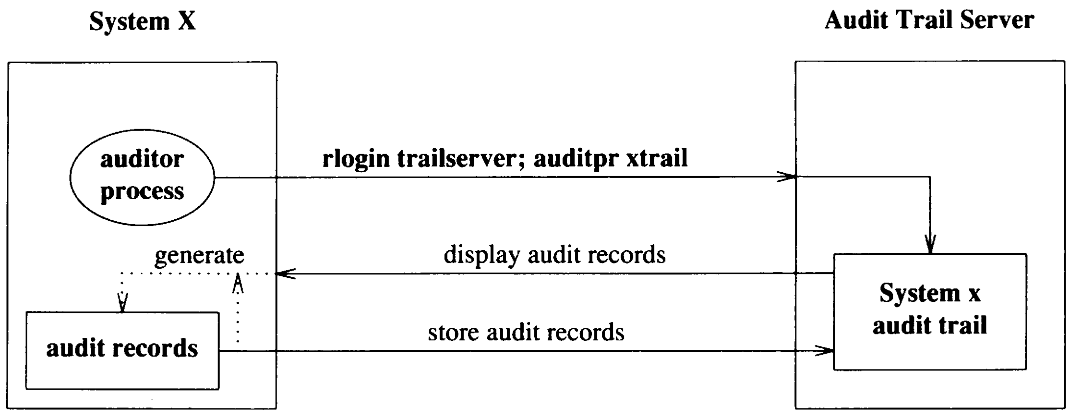


Figure 6: *Cyclic Generation of Audit Records*

generate more audit records for the kernel-level network events because the auditor process is printing the audit trail. Consequently, system x appends the newly generated audit records to its audit trail on the audit-trail server. The action of appending the audit records to the audit trail also generates audit records. Again, whenever the newly generated audit records are printed, additional audit records are generated. Cyclic generation of audit records degrades system performance severely whenever the rate of audit record generation is higher than the rate of printing audit records. In fact, this problem also illustrates another instance of the producer-consumer overflow problem mentioned above.

Cyclic dependencies among audit events cannot be completely avoided because they are inherent to the system structure. However, cyclic dependencies do not always cause cyclic generation of audit records. Thus, experiments must be performed to determine the potential cyclic generation of audit records. After potential cyclic generation of audit records is discovered, the events which have cyclic dependencies must be defined in mutually exclusive audit classes. If only one of two mutually exclusive audit classes is turned on, then a cyclic generation of audit records can be avoided. Moreover, if the removal of certain events from an audit class can prevent a cyclic generation of audit records, it is not necessary to turn off the entire event audit class.

Flow Control

If the selection of audit events and the prevention of cyclic generation of audit records cannot eliminate the producer-consumer overflow problem, we have to perform flow control on audit-record production since we cannot tune the consumption rate. For stream collection, flow control is already accomplished because a circular buffer inside the kernel is used to collect the audit records. For bin collection, if we place the buffers **/audit/bin1** and **/audit/bin2** onto a remote site, we can reduce the production rate but also significantly degrade the overall system performance. If system performance is not a major concern and flow control can be enforced, a remote filesystem in the audit trail server can replace the local **/audit** filesystem for the buffers whenever **/audit** reaches a predefined high-water-mark.

Increasing the Size of the Audit Filesystem

The size of the audit-bin filesystem is significant if the generation rate, on average, is almost equal to the consumption rate. Increasing the size of audit-bin filesystem may delay, or occasionally prevent, the filesystem from being exhausted.

4.2. The User ID-User Name Mapping Problem for Audit Trail Analysis

To trace a user's activities in a distributed system, the auditor has to select the audit records associated with the user from the audit trails. If users do not have universally unique user IDs in the distributed system, the auditor cannot identify the users' records unless user ID to user name *mappings* have been done. This is the case even if user activities on remote systems can be associated with local user identifiers (as in the example of Figure 3). For example, users u_1, u_2 and u_3 have the same user ID 201 in system1, system2 and the audit trail server, respectively. If no provision for making user IDs unique has been made, the auditor in the audit trail server may mistakenly take u_1 on system1, u_2 on system2 and u_3 on the audit trail server as representing the same user. Therefore, the auditor must have the mapping between user IDs and unique user names for each system during analysis. For simplicity, the **/etc/passwd** files can be mounted on a directory in the audit trail filesystem. For instance, the **/etc/passwd** file of system1 can be mounted on the audit-trail filesystem **/netaudit** over the file **/netaudit/system1.passwd**. Consequently, the auditor can easily map a user ID to a unique user name.

5. Open Issues

Scaling the Number of Systems. The consumer-producer overflow problem in distributed audit may be exacerbated when the number of systems increases. Furthermore, audit-event selection appears to complicate this problem. At this point, we cannot provide the statistics of relating network bandwidth and audit-event selection. Such statistics are necessary for audit buffer and file configuration in distributed systems.

Scaling the Number of Audit-Trail Servers. The creation of a central auditor for distributed audit simplifies audit system management. Consequently, the prototype of distributed audit with centralized control discussed herein can be extended to support multiple audit-trail servers, especially for an environment with a large number of LANs, since audit-trail servers for each system are individually configurable. The storage capacity (e.g., for write-once optical disk or even 8mm tape) and transfer rate of each server's audit trail filesystem may affect the period for retaining audit trails. Therefore, further study is needed to determine such requirement for each audit-trail server and audit-trail retention.

Auditing in the Andrew Computing Environment. Another issue is that of whether the type of distributed computing environment can obviate "micro-event" auditing. In a distributed computing environment based on the Andrew File System [Sat89a], "macro-event" auditing may suffice at trusted servers with no auditing at all at untrusted workstations; this greatly simplifies distributed audit. However, for a cluster of trusted Andrew servers in the same administrative cell, we would still need a central auditor role and audit trail servers.

Acknowledgments

We are grateful to Janet A. Cugini for her insightful suggestions on this paper. We also thank Mark E. Carson for his help with the graphic presentaion and Tom Tamburo and Gary Kennedy for their support of this effort.

References

[Age85a] Defense Communication Agency, *DDN Protocol Handbook*, Dec. 1985.

[Apo87a] Apollo, "Domain: Network Computing System (NCS) Reference," Order No. 010200, Revision 00 (June 1987).

[Cug89a] J. A. Cugini, S. P. Lo, M. S. Hecht, C. R. Tsai, V. D. Gligor, R. Aditham, and T. J. Wei, "Security Testing of AIX System Calls Using Prolog," pp. 223-237 in *Proceedings of the 1989 Summer USENIX Conference*, Baltimore, Maryland (June 1989).

[Gli87a] V. D. Gligor, C. S. Chandersekaran, S. Chapman, L. Dotterer, M. S. Hecht, W. D. Jiang, G. L. Luckenbaugh, and N. Vasudevan, "Design and Implementation of Secure Xenix," pp. 208-221 in *IEEE Transactions on Software Engineering, 13:2* (Feb. 1987).

[Hec88a] M. S. Hecht, A. Johri, R. Aditham, and T. J. Wei, "Experience Adding C2 Security Features to Unix," pp. 133-146 in *Proceedings of the 1988 Summer USENIX Conference*, San Francisco, California (June 1988).

[IBM88a] IBM, *Managing the AIX Operating System, Version 2.2.1*, 1988.

[San85a] R. Sandberg, "Design and Implementation of the Sun Network Filesystem," pp. 119-130 in *Proceedings of the 1985 Summer USENIX Conference* (June 1985).

[Sat89a] M. Satyanarayanan, "Integrating Security in a Large Distributed System," pp. 247-280 in *ACM Transactions on Computer Systems, 7:3* (Aug. 1989).

[Sau87a] C. H. Sauer, D. W. Johnson, L. K. Loucks, A. A. Shaheen-Gouda, and T. A. Smith, "RT PC Distributed Services Overview," pp. 18-29 in *Operating Systems Review, 21:3* (July 1987).

[Sib88a] W. O. Sibert, "Auditing in a Distributed System: Secure SunOS Audit Trails," pp. 82-90 in *Proceedings of the 11th National Computer Security Conference*, Baltimore, Maryland (Oct. 1988).

[Ste88a] J. G. Steiner, C. Newman, and J. I. Schiller, "Kerberos: An Authentication Server for Open Network Systems," pp. 191-202, in *Proceedings of the 1988 Winter USENIX Conference*, Dallas, Texas (Feb. 1988).

[Tsa89a] C. R. Tsai, V. D. Gligor, W. Burger, M. E. Carson, P. C. Cheng, J. A. Cugini, M. S. Hecht, S. P. Lo, S. Malik, and N. Vasudevan , "A Trusted Network Architecture for AIX Systems," pp. 457-471 in *Proceedings of the 1989 Winter USENIX Conference*, San Diego, California (Feb. 1989).

The Xprint printing system

P. Seghin
D. Buyse

Siemens Software S.A.
Rue de Neverlee, 11
B5810 Namur (Rhisnes)
seg@swn.siemens.be
dbu@swn.siemens.be

ABSTRACT

The penetration of UNIX systems in large commercial environments is now a fact, and industry observers foresee an even higher growth for the future. At the same time, computer sites interconnect more and more heterogeneous machines from different manufacturers but with one common denominator: they all run a particular version of UNIX.

It is important to be able to access resources from any of the machines, quickly and reliably. The differences between machines and methods of accessing the resources on them should be invisible to the user. This does not mean, however, that such differences are invisible to the administrators of the organization or individual machines.

In such a situation, a printing system is no exception. Most of the existing UNIX spoolers although they are distributed, lack the administration and configuration functions required by a professional printing environment.

Therefore, Siemens has developed a networked printing system, Xprint, that is distributed, portable on the widest range of architectures, easy to install, use and administer, supports the most commonly used printer types and offers modern user interfaces (toolkit, command line and graphical interfaces). Furthermore, Xprint respects the ECMA [Ass88a] standards and uses XPG3 interfaces by preference.

1. Requirements for a modern printing system

Since the traditional UNIX spoolers are increasingly being regarded as inadequate services, Siemens has looked for a robust and commercially acceptable printing system.

An extensive study induced a technology that fulfills following requirements:

1.1. Networking

- **Transparency.** Xprint does not discriminate between local and remote spooler objects. All objects are referred to as logical entities. They are not defined by topology dependent names (such as `object@host`), (i.e. the network configuration is hidden "for the user by the application)
- **DFS Independency and Stand alone Operation.** Xprint operates without the presence of a Distributed File System (e.g. NFS, RFS, ...). All functions of the application (except for remote operations) are available on isolated nodes.

1.2. Administration

- **Centralized Administration.** Xprint allows for a simple, centralized administration from any node in the network. Also, spooler objects can be administered network wide without "superuser" privileges (such as "root" in traditional UNIX systems)
- **Configuration.** All parameters of the configuration can be modified dynamically while the system is in operation. Without affecting any other system services, it is possible to:

- remove/add and enable/disable print services
- define/add new users
- enable/disable scheduling algorithms

1.3. Functionality

- **Printer Support.** A table driven back-end strategy is integrated in Xprint providing a quick support of new printers. To achieve this, the enabling technology relies on a flexible Printer Control Language description embedded in the software entity that directly controls the printer.
- **Scheduling.** Xprint allows for different scheduling algorithms, such as:
  - First In First Out
  - Large/Small Files first
  - Security scheduling

  Also, in order to maximize resource consumption, the system endorses the "late binding" concept (i.e. the assignment of a job to a printer is delayed until the specific device is ready to process the job).
- **Job Submission and Notification.** It is possible to submit jobs, simply by specifying the job attributes, instead of naming a specific printer. The system is then responsible for routing the job to an adequate printer.

  At any given time, a user is able to request the status of submitted jobs, regardless of his physical location on the network.

  A user definable and event driven notification service for exception handling during job processing is available.
- **Load Balancing, Quotas and Accounting.** Xprint enables an effective load distribution amongst equivalent print servers. The technology also provides a quota mechanism on a per user basis to control the consumption of spooler resources.

1.4. Implementation

- **Fault Resistance.** On the machine level, the technology was designed such that the crash of any host in the network doesn't restrict the print services on the active nodes. A distributed resource database mechanism automatically reflects all changes across the network and guarantees continuity of operation on the active nodes.

  On the network level, the necessary algorithms were embedded to allow for consistent operation in case of a network failure.
- **Security.** User access to spooler resources and services is denied if proper authentication and authorization fails. Furthermore, a proper distinction – in terms of functional capabilities and resource access rights – is made between the non-privileged user, the local administrator and the systemwide spooler administrator.
- **Interoperability.** Xprint allows for a cooperation with the UNIX V.4 spooler on remote systems and should interoperate with implementations of the Palladium [Han89a] printing system.

2. The architecture of Xprint

The Xprint printing system is composed of five basic entities. They are the Client, Server, Supervisor, DB_manager and the XP_daemon. Each entity is responsible for a part of the processing needed to satisfy any Xprint request. Each entity implied in the processing of a Xprint request may run on different machines in the network.

The Xprint system decomposition requires the definition of exchange protocols between the different entities.

2.1. The Client

The Client validates the user's request, invokes the entity that will execute the request and returns the results of the request and any possible error notification to the user.

The Client also checks the user's authorization with respect to the requested service.

The programs defining the Client are executed under the control of the process that invoked them. The Client interface is the lowest user interface provided by Xprint. The command line interface and the graphical interface of the Xprint are implemented with the Client interface.

2.2. The Server

The Server receives requests from both the Client and the Supervisor. It is responsible for administering the Supervisors, the jobs that will be processed by them and for scheduling the jobs between the different Supervisors.

The Server manages the Job Database by entering new jobs or modifying existing ones upon request of a Client, or by deleting the jobs from the Database on request from both Client and Supervisor. The Job DB consists of files located in the spoolin directory which is one of the Server's attributes.
A Job is described by one control file and the data files. The control file contains the job attributes given when it was submitted, a reference to the data files and the job state.

The Server is implemented in two levels of processes. One main process listens to the network. Each time a connection request is asked, the main process forks a child to read the request from the communication endpoint and handle it. This is to reduce the Server's bottleneck in increasing the number of requests which may be processed at the same time. The child's lifetime is the time needed to process the request.

One Server's child is called the *Scheduler*. It is permanent and responsible for selecting the jobs for the Supervisors. It receives requests from its evanescent brothers through a pipe created by the main process.

There may be several Servers running on the same host.

2.3. The Supervisor

The Supervisor is responsible for executing the jobs selected by its Server. Job execution implies the following steps:

- transformation of the user's data if required by the user.
- data presentation to the device.
- device control and job recovery in case of error.

In addition to and during the job processing, the Supervisor is able to receive requests from both the Client and Server entities to:

- send either the job or device status.
- interrupt the job processing.
- terminate itself.

Like the Server, the Supervisor is implemented in two levels of processes. This guarantees the Supervisor's availability during job processing. One child formats and sends the buffers to the device and performs the device error recovery, while the main process remains available to catch and answer external requests for job information or interruption, spoolout termination or system shutdown.

Currently there must be as many Supervisors as devices managed by Xprint. A Supervisor may be served by a remote Server. A device may be controlled by one Supervisor only. A Supervisor is controlled by one Server only.

2.4. The DB_manager

Introduction

The Database Manager stores all information about Xprint objects in the Xprint Database, except the print jobs. The Xprint Database is distributed and replicated on each host.

Since the majority of the Database transactions are made up of simple interrogations, this replication of data significantly enhances the performance. Moreover, a mechanism of authority transfer guarantees continuity of operation in case of master failure.

Components

The entire data base is present on all the hosts. The DB can be interrogated locally and does not involve interprocess communication. The interrogation requests are performed under control of the requesting process.

In case of update requests however, the other hosts must be notified of the modifications to be brought. This requires inter communicating processes running in the background on each host, ensuring the updates of the DB local copies: the DB daemons.

At any moment, one and only one host is the *DB master*. It owns the original data base. All other hosts contain copies of this original. An update request issued by any host is always converted into a request to the master. When the master daemon receives an update request from another host, it firstly updates its own DB (the original one), and then propagates the modification to the other daemons. A (non-master) daemon is only allowed to update his own copy of the DB after receiving an order from the master (and not directly after a user request).

Failure tolerance

In case of master failure, the Database can still be consulted on each other host, since it involves only local actions. Modifications, however, are no longer allowed. As this seems somewhat stringent, a mechanism must be implemented that ensures the nomination of another master. In order to extend this "saving principle" in case of failure of the "secondary master", the developed mechanism must allow an arbitrary number of recovery levels. A solution is the establishment of a hierarchy between the hosts, fixing their relative authority. This order among the hosts, known by every DB_manager, dictates unequivocally who is the master for a given failure configuration.

Startup

The startup of a DB_manager (A) can roughly be described as follows:

- (A) contacts whichever host to get the current master identity.
- If the current master (M) has more authority than (A), (A) remains a slave DB_manager. However, it contacts (M) to get a copy of the data base. The transaction ends.
- If (A) has more authority than (M), (A) will be the master: it contacts (M), specifying that it wants to become the new master. From that instant, (M) is locked for any modification request issued by other DB_managers. (M) sends a DB copy to (A).
- (A) broadcasts the news that it is master again and waits for incoming requests. When it gets this last message, (M) becomes a slave and then rejects update requests coming from Clients.

Shutdown

The shutdown implies a clean termination mode requested by the administrator, unrelated to sudden and unexpected events like power or hardware failure.

Before shutdown, the DB_manager is locked to deny new requests. If the DB_manager to be stopped is not the master, it may discard the outstanding requests and die silently.

The termination of the master DB_manager is less straightforward. A master DB_manager gives its secondary master notice of its impending shutdown by asking it to become master. When it gets the message, the secondary master broadcasts the news it is the current master. The old master waits for the message notifying that the secondary master has become the new master, and rejects the pending requests, instead returning the identity of the new master. This improves the response time by avoiding Clients having to wait for connection timeouts during their attempts to contact the terminating master.

2.5. The XP_daemon

Because it is distributed, the Xprint printing system needs to start programs on a remote machines Servers and Supervisors and also transfer files. For this task the *rsh* or *rcp* command seems not to be powerful enough, while the portability to non-UNIX systems is not guaranteed. Also, the system's availability must be guaranteed, even in the case of abnormal process termination, and a clean shutdown is to be controlled.

This is the XP_daemon's responsibility. One XP_daemon runs on each machine. The XP_daemon process is the first process appearing at local Xprint startup. It creates the DB_manager, performs the startup actions and then waits for requests from other entities. The XP_daemon is also responsible for controlling the availability of all the entities running on its machine and the correct termination of those entities when Xprint terminates. For that purpose, the XP_daemon maintains a process table, in which one entry is created for each active child.

3. The Implementation of Xprint

3.1. The Xprint Request Processing

3.1.1. The Startup

The Client syntactically checks the request and the user's authorization. It then checks whether the system is running and eventually creates the XP_daemon. The XP_daemon starts the local DB_manager. Because accesses to the Xprint DB are essential for Xprint request execution, the XP_daemon waits for the correct DB_manager initiation. It then executes the commands given in the startup file that allows the automatic enabling of the local Xprint resources (Server, Supervisor or Device) and makes them available to the other hosts already started. Each time a Server is created, it is notified by the XP_daemon whether the Job DB it owns must be cleaned or not (cold or warm startup).

3.1.2. The configuration requests

Processing configuration requests consists mainly of updating the Xprint DB. It is important to notice that the Xprint system will only update the Xprint DB over the network but will not inform the possible active entities that they have been reconfigured. It is the system administrator's responsibility to disable the entities concerned before reconfiguring them.

(1) The Client

- syntactically validates the request and checks the user authorizations
- invokes the master DB_manager to modify the original Xprint DB
- waits for acknowledgement

(2) The Master DB_manager

- locks the original Xprint DB to prevent local Client accesses
- updates the original Xprint DB
- unlocks the DB and acknowledges the waiting Client
- calls each slave DB_manager to update their copy

(3) The Client

- is freed from Xprint

(4) The Slave DB_manager

- locks the Xprint DB to prevent local Client accesses
- updates the Xprint DB and unlocks it

3.1.3. Job request processing

The complete execution of a job needs processing by the Client, Server and Supervisor entities. During its execution, the job's owner or system/device administrators may act on the job definition or status, or request information about the job.

It is necessary to establish exchange protocols between the three implied entities. The Server and Supervisor entities may be invoked from both the Client or Supervisor and Client or Server entities. The protocols must all be the same, therefore Xprint uses the Print protocol of [Ass88a].

1. Job submission and execution

(1) The Client

- syntactically validates the request and checks the user's authorizations
- builds the job parameter list consisting of both user options and Xprint defaults

- selects and invokes the Server that will handle the request
- waits for acknowledgement

(2) The Server

- adds the new job to its Job DB
- acknowledges the Client and returns the job unique identifier, if any

(3) The Client

- sends back to its caller a unique job identifier or an error notification
- is freed from Xprint

The Server

- calls its Scheduler to find a free device that will execute the job
- updates the job information by adding the Supervisor's name and address and the new job status for further information inquiries
- invokes the Supervisor to process a new job

  (It is important to notice that the parameter list transmitted to the Supervisor has exactly the same format as the one transmitted to the Server by the Client, but the information contained in it may be completed by the Server.)

(4) The Supervisor

- provides the device it controls with the user's data possibly transformed and preformatted as a job execution parameter
- performs the job recovery in case of device failure
- catches requests for job interruption, termination or information which may be requested by its Server or any Client
- informs its Server of the new job status, if the job terminated abnormally
- requests job deletion if the job has been successfully executed
- requests a new job to process from its server and waits for the Server's reply

(5) The Server

- calls its Scheduler to find a waiting job whose print options match the Supervisor characteristics
- awakes the Supervisor if a job has been found

2. Job information inquiry

(1) The Client

- syntactically validates the request and checks the user's authorizations
- invokes the Servers to send back information about the specified jobs

(2) The Server

- returns information to the Client on those jobs which satisfied both the selection criteria and the user's privileges.

  (The system administrator may request information on any job. A device administrator may request information on any job for the device(s) he/she administrates. A normal user may only receive information on the jobs he/she owns.)

3. Job attributes modification

(1) The Client

- syntactically validates the request and checks the user's authorizations
- calls the Servers for a job attribute modification
- waits for acknowledgement

(2) The Server

- if the request does not alter the Server's selection, modifies the non-active jobs which match the user's selection criteria and privileges

- if the modification alters the Server's selection, modifies the concerned jobs, applies to them the Server's selection algorithm, calls the job's new Server to add them and deletes the original(s) from its Job DB
- acknowledges the Client and returns the new job identifiers, if any

(3) The Client

- is freed from Xprint

4. Job deletion

(1) The Client

- syntactically validates the request and checks the user's authorizations
- calls the respective Servers for a job deletion request
- waits for acknowledgement

(2) The Server

- deletes from its DB the inactive jobs which match the user's selection criteria and privileges
- calls each Supervisor executing a job which matches the user's selection criteria and privileges
- acknowledges the Client

(3) The Client

- is freed from Xprint

The Supervisor

- gracefully ends the job's processing
- calls its Server for the job deletion

(4) The Server

- removes the job from its DB

The Supervisor

- calls the Server to schedule a new job and waits for the Server's reaction

5. Job state modification

(1) The Client

- syntactically validates the request and checks the user's authorizations
- calls the concerned Servers for a job state modification
- waits for acknowledgement

(2) The Server

- modifies the state of the non-active jobs which match the user's selection criteria and privileges
- calls each Supervisor executing one job matching the user's selection criteria and privileges
- acknowledges the Client

(3) The Client

- is freed from Xprint

The Supervisor

- gracefully terminates the job
- calls its Server to modify the job state

(4) The Server

- modifies the job state

The Supervisor

- calls its Server to schedule a job and waits for the Server's reaction

6. Server selection

The Client selects the Server that will handle a new job by defining the set of devices able to process the job. A device is able to process a job if:

- its Supervisors's characteristics match the job options.
- it is enabled for spoolin.

For this set of devices, the Client defines the subset of Servers that serve the devices. The Client selects the first Server that:

- is enabled for spoolin.
- serves the greatest number of devices able to handle the request and enabled for spoolout.
- minimizes the Client-Server-Supervisor path length.

7. Device selection

The device selection algorithm is executed by a Server each time a Client requests addition of a new job in the Job DB. The Server receives a parameter list indicating among other things the set of devices capable of processing the job. From this set of capable devices, the Server selects the first Supervisor which:

- is enabled for spoolout.
- is idle.
- minimizes the Server-Supervisor path length.

8. Job selection

The job selection consists in finding the "best" job for the Supervisor needing work. The best job:

- is waiting.
- wins the Server scheduling policy algorithm.

3.1.4. The System Control Processing

Xprint offers interfaces to control the system execution. These interfaces concern the Servers, Supervisors and Devices. The processing of requests to control these objects are described here.

1. Starting a Server

(1) The Client

- syntactically validates the request and checks the user's authorizations
- checks the Server is not yet active
- calls the Server's local XP_daemon to start it
- waits for acknowledgement

(2) The Server's XP_daemon

- creates a new process to run the Server program
- remains able to perform the Server's recovery
- acknowledges the Client

(3) The Client

- is freed from Xprint

The Server

- initiates itself, and depending on the options it is started with, accepts/rejects all add_job/schedule requests

2. Starting a Supervisor

(1) The Client

- syntactically validates the request and checks the user's authorizations
- checks whether the Supervisor is not yet active and the respective Server is already active
- calls the appropriate Server to start the Supervisor

- waits for acknowledgement

(2) The Server

- calls the Supervisor's XP_daemon to start it
- waits for acknowledgement

(3) The Supervisor's XP_daemon

- creates a new process to run the Supervisor program
- remains able to perform the Supervisor's recovery
- acknowledges the Server

(4) The Client

- is freed from Xprint

The Server

- depending on the request options, accepts/rejects the add_job/scheduling requests for/from this Supervisor
- acknowledges the Client

The Supervisor

- initiates itself

3. Starting a Device

(1) The Client

- syntactically validates the request and checks the user's authorizations
- checks that the respective Server and Supervisor are both active
- calls the appropriate Server to start the Device
- waits for acknowledgement

(2) the Server

- depending on the request options, accepts/rejects the add job/scheduling requests for the Device
- acknowledges the Client

(3) the Client

- is freed from Xprint

The Server

- calls the appropriate Supervisor to start the Device

(4) The Supervisor

- locks and initiates the Device
- calls its Server to schedule a job and waits for the Server's reply

4. Active state modification

(1) The Client

- syntactically validates the request and checks the user's authorizations
- checks that the respective objects are already active
- calls the respective Server to modify the object state
- waits for acknowledgement

(2) The Server

- if it is the target, updates the Xprint DB with its new state
- if it is not the target, passes the request to the appropriate Supervisors
- adapts its processing
- acknowledges the Client

(3) The Client

- is freed from Xprint

The Supervisor

- updates the Xprint_DB with the concerned object's new state
- locks and initiates the Device, if the Device is the target and is enabled for spoolout

5. ***Stopping a Server***

(1) The Client

- syntactically validates the request and checks the user's authorizations
- checks the Server is active
- calls on the Server to stop itself
- waits for acknowledgement

(2) the Server

- returns an error if it still finds active Supervisors
- acknowledges the Client

(3) The Client

- is freed from Xprint

The Server

- updates the Xprint DB with its new state
- terminates itself

(4) The Server's XP_daemon

- cleans up its process table

5. ***Stopping a Supervisor***

(1) The Client

- syntactically validates the request and checks the user's authorizations
- calls the relevant Server to stop the Supervisor
- waits for acknowledgement

(2) The Server

- checks the Supervisor is still active and the associated Device already inactive
- calls on the Supervisor to stop itself
- acknowledges the Client

(3) The Client

- is freed from Xprint

The Server

- cleans up its tables about the terminating Supervisor

The Supervisor

- updates the Xprint DB with its new state
- terminates itself

(4) The Server's XP_daemon

- cleans up its process table

5. ***Stopping a Device***

(1) The Client

- syntactically validates the request and checks the user's authorizations
- calls the relevant Server to stop the Device
- waits for acknowledgement

(2) The Server

- checks whether the Supervisor and the relevant Device are still active

- rejects all add_job/scheduling request for the Device
- calls on the Supervisor to stop the Device
- acknowledges the Client

(3) The Client

- is freed from Xprint

The Server

- cleans up its tables about the terminating Device

The Supervisor

- gracefully terminates the active job if one exists and following the request options,
- releases the Device
- updates the Device state in the Xprint_DB
- waits for a request to handle

3.1.5. The shutdown

The shutdown is the way to gracefully make Xprint unavailable on a host.

(1) The Client

- syntactically validates the request and checks the user's authorization
- informs the local XP_daemon that the shutdown has been requested
- notifies each active local Server that the shutdown has been requested

(2) The XP_daemon

- SP_Daemon rejects any request
- waits to be notified of its local process termination

The Server

- rejects any Client request
- calls its Supervisors to stop the Device
- calls its Supervisors to stop
- waits to be notified of all its Supervisors termination

(3) The Server

- updates its state in the Xprint DB
- terminates itself

(4) The XP_daemon

- calls the local DB_manager for termination
- terminates itself

(5) The DB_manager

- releases the resources it owned
- notifies its successor, if it is the master
- terminates itself

3.1.6. Process recovery

Each entity has a process associated with it. The abnormal termination of such processes should never occur in reliable systems. Nevertheless, because the reality is somewhat different, the Xprint system has to guarantee a minimum recovery in case of crashed entities.

Because the XP_daemon is the father of each local Xprint process, it may be informed by the system of its children's abnormal termination.

If the child is the DB_manager, the XP_daemon creates a new DB_manager.

If the child is a Server, the XP_daemon creates a new Server.

After the new Server retrieved its operational data from the Xprint DB (active, enabled or disabled objects, ...), it answers normally to the incoming requests.

In case of a Server downtime, the Client entities stop their processing and notify their caller, while the other entities retry their call until the Server processes them.

The Server Job DB is made of files. The Server is the only entity which access it. This explains why the Server Job DB should not become inconsistent during a Server downtime.

If the child is a Supervisor, the XP_daemon only calls the corresponding Server for the Supervisor recovery.

The Server sets the possible active job to the "SUSPEND" state and performs the same actions as if the Supervisor had notified its termination after a stop request.

3.2. File transfer

The following transfer strategies have been investigated:

- Transfer by Server in two steps.

  The data are transferred to the Server's host at submission time and to the Supervisor's host at scheduling time.

  The only advantage is that if the host providing the data shuts down before the job is scheduled, the data can still be given to the Supervisor.

  The major disadvantage consists of a possible double file transfer if the Client, Server and Supervisor run on different hosts. If the Server and Supervisor are not on the same host, the Supervisor remains idle for some time.

- Transfer by Server in one step.

  The data are transferred from their original location to the Supervisor's host at scheduling time. The Server initiates the transfer.

  The double transfer possibility disappears, but the Supervisor's idle time remains. The data's possible unavailability is an additional disadvantage.

- Transfer by Supervisor.

  The data are transferred at job execution time by one child of the Supervisor's main process, the frontend process. In the meantime, another child process, the backend, may initiate the printer according to the job's options.

  The data's possible unavailability remains the only disadvantage. The data are transferred only once and the Supervisor's waiting time is reduced to the Server's scheduling time.

Xprint uses the third file transfer strategy, in order to reduce network loading and Supervisor's idle time.

3.3. Inter process communications

Communication endpoints

Each entity is represented by at least one process. The former section presented algorithms requiring information exchange between those entities.

Because they are distributed across the network, the entities exchange their information through what it is called a *communication endpoint.*

A communication endpoint is either a *BSD socket* or a *XTI transport endpoint.* It is a compile option that will select the appropriate networking service (BSD sockets or X/Open XTI) depending on the available libraries and the underlying network protocols.

The transport level interface has been preferred to the RPC application level interface because, as far as we know, the RPC interface does not support the ISO protocol yet. However, Xprint is designed in such a way that the code using the network services could easily be replaced or modified, and it is foreseen to develop further Xprint versions using the RPC interface as a network service.

Through the communication endpoints, the entities exchange *Xprint packets.* An Xprint packet is a stream of bytes made of:

- an header specifying among others the request, the requestor's identity, the length of the request parameters.

- a variable length message specifying the parameters of the request.

Signals

In rare cases, processes may exchange information through signals. This is the method by which the XP_daemon is notified of its children's termination. It is also the interface used by the XP_daemon to force its children to terminate if they become immune to the shutdown request sent through the communication endpoints.

Pipes

Processes communicate through a pipe if they are children of the same main process. Thus the Scheduler receives requests from its brothers through a pipe created by the Server's main process.

The execution of a job by the Supervisor implies processing in at least two processes. The frontend process reads the user's data. The backend presents them to the device and ensures the device error recovery. The frontend and the backend are both children of the Supervisor's main process. Data filtering is achieved by inserting additional processes between the two endpoints. Two subsequent processes exchange the user's data through a pipe created by the Supervisor's main process.

3.4. Fault and orphan detection

The detection of faulty children is effected by the XP_daemon which catches all SIGCHLD signals and with the help of its process table, performs process recovery in case of abnormal termination. System availability is thus guaranteed in case of abnormal process termination.

Each Xprint entity is made at least of one main process whose responsibility is to listen to the communication endpoint. Each time a request connection is desired, the main process creates a child process which will read the request and handle it. The main process is thus able to ensure entity availability even in case of an endless loop in one of its children.

4. Project status and perspectives

The first release of the Xprint project is now available on the SINIX 5.2x, SunOS 4.0, SCO Xenix V/386 and SCO ODT platforms with TCP/IP as network protocol.

The foreseen main developments are:

- the support of the ISO protocol,
- the integration of the OSF Distributed Computing Environmement,
- perfomance enhancements for the support of large network topologies,
- and the porting of Xprint on additional UNIX and proprietary platforms.

References

[Ass88a] European Computer Manufacturers Association, *Document Print service description and print access protocol specification*, October 1988.

[Han89a] B. Handspicker, R. Hart, and M. Roman, *The Athena Palladium Print System*, February 1989.

Computer Emergency Response – An International Problem[†]

Richard D. Pethia
Kenneth R. van Wyk

Computer Emergency Response Team / Coordination Center
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213-3890
U.S.A.

ABSTRACT

Computer security incidents during the past few years have illustrated that unauthorized computer activity does not obey traditional boundaries (e.g., national, network, computer architecture). Instead, such activity frequently crosses these boundaries not just once, but several times per incident [Sto89a].

International cooperation among computer security response groups can be an effective means of dealing with computer security issues faced today by the computer user community. This paper addresses the need for such cooperation and suggests methods by which individual computer security response groups can work together internationally to cope with computer security incidents.

1. Background

The increasing use and dependence on interconnected local, regional, and wide area networks, while bringing important new capabilities, also brings new vulnerabilities. Widely publicized events such as the November 1988, Internet Worm, which affected thousands of systems on the international research network Internet, or the October 1989, WANK worm, which affected hundreds of systems on NASA's Space Physics and Analysis (SPAN) network are unusual, although dramatic, events. There are many more events such as intrusions, exploitation of vulnerabilities, and discovery of new vulnerabilities that occur with much greater frequency and require effective methods of response. Several examples are listed below.

1.1. Incidents

From August 1986 until late 1987, staff members at Lawrence Berkeley Laboratory worked with investigators to trace the paths of a computer intruder; the trail eventually lead them to a KGB-funded intruder operating out of Hannover, Germany [Sto88a]. The investigation was often hampered by a lack of cooperation among "bureaucratic organizations" [Sto88a]. On the other hand, "cooperation between system managers, communications technicians, and network operators was excellent" [Sto88a]. Still, it was only when the investigators in both countries got involved that the intruder was apprehended [Sto89a]. It is worthwhile to note that the break-ins in this case utilized the same attack methods over and over (such as repeatedly guessing common and system default username/password combinations, exploiting well known security holes which had not yet been fixed by the system administrators, etc.); through diligent, methodical application of these methods, the intruders were successful at entering dozens of computer systems [Sto89a].

In November 1988, a rogue worm program entered the Internet and caused widespread system failures [Spa88a]. The worm, written by Cornell University graduate student Robert Tappan Morris, Jr. [Mar90a], exploited lax password policies as well as two software implementation errors in specific versions of UNIX, the predominant operating system on Internet computers.

† Sponsored by the U.S. Department of Defense

More recently, three Australian computer intruders were arrested by Australian Federal police "after a two-year investigation that included cooperation with United States authorities" [Mar90b]. Again, the intruders exploited known vulnerabilities to gain unauthorized entry onto systems [Dan90a].

In October 1989, a worm program called Worms Against Nuclear Killers (WANK) infected a National Aeronautics and Space Administration (NASA) network [Joh89a]. The worm program spread to many computers in different countries by using system vulnerabilities that "should have been closed months ago" following a similar incident in December 1988 [Joh89a].

In another, albeit domestic, case, two computer intruders were arrested and charged with illegal use of computer systems at Pennsylvania State University. The intrusions took place on a computer system at the University of Chicago. University of Chicago officials contacted CERT/CC, which then contacted administrators at Penn State. Eventually, through the cooperation of the administrators and investigators, the two Penn State students were charged [Gra90a].

These cases all illustrate the need for cooperation among computer security response groups.

1.2. System Vulnerabilities

Another situation in which cooperation across multiple organizations becomes essential is in dissemination of system vulnerability alerts (and, more importantly, their solutions). As system intruders successfully gain access to systems which have weak passwords or systems where known security vulnerabilities have not been closed, they often share information on vulnerabilities in these systems with others. Likewise, as intruders discover new vulnerabilities in particular operating system or other software packages, information on the vulnerabilities is quickly communicated through various bulletin boards and other electronic forums.

As a result, many large communities of system users quickly become vulnerable. Traditional methods of dealing with vulnerability information, including closely protecting information on the existence of the vulnerability, are not effective once intruders have learned of system weaknesses. In these cases, supplying password guideline and security vulnerability information to system administrators is crucial in raising security levels and deterring attacks.

The Computer Emergency Response Team Coordination Center (CERT/CC) (see Section 2.1) frequently distributes CERT Advisories that, among other things, inform the public of vulnerabilities, fixes, and active methods of attack.

2. Emergency Response Groups

2.1. Introduction to CERT

Shortly after the Internet worm of November 1988 [Spa88a], the Defense Advanced Research Projects Agency (DARPA) started the Computer Emergency Response Team (CERT), whose Coordination Center (CERT/CC) is located at Carnegie Mellon University's Software Engineering Institute (SEI) [Sch88a]. "The CERT is a community group intended to facilitate community response to computer security events involving Internet hosts" [Den90a]. CERT consists of hundreds of highly qualified volunteers throughout the computer community, as well as the staff of the CERT/CC and of the other emergency response groups in the CERT-System (see Section 2.2 for details). The CERT/CC serves as a focal point for response to Internet computer security problems [Den90a]. Since it would be impossible for any one response group to address the needs of all constituencies,† the need for multiple CERT groups exists. (This issue, too, is covered in more detail in Section 2.2.)

CERT groups must have sufficient in-house technical expertise to handle a reasonable portion of day to day security incidents, leaving the volunteer contacts for situations which require additional expertise. However, because emergency response involves addressing more than just technical issues, CERT membership includes not only technical experts, but site managers, security officers, industry representatives, and government officials [Den90a].

One of the essential characteristics of a CERT group is being available to its constituency on a 24 hour per day basis. There must be a well publicized central point of contact which is available continuously. This should include, at a minimum, a "hotline" telephone which is constantly manned, and an electronic

† The term "constituency" is used here to define a group with some common needs.

mailbox which is monitored during business hours. The CERT/CC hotline number is (412) 268-7090, and its electronic mailbox address is cert@cert.sei.cmu.edu, on the Internet.

It is critical that a CERT group build and maintain a collection of contacts, both within the group's constituency and externally [Dal90a]. The contact information should include other CERT groups, system vendors, law enforcement, network operation centers, technical experts, site administrators, etc. Building the contact information is an on-going process in which contacts are developed and maintained over time. Each contact must be aware of its responsibilities and/or expectations in the emergency response process [Dal90a].

In addition to the contact information, a CERT group should maintain an information repository which will be drawn upon in future incidents. The information in the repository will include contact information (as detailed above), system vulnerability details, security incident reports, electronic mail archives, and other relevant information [Den90a]. Due to the nature of this information, the security on the system on which it resides must be beyond reproach. CERT/CC maintains its information database on an off-line system, which is not accessible via network connections.

As system vulnerabilities (and their fixes), break-in warning information, and other relevant information becomes available, CERT groups should issue advisories to members of their constituency [Den90a]. Past CERT/CC advisories have included vulnerability notification (along with appropriate solutions), warnings of widespread break-ins and symptoms thereof, and secure system administration suggestions. The entire collection of CERT/CC advisories are maintained on-line and are accessible to CERT/CC constituents.

2.1.1. Example CERT Incident Handling Procedures

As an ongoing process, CERT/CC has developed and is continuing to improve upon its event handling procedures. Naturally, the procedures are different for each distinct type of event (e.g., system break-in, vulnerability report, worm). This section presents an overview of some of these procedures.

When CERT/CC receives a report of a system break-in, it first works together with the affected system administrator(s) in determining how the intruder gained access to the system. This is generally in the form of offering guidance on what sort of signs the administrator should look for to determine means of access. Next, CERT/CC offers assistance in repairing the exploited hole(s), as well as other commonly known vulnerabilities. Examining systems for backdoors or trojan horses that have been planted by the intruder is an especially important activity. If the break-in came from other sites, or if the intruder broke into other systems from the current system, CERT/CC notifies other affected site administrators (from time to time, the administrator will already have contacted other affected sites; in such a case, CERT/CC requests to be kept up to date with the relevant flow of information between the sites). In some cases, other affected, or potentially affected, sites are not Internet sites. In these cases, communication across traditional "territorial" boundaries is especially important. It is important to note that, when contacting sites, CERT/CC always maintains the confidentiality of the affected sites unless the sites specify otherwise.

As system vulnerabilities are reported to CERT/CC, they are first authenticated and then reported to the affected vendor(s). CERT/CC offers guidance to the vendor community by reporting the magnitude of the threat (e.g., whether the hole is being actively exploited, whether the hole is known to a widespread audience, whether the hole can be exploited from a remote system or requires existing system access in order to be exploited). CERT/CC also offers technical input, if desired by the vendors. The vendor community will generally respond with a fix, a workaround, or a reference to same for the problem. In many cases, the CERT/CC has received advanced versions of fixes from vendors and has received the vendor's authorization to release the fix to selected members of the technical community for review and comment. This technical review process shows promise of improving the quality of corrections to vulnerabilities.

Depending upon the situation, CERT/CC then drafts an advisory for review by the vendors, the CERT-System, and/or technical affiliates. When the draft advisory is mutually accepted, it is distributed electronically to CERT/CC's constituency, the Internet research community. For this, CERT/CC operates a CERT Advisory mailing list, in addition to a Usenet newsgroup, comp.security.announce [Qua90a]. (See Appendix 1 for an example CERT/CC advisory.)

2.2. CERT-System

As mentioned in Section 2.1, no single emergency response group can be expected to address the needs of every portion of the computing world, due to the diversities and scale of all of the various computing environments [Den90a]. Individual communities each have their own distinct policies, rules, regulations, procedures, and culture. Methods effective in one community (e.g., the Internet research community) would not likely succeed in other communities that have significantly different cultures (e.g., the military community or the banking community).

In addition, implementation platforms (operating systems, networking software and protocols) vary widely. A single CERT group would not likely be successful in dealing with technical diversity, or at least could not do so economically.

The "CERT-System" model, therefore, includes multiple, cooperating individual CERT organizations. Each individual CERT group in the CERT-System focuses on a particular constituency.

Each constituency in the model can be defined by either user or technology boundaries. The user constituencies consist of groups with common networks, needs, and/or policies, while the technology constituencies are groups with common computing architectures [Den90a]. An example of a user constituency is the Internet research community, which is made up of organizations in academia, government, military, as well as commercial groups. These groups are bound together by being members of the Internet network. An example technology constituency is the IBM mainframe community, which is bound by a common computer architecture.

The CERT/CC group addresses both a user constituency (Internet research community) and a technology constituency (UNIX-based workstations and mainframes, which is the primary type of system on the Internet).

The CERT model lends itself well to network groups such as the Internet research community, as well as corporate [Fed90a], government, military, etc., groups.

In times of crisis, many CERT groups can be active with a technology coordination center analyzing problems and coordinating the search for solutions and with user constituency coordination centers gathering information and informing their constituents as appropriate.

In less troubled times, the CERTs work together to build effective communication mechanisms, share information on effective computer security tools and techniques, and conduct proactive campaigns aimed at increasing the awareness of computer security issues and improving the security of operational systems.

The CERT-System model has been widely accepted and eleven groups funded by U.S. government agencies and several private firms now participate in the system. Interest in participating has been expressed by several other organizations and steps are being taken to structure the CERT-System more formally. This structure, including a charter and by-laws that are being reviewed and approved by existing CERT-System members as this paper is being written, will provide a framework to enable wider participation.

3. Conclusions

It has been shown that the CERT concept can be an effective means of responding to computer security-related incidents [Gra90a]. In incidents prior to the existence of CERT, system administrators were frequently at a loss for outside assistance when handling security incidents [Sto88a, Sto89a]. It has also been shown that computer system security incidents do not obey network, national, or architectural boundaries [Sto88a] and that the intruders frequently exploit lax security procedures (due, perhaps, to a lack of specific knowledge on the administrators' part) [Sto88a, Dan90a, Joh89a].

Effective computer security incident response requires communication and coordination across multiple communities. While many incidents occur because software design or implementation deficiencies are exploited, resolution of the incidents requires more than a technical solution. Communication of threat and vulnerability information across computing communities is essential to resolving specific incidents and improving the security of operational systems.

A well formed CERT-System will raise security awareness and knowledge among site administrators as well as give the administrators sources of assistance in times of computer emergencies. By drawing on the experiences of individual CERT groups, the knowledge level of the CERT-System as a whole will grow, enabling all members to more effectively and efficiently deal with computer security incidents as they arise.

4. Example CERT/CC Advisory

CA-90:02

CERT Advisory
March 19, 1990
Internet Intruder Warning

There have been a number of media reports stemming from a March 19 New York Times article entitled "Computer System Intruder Plucks Passwords and Avoids Detection." The article referred to a program that attempts to get into computers around the Internet.

At this point, the Computer Emergency Response Team Coordination Center (CERT/CC) does not have hard evidence that there is such a program. What we have seen are several persistent attempts on systems using known security vulnerabilities. All of these vulnerabilities have been previously reported. Some national news agencies have referred to a "virus" on the Internet; the information we have now indicates that this is NOT true. What we have seen and can confirm is an intruder making persistent attempts to get into Internet systems.

It is possible that a program may be discovered. However, all the techniques used in these attempts have also been used, in the past, by intruders probing systems manually.

As of the morning of March 19, we know of several systems that have been broken into and several dozen more attempts made on Thursday and Friday, March 15 and 16.

Systems administrators should be aware that many systems around the Internet may have these vulnerabilities, and intruders know how to exploit them. To avoid security breaches in the future, we recommend that all system administrators check for the kinds of problems noted in this message.

The rest of this advisory describes problems with system configurations that we have seen intruders using. In particular, the intruders attempted to exploit problems in Berkeley BSD derived UNIX systems and have attacked DEC VMS systems. In the advisory below, points 1 through 12 deal with UNIX, points 13 and 14 deal with the VMS attacks.

If you have questions about a particular problem, please get in touch with your vendor.

The CERT makes copies of past advisories available via anonymous FTP (see the end of this message). Administrators may wish to review these as well.

We've had reports of intruders attempting to exploit the following areas:

1. Use TFTP (Trivial File Transfer Protocol) to steal password files.

   To test your system for this vulnerability, connect to your system using TFTP and try "get /etc/motd". If you can do this, anyone else can get your password file as well. To avoid this problem, disable tftpd.

   In conjunction with this, encourage your users to choose passwords that are difficult to guess (e.g. words that are not contained in any dictionary of words of any language; no proper nouns, including names of "famous" real or imaginary characters; no acronyms that are common to computer professionals; no simple variations of first or last names, etc.) Furthermore, inform your users not to leave any clear text username/password information in files on any system.

   If an intruder can get a password file, he/she will usually take it to another machine and run password guessing programs on it. These programs involve large dictionary searches and run quickly even on slow machines. The experience of many sites is that most systems that do not put any controls on the types of passwords used probably have at least one password that can be guessed.

2. Exploit accounts without passwords or known passwords (accounts with vendor supplied default passwords are favorites). Also uses finger to get account names and then tries simple passwords.

   Scan your password file for extra UID 0 accounts, accounts with no password, or new entries in the password file. Always change vendor supplied default passwords when you install new system software.

3. Exploit holes in sendmail.

   Make sure you are running the latest sendmail from your vendor. BSD 5.61 fixes all known holes that the intruder is using.

4. Exploit bugs in old versions of FTP; exploit mis-configured anonymous FTP.

   Make sure you are running the most recent version of FTP which is the Berkeley version 4.163 of Nov. 8 1988. Check with your vendor for information on configuration upgrades. Also check your anonymous FTP configuration. It is important to follow the instructions provided with the operating system to properly configure the files available through anonymous ftp (e.g., file permissions, ownership, group, etc.). Note especially that you should not use your system's standard password file as the password file for FTP.

5. Exploit the fingerd hole used by the Morris Internet worm.

   Make sure you're running a recent version of finger. Numerous Berkeley BSD derived versions of UNIX were vulnerable.

   Some other things to check for:

6. Check user's .rhosts files and the /etc/hosts.equiv files for systems outside your domain. Make sure all hosts in these files are authorized and that the files are not world-writable.

7. Examine all the files that are run by cron and at. We've seen intruders leave back doors in files run from cron or submitted to at. These techniques can let the intruder back on the system even after you've kicked him/her off. Also, verify that all files/programs referenced (directly or indirectly) by the cron and at jobs, and the job files themselves, are not world-writable.

8. If your machine supports uucp, check the L.cmds file to see if they've added extra commands and that it is owned by root (not by uucp!) and world-readable. Also, the L.sys file should not be world-readable or world-writable.

9. Examine the /usr/lib/aliases (mail alias) file for unauthorized entries. Some alias files include an alias named "uudecode"; if this alias exists on your system, and you are not explicitly using it, then it should be removed.

10. Look for hidden files (files that start with a period and are normally not shown by ls) with odd names and/or setuid capabilities, as these can be used to "hide" information or privileged (setuid root) programs, including /bin/sh. Names such as ".. " (dot dot space space), "...", and .xx have been used, as have ordinary looking names such as ".mail". Places to look include especially /tmp, /usr/tmp, and hidden directories (frequently within users' home directories).

11. Check the integrity of critical system programs such as su, login, and telnet. Use a known, good copy of the program, such as the original distribution media and compare it with the program you are running.

12. Older versions of systems often have security vulnerabilities that are well known to intruders. One of the best defenses against problems is to upgrade to the latest version of your vendor's system.

    VMS System Attacks:

13. The intruder exploits system default passwords that have not been changed since installation. Make sure to change all default passwords when the software is installed. The intruder also guesses simple user passwords. See point 1 above for suggestions on choosing good passwords.

14. If the intruder gets into a system, often the programs loginout.exe and show.exe are modified. Check these programs against the files found in your distribution media.

If you believe that your system has been compromised, contact CERT via telephone or email.

J. Paul Holbrook
Computer Emergency Response Team (CERT)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Internet: cert@cert.sei.cmu.edu
Telephone: 412-268-7090 24-hour hotline: CERT personnel answer
7:30a.m.-6:00p.m. EST, on call for emergencies other hours.

Past advisories and other information are available for anonymous ftp from cert.sei.cmu.edu (128.237.253.5).

References

[Dal90a] J. Dalton, "Building a Constituency – An Ongoing Process," *Proceedings, Computer Emergency Response Team Workshop* (1990).

[Dan90a] R. Danca, "Officials Confirm Latest Attempt to Invade Internet," *Federal Computer Week* **4**(12) (1990).

[Den90a] P. Denning, *Computers Under Attack,* ACM Press (1990).

[Fed90a] A. Fedeli, "Forming and Managing a Response Team," *Proceedings, Computer Emergency Response Team Workshop* (1990).

[Gra90a] J. Graf, "2 Charged With Illegal Computer Use," *Centre Daily Times* (February 17, 1990).

[Joh89a] M. Alexander, M. Johnson, "Worm Eats Holes in NASA's Decnet," *Computer World* (October 23, 1989).

[Mar90a] J. Markoff, "3 Arrests Show Global Threat to Computers," *New York Times* (April 4, 1990).

[Mar90b] J. Markoff, "Student Says His Error Crippled Computers," *New York Times* (January 19, 1990).

[Qua90a] J. Quarterman, *The Matrix: Computer Networks and Conferencing Systems Worldwide,* Digital Press (1990).

[Sch88a] W. Scherlis, "DARPA Establishes Computer Emergency Response Team," *DARPA Press Release* (December 6, 1988).

[Spa88a] E. Spafford, "The Internet Worm Program: An Analysis," Technical Report, Purdue University Department of Computer Sciences (1988).

[Sto88a] C. Stoll, "Stalking the Wily Hacker," *Communications of the ACM* **31**(5) (1988).

[Sto89a] C. Stoll, *The Cuckoo's Egg – Tracking a Spy Through the Maze of Computer Espionage,* Doubleday (1989).

Distributed Computing for the Technical Workplace

Daniel E. Geer, Jr.
George A. Champine

External Research Program
Digital Equipment Corporation
{geer,champine}@crl.dec.com

ABSTRACT

A quantum change in the way in which technical computing is *best* done is well under way. Hardware progress (such as cheap cpu cycles), software progress (such as network transparent window systems), and political progress (especially the appreciation of open systems) represent substantive differences between the world of even two years ago and the world now possible. Project Athena has spent the last six years constructing one of the best examples of distributed computing environments. At M.I.T., ten thousand users are sharing a pool of thirteen hundred workstations and one hundred various servers; all figures are growing in real time. This talk will take our experience in creating a network services model of computation subject to the real life constraints of the academic setting. Particular emphasis will be placed on wide-area systems management and the maintenance of a coherent computing environment on a large-scale, heterogeneous computing infrastructure.

We will further claim that the model Athena has settled on is the winning model – in fact that it has already won. The organizations that can grasp the idea that a particular computational activity is really a boat in a common sea of network services will survive; those that rely on single-source, stand-alone hardware will not.

1. Introduction

Cheap cycles on every desk, device-independent, network-transparent window systems, nationwide file systems, rampant heterogeneity in the computing plant, coordination costs scaling faster than linearly with node count, *etc.*, *etc.*: what are these all part of? The answer is where computing is, and is going. There is so much technical progress being made that managing the complexity of opportunity so as to turn it into a richness of facility is a challenge that needs new thinking. Any organization that already has 100 workstation-class machines and a bit of network knows this. Any one with 1000 is bleeding for it.

This article will discuss the Model T of the age, the computing environment called *Athena*. Starting with the model of computation, the implementation of it at MIT, experience with it, and what the lessons are both for companies who need it and those who want to profit from its experience.

2. The Current Model Is Broken

The model of computation called time-share is the woolly mammoth of today. Arbitrarily big and hairy, not safe to approach without teams of trained people, and completely managed by a single well constructed message delivered at arm's length, it has been the mainstay of those primitive societies willing and able to come to it and to go where it goes. It got us out of the trees and into caves. But now that the ice is receding, it is time to domesticate some smaller critters, ones we can keep in great flocks and manage with more affordable labor. The shepherd, the sheep, and the dogs will get us out of caves and skins, into villages and cloth.

Seriously, the time share model was a great allocator of scarcity – making the absolute most out of two expensive commodities, person time and computing horsepower. It also gave a coherence that only central direction can readily provide. But the conditions that made that paramount have changed; most of the MIPS in most organizations are now idling on desks when they are not being spent on user interface. Most of the buying decisions are noncentralized, or kept central only by force of will. Nearly everyone wants more horsepower *where they are* and devil take the hindmost. Till now, nobody could manage – the proportionality constants of labor and infrastructure costs relative to node count are far too high. The real issue, the one that counts, is how the organization can get the availability and facility of its investment to grow upward by an order of magnitude or two and yet maintain (or even improve) coherence at low to no additional cost. Let's look at what we can learn from the one place that has already demonstrated scaling up without topping out, Project Athena at the Massachusetts Institute of Technology.

3. It Can Be Fixed

Project Athena, an eight-year joint effort between MIT, Digital Equipment Corp, IBM, and others, was created to change the way in which education was delivered. To do this, a computing plant with these design goals was required:

Scale to 10,000 heterogeneous hosts

Labor cost 1-2 orders of magnitude below traditional

Ubiquitous, location independent computing

Coherent, both for programmer and user

Reliable, available, and predictable

Sustainable after the experimental phase

In this, it has succeeded. At MIT today, 10,000 users share a pool of 1500 various workstations and server hosts spread over 30 networks covering a campus two miles wide and thirty departments deep. Four thousand persons use Athena on an average day. Fifty gigabytes of spinning store and 150 printers support them. All these figures are growing in real time; all this environment is run by an Operations staff of six FTE's, five not counting the manager. We are talking workstation to systems manager ratios an order of magnitude better than the industry average of 25. We are talking a model of computing that can double the computing supply side at (non-hardware) incremental costs of less than 10%. We are talking about allocating labor exclusively to exception handling and planning/anticipation, turning over the routine almost entirely to centrally automated means. We are talking an enterprise-wide computing model that has had no downtime in three years (outside of electric power failures) and is immune (within logical limits) to single point failures of the network.

The advantage to a company that is already over some threshold number of workstations is to pick a new point in the multidimensional space of performance, management cost, and availability. A 10 workstation environment can retain all the control of a timeshare world with the flexibility of a distributed environment. A 100 workstation environment can save labor, introduce location-independence, and achieve sufficient service security to direct-connect to the Internet. A 1000 workstation environment can support heterogeneity via protocol coherence, make installation and recovery operations accessible to peripheral staff, lower complexity cost of coordinating central administrative operations with field operations. A 10,000 workstation environment will be able to access network-wide monitoring and event notification so as to make (sufficiently) self-healing environments for 24x7 operations, and support global consistency while allowing local administrative modification.

4. How to Fix It

4.1. Practical Constraints

To really scale up, you first need to recognize and obey the seven **NO**'s

1. No shoe leather – it is not practical to visit every node on any regular basis, and never in synchrony with any other set of events
2. No broadcast – denying cross-subnet broadcast is to networks what firewalls are to factory buildings
3. No interdependencies – services should not depend on each other for fundamental operation; it might be nice, but there must be a fallback
4. No synchronous administrative handshakes – important managerial steps must be unconstrained with respect to the order in which they are done lest bottlenecks and gratuitous error conditions be created
5. No local state – local state is a maintenance problem, no matter who creates it; define away local state, and you define away management of the client leaf nodes in the computing tree
6. No costs linear with hosts or users – economies of scale must come into play, else it will not be possible to serve the entire community of interest

7. No permanent development staff – a permanent development staff means you have missed the point somewhere; ubiquitous computing demands an end to hand-tailoring

This set of constraints leads naturally to the:

4.2. Model of Computation

Traditional time-share assumes all resources are local, that a file's pathname and its logical name are identical, that security is uniquely that of the containment barrier of a user's password at login, that service location is local and time-invariant, that modifications in resources such as computational capacity are made unilaterally and unitarily, that system management is organizationally and administratively centralized, *etc*. This is not distributed computing; this is not conducive to location independent cooperative work; this is not Athena.

Athena is a set of system software, applications, and, most importantly, a style of configuration and a practice of operation that permits scale up to large numbers of hosts, users, services, brandnames, and more. We will cover various aspects of this below, but the point is exactly one – converting a time-share model to a distributed model means modifying practices and the tools that support them in such a way that scale of operation is not constrained in any way thereby. The Athena model can already be applied to most brand names of equipment. It already is piecemeal in the product lines of many companies, including the specifications of such standard setters as the Open Software Foundation and UNIX International.

With a distributed computing model, workstations become independent computational entities floating, we argue, in a sea of services. To accomplish that end, Athena uses as its prototype a "public workstation", one which is serially reusable and which it is possible to make widely available simply by cloning. There are, of course, many private modifications that are variations on this theme, but the basic idea is that of a model with options. The prototypical model is that which is most supportable in large numbers, and so forms the backbone of the service provided to Athena's customers. Variations on this model support privately held workstations and, perhaps more surprisingly, are the basis for server hosts as well. In short, it is possible to have a basic idea of what an Athena computational engine is, and to make all other entities differences to it. In so doing, it becomes possible to support large numbers of workstations in public areas with a cookie-cutter approach, and to provide other environments be they server or client by reference.

4.3. System Libraries

Timesharing systems typically have large libraries of software available to their users. Storing a copy of that system library on each workstation would be prohibitively expensive if not technically infeasible. This situation is typically ameliorated by the use of dataless workstations and network file services. Even so, it is natural for a large environment to need several, geographically distinct copies of the software pool. In such a situation, configuration control, software support, and software maintenance are all made difficult if there is a requirement for multiple versions of the system library, whether to support software revision levels or heterogeneous workstation types.

In an Athena environment, there is one master copy of the system library, and a service management system that controls its location and availability throughout the network. This master copy is replicated to the extent necessary to get adequate network performance, such as by providing one complete copy per LAN. Partial copying is supported, and workstation consumers initiate local updates asynchronously. The fileservice is non-monolithic; a given filesystem can be one of several protocol types, mixed and matched as usage patterns indicate (a fact that is transparent to the user). The Athena solution, then, is (effectively) dataless nodes, fileservice replication as seems useful, naming services to provide location independence, and automatic, client initiated software updates.

4.4. Name Resolution

In older time sharing systems, object names are translated into addresses through the use of static configuration files. As UNIX examples; the file /etc/hosts maps machine names to network addresses, the file /etc/passwd provides the pathname to a user's home directory, and the file /etc/printcap contains printer capability information including service name to host name translation. At some size or effective dynamism, the approach of using static files on each system is impossible, and will be impractical well before that. Making indirect data updates accurate, consistent, and timely is hard, but necessary.

The Athena approach is to provide a replicable, hierarchical nameserver under central control to translate logical names to service names at the time of service connection. A small number of replicated copies of the nameserver data are generally useful to meet performance and/or availability constraints. Hierarchical delegation is a feature of the underlying technology, namely the Berkeley Internet Name Domain (BIND) system. The contents of the nameservice, that is to say the data it can return, are controlled by the Moira Service Management System (see below).

4.5. Authentication

Traditional operating systems are not compatible with open networks in that they often make unwarranted, indefensible assumptions about network security. In such manner, traditional UNIX stores encrypted passwords for every user on every machine. It is **very** difficult to maintain accurate and consistent password and group access files on 50 workstations, to say nothing of a larger number. But, more importantly, access to services on hosts other than the one to which the user is directly connected will require one of two highly insecure alternatives. As alternative one, the various remote services can simply trust that all potential clients are who they themselves claim to be. Alternative two is to trust no one, but require a password be provided – ensuring that that password will have to be typed through the suspect network in the clear. Neither solution is any solution at all.

The Athena approach is to use a trusted third-party, private-key, key-distribution center as an authentication service. A single, isolated, defensible "trusted third-party" is a host on the network which provides tokens of proof of identity to users as required. It is rather like the Prince/ss of old whose signet ring in sealing wax was not forgeable. The "private-key" is a description of the style of encryption used to hide the secrets on which proof of identity is based. "Key distribution center" indicates that (encryption) keys are being distributed in support of proofs of identity. All in all, the transaction is one of asking the Prince/ss for a letter of introduction to the remote service, getting that letter back wrapped and stamped, and shipping it off to the service you wish an introduction to, and trusting that both you and the remote service recognize the same Prince/ss. Such a method is as good as is presently available anywhere, and provably achieves secure network authentication no less strong than the underlying encryption technology.

Note that this is an authentication (who you are) system; authorization (what you can do, given who you are) is a matter for each service provider to decide as authorization is unique to the access policy of that service provider. Yes, all services provided by Athena are managed centrally and under authorization policy control, ipso facto. However, in future other service providers besides Athena proper will be players, and it is the right modularization to vest identity control in the network service infrastructure so that the individual service provider can exercise access control appropriate to its task.

4.6. Spooling

(Spooling here applies to both print spooling and outgoing mail spooling. The latter is discussed later; here we discuss printing.) UNIX assumes that printers are local, both physically and informationally (capability information). Two major drawbacks of this approach are local queueing and local static network print addresses. Normal operation puts the print queue on the local machine where a line printer daemon either prints it or requeues it to a remote machine as specified in /etc/printcap. Such local spooling is incompatible with commodity oriented workstation computing where the next user of the host may well not respect files in local spool queues. Lpr gives no indication as to whether the printer is actually accepting jobs, thus there is no guarantee that the job will ever be printed. The use of /etc/printcap as a static configuration file is also a problem. If a printer is added, deleted, or moved in the system, every /etc/printcap file must be changed, obviously a situation incompatible with scale.

In Athena, print spooling is done at the printer, with appropriate error messages, and configuration files are handled by the name server. In this way, the printservice can be located independently of the print clients, reconfigured dynamically as required, and the successful completion of a print spooling command will indicate the acceptance of the print request by the printing system, per se.

4.7. Mail

In standard UNIX, mail is addressed point-to-point, typically "username@node". That the user can get mail at only one location has obvious drawbacks in reliability, security, reconfigurability, and location independence. In addition, it ensures that mail to mailing lists is unlikely to complete as to do so would require all recipients be able and willing at all times to accept mail. Finally, traditional UNIX expects that all hosts have mailer configuration information local, and that problem is one that is totally unscalable (frequency of update rises linearly with the number of users, but the net-wide volume of information to be managed rises with the square of the number of users).

In Athena, mail is sent to "username@athena.mit.edu", a mail hub analogous to a central mail sorting facility of the USPS. On that hub, mail is rerouted to any one of a number of "Post Offices", mail spoolers running a version of the Rand MH system modified to require Kerberos mediated authentication as access control on the ultimate delivery of mail the recipient. Recipients can obtain their mail at any workstation in the system by locating the relevant Post Office (via the nameservice), sending a(n authenticated) request for spool file final delivery over the net, and entering the mail reading client of the user's choice.

4.8. Notification

A traditional operating environment had no direct method of determining the instantaneous community of interest in a particular facility. In other words, if the operations staff has to disable the computer, they have to broadcast a warning and hope everyone is awake, present, and interested. If a user wants to directly reach another, they need prior information as to where to look.

Look at a short-list description of an electronic mail message;

Arbitrary length
ASCII text
Unauthenticated
Deliver by best method when available
Known recipients

Take the converse of each of these:

Single packet
Not necessarily ASCII text
Authenticated
Deliver now or discard
Unknown recipients

and you have our concept of a "notice" and, hence, a notification system. Such a service is easier to demonstrate than to describe, but in simple operation, it is a matter of making read-me-now messages available to whomever wants to read them from whomever wants to send them. The catch is rather like the question of whether an unwitnessed treefall makes a sound – messages carry type and class information that may or may not select a recipient from among the active subscriptions present at the time of message transmission. Messages can be from a person to a person (Joe gets "Authentic message from Jimi: I hear you shot your old lady"), from a service entity to a service entity (syslog datagrams going to a sorting and filtering process for remulticast in summary analysis), from a person to a service (consulting service gets "Message from luser: What is a compiler?"), or from a service to a person ("The current weather for Cambridge is ...").

4.9. On-line Consulting

UNIX support has traditionally been little more than on-line manual pages. Many of these were not intended for and are not adequate except for experts. Further, self-help is not always a workable solution, particularly where the user population has high turnover. To truly support computing independent of time and place, it is necessary to provide consulting services over the wire.

Athena supports an On-Line Consultant (OLC). The OLC system provides a rendezvous mechanism (to connect the person with a question with a consultant), a user interface (usually a two-panel typing window like *talk* from Berkeley UNIX or *phone* from VMS), and management controls (such as logging, promotion, and forwarding through an OLC server). Consultants can answer frequent questions by pulling in answers from a stock-answer directory. This directory, equipped with a browsing tool, is a useful adjunct for the user wanting to self-help but who finds the man pages insufficient.

The average time to get an answer to a question is about four minutes, but if answers must be delayed they are returned through electronic mail. The volume of questions is growing by 15% per year, faster than other use measures, indicating the a growing reliance on the OLC system for system navigation and use.

4.10. Other network services

We won't discuss the rest of them here, but approximately 30 services exist. The X Window System is the best known, and by now nearly everyone sees the benefit of a network-transparent, vendor-neutral, device-independent window system. Time-synchronization services, conferencing systems, message-of-the-day propagators, privatized versions of consulting and administrative systems, and on and on. A point of design is worth mentioning – in an Athena environment, services and their control are arranged on the assumption that in future all nodes in the network will be both consumers and producers of services. Whether, as in the academic model, it is a professor constructing, testing, advertising, authenticating, timing, and scoring an electronically administered examination, or something similar to the industry the reader knows, the point is that what is described here as a centralized management model is, more precisely, centralizable. The hooks are there for whatever administrative structure exists; what is present today is the one that fit the inventing institution, MIT.

4.11. Macro-configuration

Traditional UNIX is completely silent on macro-configuration of computing services. Redundancy, fail-soft and fall-back arrangements, asynchronous "flag-day free" updates to field operations, self-sequencing cold-start procedures, and computing as a commodity are not part of a traditional UNIX repertoire.

At Athena, the system is available on a continuous basis. Because of the large amount of equipment, hardware failures happen routinely; but except for logically single point failures (such as the workstation in front of which an individual user is sitting, such as the private home directory of a particular user or workgroup), all services are provided redundantly. Access to services is indirected through the nameservice, making planned service outages and many unplanned ones totally transparent. Important services are redundant and geographically dispersed so that no single point of failure in the entire extent of the network results in a service degradation for the typical user.

Workstations, being dataless, are easy to make whole in the event of failure – if a workstation is misbehaving, reload its software. If it still misbehaves, call in a hardware trouble report. Such a triage operation requires only treatment skills, and is suitable for personnel available in far greater numbers than those with functionally complete diagnostic skills.

Software installation, repair, and update are accomplished by the same mechanism – that of making the target look like a reference source. Install media are subnet independent, require no prior administrative handshake (such as Internet address assignment) to use, and a workstation can be taken from crated to on the air in order ten minutes. That figure can be reduced to order 1 minute should it become attractive to provide a full reload as part of a login/startup sequence.

A distributed environment is subject to partial failures and, occasionally, to complete failures (such as due to power failure). Athena has configured its hosts and service to restart in a fashion consistent with completely hands-off cold-start. Cold-start to full operation is rate limited only by file-system integrity check time, itself a function of disk acreage and cpu horsepower. Typical Athena timings are order one hour from complete power failure to complete normal operation.

Labor at Athena is spent exclusively on exception handling, and on mechanical operations (such as tape hanging). This is no accident, and new user registration is illustrative of the style by which such a claim can be made. New users are identified to the Athena plant by a regular tape feed from the University Registrar's office. That tape provides two items of information, a name and an MIT ID number. The former serves as an identifier (the prospective user types his/her own name to a "register for a new account" client available as a button click on the faceplate of any unused workstation) and the latter serves as a temporary authenticator (for the duration of the registration session). After answering the obligatory few questions (such as picking a username, providing an office telephone number, *etc.*), the user will be thanked and asked to return on the morrow. Overnight, the Moira system will do everything required to instantiate a new user account, including, but not limited to, assigning a UNIX UID to the (user-selected) login name, finding a fileserver with low utilization and making a homedirectory for that user there (including modifying ownership and quota control), announcing these facts to/through the nameservice, assigning a mail delivery post-office (an activity similar to a homedirectory construction sequence), and so forth. Several hundred students a day can claim accounts, and a 2 MIP host will be loafing.

4.12. Service Management

Service management is typically accomplished by a (team of) wizard system manager(s) with a fanout to hosts of 25-50 hosts per. Such a system is not scalable to real ubiquity (without which, there is no location independence), and perpetually suffers from version and managerial discontinuities. In addition, no standards exist in the remote management area, so it is every application for itself. Until that situation changes, that is to say until standard management interfaces are defined and come to exist in actual vendor products, it will be the management system that is formfit to the service, not the other way around.

In order to improve the ease and accuracy of building the tables for the nameserver (and other system users of this information), and to generally provide the tools to bring arbitrarily constructed network services under central control, Athena provides the Moira Service Management System. Organized as a highly access-controlled, omniscient database and an omnipotent, omnipresent data propagator, Moira ensures two things: the integrity of the database and the (loose) consistency of the service fleet with the database contents. Constrained only by reliability and integrity constraints, Moira provides no service directly – but, rather, it makes the (eventual) conformance of the service environment to its idealized (database resident) model a sure thing. With such a system, Athena enjoys a system manager(s) fanout of 300+ hosts per. Its simplicity, that of providing an interface to data entry and editing and a service guarantee that says data will be reflected in configuration files in the field, contributes to the serviceability of an Athena environment more than all the features of that environment discussed before.

5. Conclusions / Wrap Up

Athena is successful, and prototypical. We have here described only the high points of its facilities, the major contributors to its cost-benefit ratio. Fully half the valueful content of that environment is in the area of configuration and practices, and the other half is original software in service to that new configuration suite. All of it is on the standards track, either directly (in the cases of *X* and *Kerberos*) or indirectly (note the similarities between the OSF DCE RFT and the deliberations of the Distributed Computing Working Group of UNIX International). It embraces the open systems ideal fully, and provides a sense of basic services and value added that makes any individual's analysis of cost-benefit dominated by the benefit side of the equation. In summary, the Athena environment permits a flexibility and availability that only a distributed computing model can provide at the quality, reliability, and security standards of unitary timeshare hosts. Think of Athena as a Model T – amazing for its time, precursor to a new age, and accessible to almost anyone. The best is yet to come.

An X11–based Multilevel Window System Architecture

Mark E. Carson
Janet A. Cugini

IBM Secure Workstations Department
Building 182/3F43
800 N. Frederick Avenue
Gaithersburg, MD 20879
uunet!pyrdc!ibmsid!{carson,cugini}

ABSTRACT

We have designed and are implementing a prototype multiple-security-level window system based on AIX and the X Window System [Sch86a]. This system supports discretionary and mandatory access control and information labeling on windows, properties, events and other objects. Our goal is a system which will meet a variety of user and government security requirements, in particular the U.S. Compartmented Mode Workstation Requirements [Woo87a], while maintaining upward compatibility with the current X protocol [Sch90a] and the Inter-Client Communications Conventions Manual [Ros89a]. Our approach is to implement only the basic security policy in the X server, distributing complicated or controversial aspects to special privileged clients. One result is that the server needs only the most minimal operating system security support, making feasible its use on machines such as X terminals. In this paper, we describe the server security policy and discuss the role played by the display, window, selection, and audit managers.

1. Introduction

Window systems present both a problem and an opportunity from the standpoint of security. They are a problem in that they amount to a complex way to share the computer's most visible resources (its display hardware, keyboard and mouse), with a great potential for covert abuse. But they are also an opportunity because with care they can provide sufficient separation to give a user multiple simultaneous access rights (e.g. work at multiple security levels in a multilevel environment). Especially for the more onerous mechanisms of mandatory access control, such abilities are vital in encouraging users to use rather than avoid the mechanisms the system provides.

Previous implementations of multilevel secure window systems [Cum87a, Car89a, McI88a, Smi89a] have been kernel-based (or the equivalent for [McI88a] and [Smi89a]) with relatively limited supported functionality. This greatly simplifies the security issues, but is no longer a realistic model. In this project, we are working with a modern client-server window system, the X Window System version 11, release 4, running principally on AIX version 3.1 on RISC System/6000 hardware. Our goal is to produce a generally useful system which can be configured for a variety of specific requirements, in particular the U.S. Defense Intelligence Agency's requirements for a Compartmented Mode Workstation (CMW).

Systems nowadays tend to be used in complex, heterogeneous environments. For example, our own local network has four different machine types running five different operating systems. X, however, is basically common to all. Hence, to increase the portability of our security design, we have striven to make the window system security functionality as self-contained as possible; new security mechanisms are added following existing X models. (For example, our new security-relevant clients register with the X server by expressing interest in certain security events; the X server then notifies them when they need to take action by sending them these events. Security changes to the X server itself are structured as X extensions [Fis87a].)

Disclaimer: The work described in this paper is part of a research project. No IBM product commitment is made or implied.

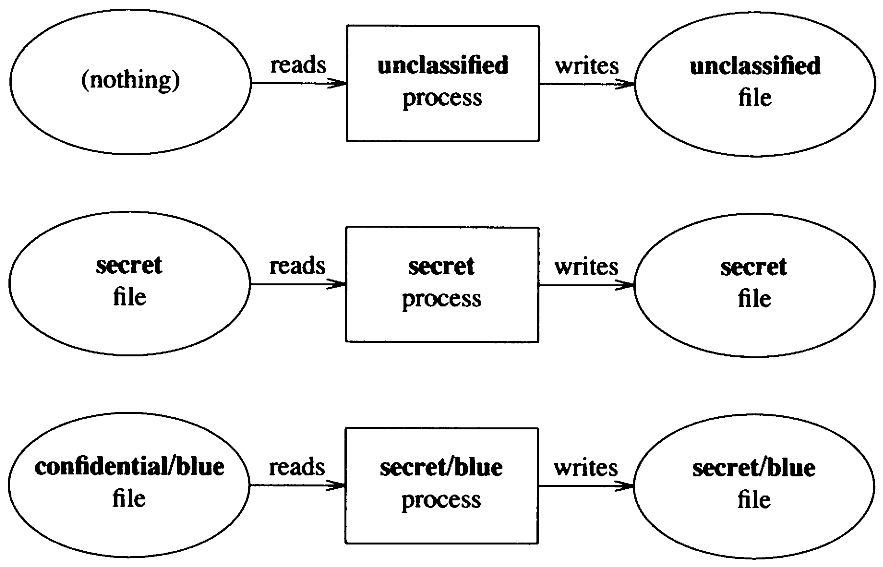


Figure 1: *The floating process*

2. Security mechanisms

For this project, we are mostly concerned with three basic security mechanisms: discretionary access control (or DAC), mandatory access control (or MAC) and information labeling. All these mechanisms control or follow the access of "subjects" to "objects." Subjects are the active entities in a system; in UNIX these are processes. Objects are passive entities; in UNIX these include files, devices, pipes, sockets, and so on. Here are brief descriptions of these three mechanisms:

Discretionary access control is an identity-based, user-controlled (hence "discretionary") means of restricting the access of subjects to objects. The UNIX system of user and group ID's on processes, and ownership and permission bits on files, is a discretionary access mechanism. The current X protocol supports a host access list for controlling connection to the server, which is a simple discretionary access mechanism.

Mandatory access control is a security label-based, administrator-assigned (hence "mandatory") means of access control. MAC is derived from government classification practices; these are based on the concepts of individual clearances and document classification labels, where a person can only read a document he or she is cleared for (the document label must be within the individual's clearance). The operating system analogue is that a process can only read objects whose security labels are less than or equal to its own ("no read up"). To prevent unauthorised declassification of data by Trojan horse programs or the like, there is an additional rule that a process can only write objects whose security labels are greater than or equal to its own ("no write down"). (In practice, this rule usually reduces to "write equal" because of the difficulties in properly handling "write ups.")

Information labeling is somewhat novel in that it is security label-based but user-controlled. Moreover, it is not intended to restrict access to data, but rather to trace its flow. As a process reads objects, its information label "floats up" to the maximum (actually least upper bound) of the information labels of the objects it has read. Then, when it writes to objects, their labels "float up" to the maximum of their old label and the process label, under the assumption they could be receiving sensitive information from the process. (See Figure 1.)

3. Basic approach in X

In the X environment, the subjects we are concerned with are the clients of the X server, while the objects are the various entities maintained by the server, such as windows, properties and selections. We decided to implement the above security mechanisms in the X server in the same way they are implemented in the (prototype) AIX operating system: X clients have subject security attributes (ID's, security labels, privileges), and X objects have object security attributes (ownership, permission bits, MAC and information security labels). The X server compares the client security attributes with the object security attributes in deciding whether to allow a client request (or whether "floating" is required, in the case of information labeling). (For compatibility with the existing X protocol, we generally deny access by pretending the object in question doesn't exist.)

Since the X server itself embodies its objects, it clearly is responsible for enforcing the basic security policy. Beyond the basic security policy, however, there are other more complex aspects of security, especially those requiring direct user interaction, which are more reasonably performed elsewhere. Our design employs four "managers" which take care of these responsibilities: the display manager (a modified *xdm*), which handles login and logout; the window manager (a modified *mwm*) which has extensions for displaying window security labels and invoking the trusted shell; the selection manager (*xsm*) which can intervene in any selection transfer activity; and the audit manager (*xam*) which takes audit data from the server to post to the system audit trail. These clients are largely independent of each other, so environments without need of the functions provided by one or the other need not run them.

3.1. Subject attributes

The server derives access-related client security attributes (user and group ID's, privileges and MAC security labels) from the corresponding attributes of the process embodying the client. The means of doing so is to a certain extent inherently operating system–, network– and level-of-trust–specific. To minimise the amount of such dependence, we make the fundamental observation:

> Client security attributes (aside from information labels) change only rarely, and then only at their direct request.

Hence, it is more sensible to transmit or change this information by some special purpose (out-of-band or higher-level) method, rather than embed it in the lower-level communications stream. (This conclusion is directly opposite that in [Pic90a].) We establish the initial client characteristics by an out-of-band inquiry to an *authorisation service*; subsequent changes to these characteristics are only made by direct request of (appropriately privileged) clients, through a new X protocol call, *XSetClientSecurityAttributes*. The interface to the authorisation service is uniform for all communication domains, hiding the details of how it is implemented.

The simplest case of the authorisation service is for UNIX domain socket connections; here it is implemented simply with a new *ioctl* option which gives the request information about one's peer. (The "authorisation service" here is simply part of the (local) kernel.)

For the TCP/IP domain in the CMW target environment, we can be assured that all machines are at the same level of trust, are uniformly administered (at least in the near term) and that the communications media are secure. In this case, it is relatively straightforward to get and trust authorisation information: all machines run an *authorisation server* (an extension of the RFC 931 [Joh85a] *authentication server*) which communicates on a privileged TCP port. The X server gets client information by talking to the authorisation server on the client's machine, which in turn receives this information from its local kernel (which stored the information at the time of the client connection request).

For the TCP/IP domain in a more general environment – where machines may be at different levels of trust, under control of different administrators, and where the network need not be physically secure – the question is much more difficult. Our (tentative) design is based on Kerberos third party authentication: one machine, which is "guaranteed" secure, runs the Kerberos authentication server and also has information on the relative trustworthiness of all machines. The X server again talks to the authorisation server on the client's machine (here with its identity guaranteed by Kerberos authentication rather than by attachment to a privileged port) to get the authorisation server's opinion of the client's attributes, but it tempers this information by the Kerberos server's opinion of the client machine (and possibly by its administrator's opinion of the client as well). Thus for example, the X server may disallow any privileges claimed by a client running on an insecure machine. This is depicted schematically in Figure 2.

Clients on machines which do not support authorisation servers can still be handled; they are treated as user `nobody`, with the lowest possible (or other appropriate) MAC label. (The server has a user-settable ACL which, among other things, can dictate whether connections from `nobody` are allowed.)

Subject information labels, unlike the other attributes, can conceivably change frequently, without direct client intervention or awareness. Hence in this case we do rely on the underlying communications means to transmit this information. (We use an *ad hoc* method for UNIX domain connections, the only type we are currently treating. For TCP/IP connections, we are contemplating making use of the extended IP security option [Joh88a], although such usage is not really consistent with its intended use for packet-level control through secure gateways.) New *fcntl*-settable flags allow changes in socket information labels to

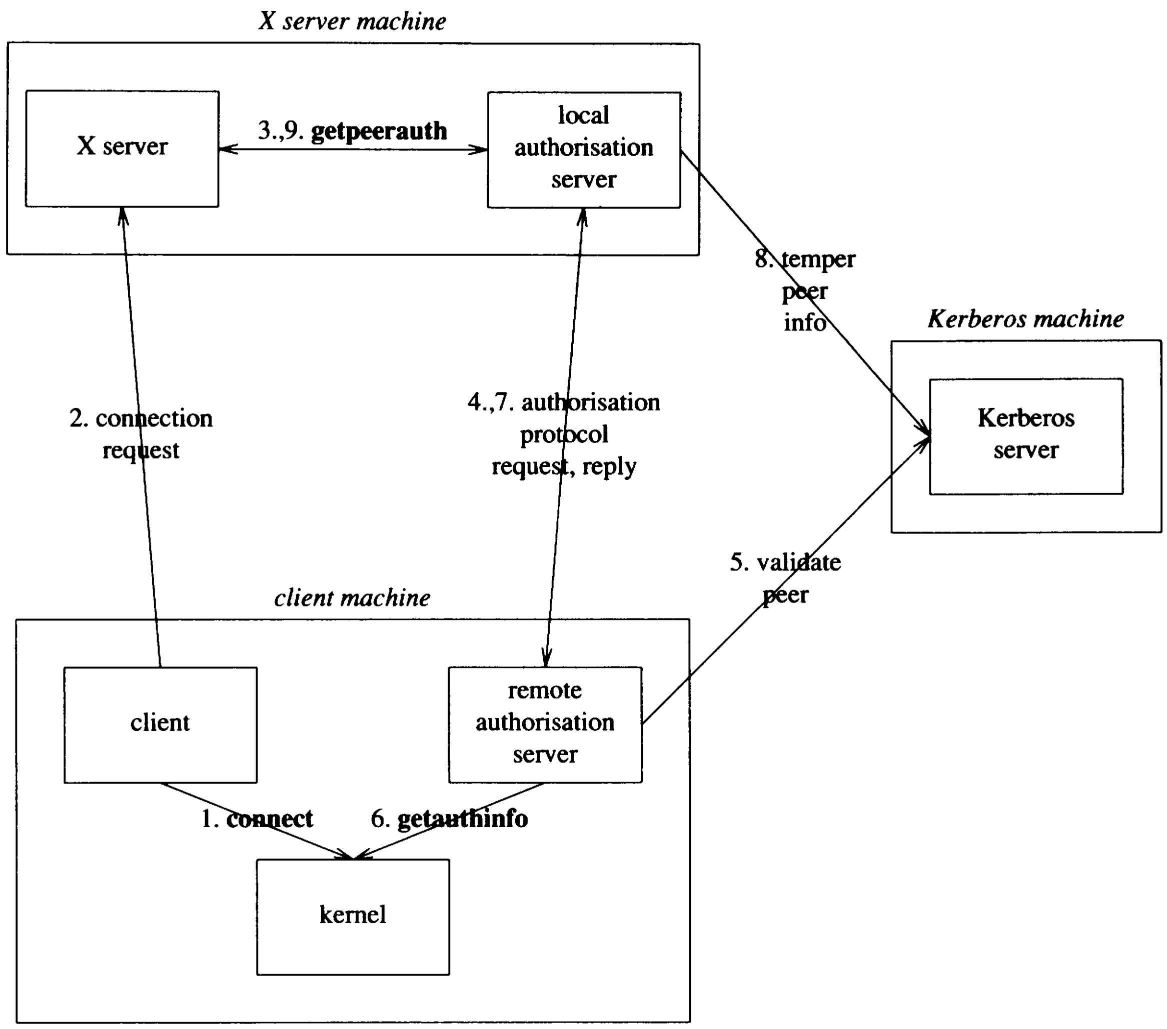


Figure 2: *Kerberos-mediated authorisation process*

be considered an "exceptional" condition reportable by *select*, or cause sending a new signal (**SIGFLOAT**) on such changes. This allows the X server (and interested clients) to be informed of such changes in a timely, convenient fashion.

3.2. Object attributes

Window system objects generally have the normal security attributes (ownership, ACL/permission bits, MAC level, information label). These are generally derived by the server from the corresponding subject attributes in the normal way (e.g. owner = effective ID of creator). Updates also happen in the normal way (e.g. information labels float on writes, and cause float on connection (and ultimately client) on read). New interfaces (*XGetResourceSecurityAttributes* and *XSetResourceSecurityAttributes*) allow getting and (for appropriately privileged clients) changing these attributes. Changes in information labels, which are the likeliest possibility, are reportable through a new X event, **LabelChange**.

Some X objects may not need all the normal access-relevant security attributes, or may need other special treatment. For example, X has the concept of `Atom`s, which are shorthand means of referring to strings. An `Atom` is created when a string is "interned" with *XInternAtom*. Its corresponding string may be read with *XGetAtomName*. There is no compelling need for any discretionary access control on atom-to-string translations, but to avoid their use as a covert means of information transfer, we do enforce mandatory access controls. To avoid unnecessary restrictions, our policy is that if an `Atom` has ever been interned at a particular MAC level, then its corresponding string can be read at that MAC level or above. Hence we associate with each `Atom` a "MAC access list" of all the MAC levels at which it has been interned. The call *XGetAtomName* will succeed if and only if the caller's MAC level is the same or greater than one of the entries in the MAC access list for that `Atom`.

It is a matter of judgement which sorts of reads and writes should cause information label floating. On the one hand, clearly writing with *XDrawText* or reading with *XGetWindowProperty* should cause floating. On the other hand, calls such as *XGetWindowAttributes* or *XRaiseWindow* should not, since window attributes such as position and stacking order are not *per se* sensitive. The question really is the intent behind the read or write: drawing text or reading properties is clearly intended to move "real," potentially sensitive information, while sizing or restacking windows clearly is not. For a discussion of this issue in a previous CMW, see [Car87a].

3.3. Alternatives

While the various security policies are structured as extensions to the X server, when employed the code enforcing them is linked to the X server itself. Conceivably this could instead be part of a special client, which the X server would call on every access decision. However, this would still require basically the same number of modifications to the server (to make the call-outs, rather than to make the decision itself), while dramatically increasing the overhead, so it is not a reasonable choice. This subject is discussed in more detail in [Pic90a].

It can be argued whether such a large set of security attributes and a security policy as complicated as the operating system's are necessary for the X server. For example, the LINX proposal [Ros89b] essentially associates only a single security attribute (an "authenticator") with X subjects and objects. Aside from "authenticator 0," subjects can only access objects with the same authenticator. It has been argued that this suffices for the security needs in X, and provides better performance than a more complicated policy.

We felt the larger set of attributes allowed a more flexible policy, and felt having the same policy as that provided by the operating system made it easier for users and programmers to understand and apply.

Although it might seem that comparing two authenticators is inherently faster than comparing a long list of subject attributes with the corresponding object ones, with the proper implementation this need not be so in most cases: the vast majority of the time, X subjects access only objects they have created, and all these objects will have the same security attributes (except possibly information labels, which are not access-relevant). Hence, we can optimise for this case by storing all the access-relevant object attributes in shared "attribute blocks." The blocks are maintained via "copy-on-write": if an object's or subject's attributes change, a new block is allocated, and the changed information is written to it. The X client data structure has added to it a pointer to a block which lists the default characteristics of objects it creates. This pointer (only) is copied to newly-created objects. For subsequent access to these objects, assuming (as is always true except in pathological circumstances) the client is set up so it can access objects it creates, only the client and object pointers need be compared; if these are the same, we can be assured the client is allowed access to the object without further checks. Hence in most cases the whole security policy can be reduced to a single integer compare. If the pointers differ, detailed comparisons are required, but in this case it is quite likely the requested access is from some other client with a different ID or MAC level, so the request is likely to be denied. In this case, the performance is relatively unimportant.

4. Display manager (*xdm*)

The display manager, derived from *xdm* from the X distribution, handles the login and logout process. Its functionality is basically the same as normal *xdm*; there are additions to set on login the server "clearance range" (the set of MAC levels allowed for connections), based on the server's "accreditation range" (the set of MAC levels appropriate for that machine) and the user's clearance; and an initial server access control list (ACL), which may be subsequently changed by the user. Any new client connections must lie within the server clearance range and be allowed by the server ACL.

The display manager is also responsible for starting up certain privileged clients (window manager, selection manager, audit manager) in addition to the user's startup list. These clients handle security responsibilities during the user's login session. To avoid any potential "loopholes," the display manager must ensure these privileged clients are up and operational before it starts any user clients. We use job control as a simple means of achieving this. Each privileged client has the single line

```
    kill(getpid(), SIGSTOP);
```

added to it after its initialisation routine. *Xdm* (or rather a child process of it) then starts each privileged client in the foreground (that is, *wait*s for its process state to change). If this occurs because of **SIGSTOP**, *xdm* then restarts the privileged client (with **SIGCONT**) in the background. A single child of *xdm* then

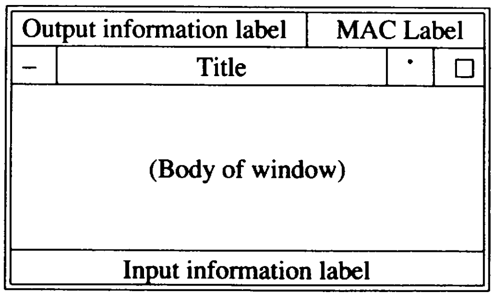


Figure 3: *Window appearance*

waits for any of the privileged children to die; such an occurrence is deemed equivalent to logging out, so the server and any clients are killed, and *xdm* restarts the login process.

5. Window manager (*mwm*)

The window manager, derived from the Motif window manager *mwm*, performs two main security-related functions – displaying MAC and information security labels for windows and handling the trusted path. Bundling these with the window manager does dictate the use of a particular window manager to get these features. However, since these features are mostly intended for the CMW, which already has dictated look-and-feel, we did not feel it to be too severe a limitation, and it certainly simplifies the implementation. (For an alternative, see [Pic90a].)

The window manager displays labels for top-level windows; here the displayed information label is actually the least upper bound of window labels for all windows in that tree. The labels are (optionally) displayed in the border area around a window (Figure 3).

Since the border labels are optional, and can in any case be spoofed by malicious clients, this information is also obtainable by request from a window manager menu. The window manager gets the information it needs by specifying **LabelNotifyMask** (a new event mask) for all new windows; the server then sends it notification of all window label changes.

The "trusted path" (a "non-interceptible" means of communication between users and trusted code) required for security-relevant operations is handled by conventional means, using passive keyboard or mouse grabs to enable user requests, followed by active grabs and/or server grabs as appropriate. (To ensure access to the trusted path, we confine active grabs and server grabs to privileged clients; we must also make other miscellaneous changes, such as disallowing warping the pointer by non-privileged clients when it has been grabbed by privileged ones.) Obviously the overall "trustedness" of this path presupposes the trustedness of the underlying connection. In the CMW target environment, this is true by assumption; in a more general environment this is an open issue.

6. Selection manager (*xsm*)

Selections are the preferred means of inter-client "cut-and-paste" provided by X [Ros89a]. Since selection transfers require two-way communication between the selection owner and requester, the normal application of the security policy would disallow any selection movement between MAC levels (either upward or downward). This is not very useful in the typical CMW environment, where analysts pull together data from various levels to create reports. Even without MAC, the application of DAC restrictions could prevent data transfer by security-innocent clients (the requester could present to the owner a property the owner does not have write access to).

To handle these situations, and more generally the (baroque) CMW selection handling rules (which also call for special information label handling and auditing), a separate client called the selection manager (*xsm*) can be set up to intervene in all selection transfer activity.

It registers itself with the server by specifying **SelectionMask** (a new event mask, reserved to privileged clients). In turn, the server then redirects all **SelectionRequest** and **SelectionNotify** events to it. The selection manager takes any needed action (auditing, user dialog, relabeling selection data), and if the action passes checks, forwards the events to the original recipients. This does require extra server round-trips on all selection transfers, but since these are presumably only user-instigated, the overhead should not be significant. To get around access restrictions, the selection manager may have the owner post the selection data to an (accessible) intermediate window; the selection manager then (perhaps after auditing or user confirmation) transfers the data to the original requester.

The selection manager will only handle selection types it understands (those listed in the ICCCM which use window properties as a data transfer mechanism); other types will work if at all only at the same MAC level, subject to ordinary DAC restrictions. No special provision is made for handling cut buffers, the older method of data transfer, which implies by default they will only work at the lowest MAC level (the level of the cut buffer properties).

7. Audit manager (*xam*)

A variety of auditable events occur in the server; however the server's machine need not be capable of auditing itself. Instead the server sends **AuditNotify** events (a new event type) to the audit manager (*xam*), which has expressed interest in them via **AuditMask** (a new event mask, again reserved to privileged clients). The audit manager is simply a data converter, taking the audit event data (plus any additional information it needs from the server) and converting and storing in the system audit trail.

8. Conclusion

The design described here provides a complex and rich set of security mechanisms, and yet is quite well self-contained: for example, the only modifications required in the network support code on the server machine are to support information labeling. Adding new features by extension of X (adding new protocol requests and event types) simplifies the design and increases its portability, while at the same time eliminates the synchronisation and implementation difficulties which arise from relying instead on changes in lower communications levels to handle these features [Pic90a]. Putting complex or controversial policy aspects in privileged clients makes for a cleaner, more easily configurable design, with almost no real performance impact. Security can be added to X in a manner consistent with its design and without a fundamental change in its nature.

9. Acknowledgements

The refinement of this architecture owes much to discussions with other people. In particular we acknowledge the contributions of R. K. Aditham, Leslie Dotterer, Gary Luckenbaugh, M. Ranganathan, Forrest Stoakes, and Debra Yakov of IBM and Khalid Asad, Sohail Malik, and Mike Muresan of VDG.

References

[Car87a] Mark E. Carson, R. Scott Chapman, Wen-Der Jiang, Jeremy G. Liang, and Debra H. Yakov, "From B2 to CMW: Building a Compartmented Mode Workstation on a Secure XENIX Base," in *AIAA/ASIS/IEEE Third Aerospace Computer Security Conference Proceedings*, Orlando, Florida (December 1987).

[Car89a] Mark E. Carson, Wen-Der Jiang, Jeremy G. Liang, Gary L. Luckenbaugh, and Debra H. Yakov, "Secure Window Systems for UNIX," pp. 441-455 in *Winter 1989 USENIX Technical Conference Proceedings*, San Diego, California (January 1989).

[Cum87a] P. T. Cummings, D. A. Fullam, M. J. Goldstein, M. J. Gosselin, J. Picciotto, J. P. L. Woodward, and J. Wynn, "Compartmented Mode Workstation: Results Through Prototyping," in *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Oakland, California (April 1987).

[Fis87a] Burns Fisher, *X11 Server Extensions Engineering Specification,* Digital Equipment Corporation (August 1987).

[Joh85a] M. St. Johns, "Authentication Server," RFC 931 (1985).

[Joh88a] M. St. Johns, "Draft Revised IP Security Option," RFC 1038 (1988).

[McI88a] M. D McIlroy and J. A. Reeds, "Multilevel Windows on a Single-level Terminal," in *Proceedings of the UNIX Security Workshop*, Portland, Oregon (August 1988).

[Pic90a] Jeffrey Picciotto, "Trusted X Window System, Volume 1: Design Overview," Mitre paper MTP 288 (1990).

[Ros89b] David S. H. Rosenthal, *LINX a Less INsecure X server,* Sun Microsystems, Inc. (1989).

[Ros89a] David S. H. Rosenthal, *X Window System Version 11 Inter-Client Communication Conventions Manual version 1.0,* Sun Microsystems, Inc. (1989).

[Sch86a] Robert W. Scheifler and J. Gettys, "The X Window System," pp. 79-109 in *ACM Transactions on Graphics* (April 1986).

[Sch90a] Robert W. Scheifler, *X Window System Protocol MIT X Consortium Standard, Version 11, Release 4,* MIT Laboratory for Computer Science (January 1990).

[Smi89a] Barbara Smith-Thomas, "Secure Multi-Level Windowing in a B1 Certifiable Secure UNIX Operating System," in *Winter 1989 USENIX Technical Conference Proceedings*, San Diego, California (January 1989).

[Woo87a] J. P. L. Woodward, "Security Requirements for System High and Compartmented Mode Workstations," DIA report DRS-2600-5502-87 (REV 1) (November 1987).

Mounting filesystems in a networked environment

or

Why do I have to wait for *that* server to come up?

Christer Bernerus

Department of Computer Science
Chalmers University of Technology
S-412 96 Gothenburg
Sweden
bernerus@cs.chalmers.se

ABSTRACT

The typical file system hierarchy dates back to the really old UNIX ages, long before the invention of networked file systems. This hierarchy has only been slightly revised through the years. This article describes a scheme for mounting NFS and local file systems in a consistent way avoiding some of the pitfalls that the networked context introduces if the tradition is followed.

1. History

Once upon a time, a long time ago (Late 1950s and early 1960s) the timesharing systems were developed. These proved invaluable in the support of program development activities compared to the older punched-card systems. The timesharing systems were built around one central computer and a number of alphanumeric terminals connected through separate cables to the CPU.

As time went on, hardware development made it possible to shrink a computer into a box so small that it could be placed on a table and at the same time be so cheap that a small university could afford having plenty of them, while the big computers were still extremely expensive.

At the same time, demands grew and big computers started to be replaced with many smaller boxes and the workstation was born. However, in the beginning, the workstation was essentially a terminal with computing power, but the ability to share programs and data between the users of a system was lost since there were no networks that tied the boxes together, so networks had to be invented. Networks replaced

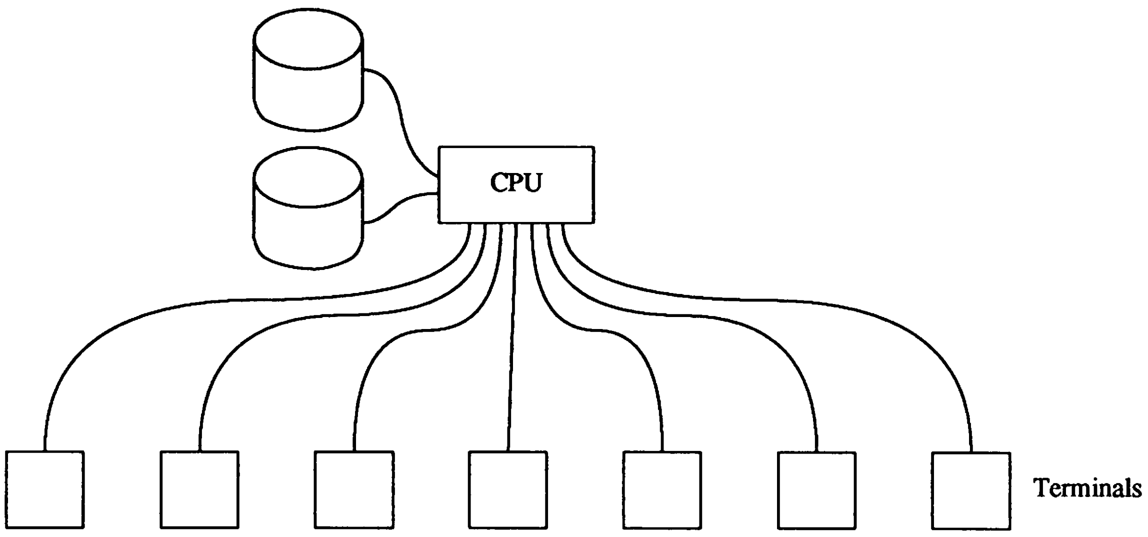


Figure 1: *A typical timesharing system*

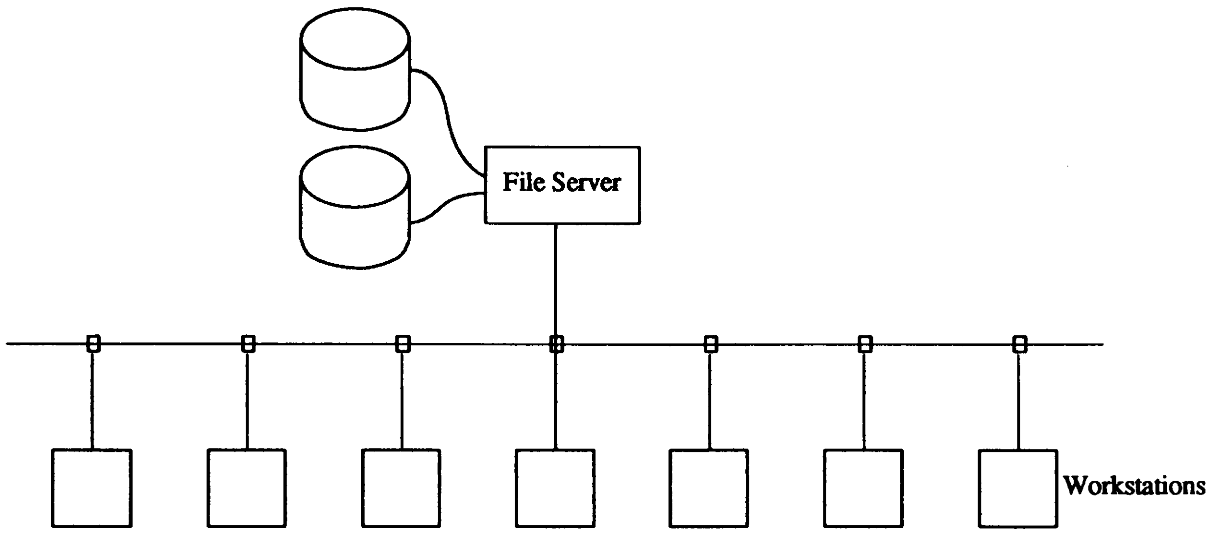


Figure 2: *A typical early server/workstations system*

the separate cables to the CPU with some communications channel where no computer had any special status, electrically seen.

Networks made it possible to copy files and programs between the workstations using special network commands. This was rather clumsy and didn't give back the old "time-sharing feeling". It also created a lot of confusion and consumed a lot of disk space since multiple copies of the files and programs had to be maintained, copies that just *might* be up to date. These difficulties pushed development further and we got networked file systems.

When these were developed, the typical computer installation still had a central machine and a lot of connected serial terminals and also some networked workstations. It seemed natural to use a central computer for storing the files and have the workstations image that filesystem locally.

The first workstation installations looked very much like the old timesharing systems but had different labels on the boxes, File server instead of CPU, workstation instead of terminal, but the typical system had one (1) server and a number of workstations.

This setup of course didn't solve the performance problems forever. Now we have many servers and just too many workstations. The file system is spread out over all of the servers (who acts as both servers and clients themselves) creating an awful lot of cross mounts between the servers. Usually, users files have been moved to different servers in order to spread out the load.

One area that has not undergone any significant change during history is the file system hierarchy and especially the places where file systems are mounted has been almost untouched.

2. Problems with the mount point placement.

The transition from one server systems to multi server systems has revealed some weaknesses in the way we usually mount file systems.

2.1. The getwd problem.

Since we at Chalmers began using UNIX in 1976/77, users home directories have always been put in a directory */u*. When the user community grew, more partitions were needed. These were named /u1, /u2 etc. and were mounted on the root and the directory /u changed to be a directory only containing symbolic links to the real home directories in /u1, /u2 etc. This setup worked for several years until the networked file systems showed up. When file server numbers increased and the system got more and more complex, it proved to be a not so good idea. Not that the links created problems but the fact that all of the users file systems were mounted directly on the root did. The problems showed up gradually when servers were added and it took quite a long time to figure out what was going on, mostly because the servers were fairly stable. The turning point came when we installed a Sequent Symmetry that a small group of researchers interested in parallel computing used. At some point these users' home directories were moved to the Sequent in order to reduce network load and the machine suddenly became a file server as well. Now when the Sequent was down for different reasons people started complaining:

Appearances of *getwd*				
admin	cpio	find	lp	pcat
at	csh	ftp	lpr	praudit
bar	dbx	gdb	lsw	prs
cc	dbxtool	gemacs	m4	pwd
chgrp	dc	get	[Mm]ail	ranlib
chmod	delta	iconedit	make	rcs
chown	df	in.ftpd	mount	rcsdiff
ci	emacs	ld	mount_tfs	rcsmerge
co	filemerge	lint	on	refer

Table 1: *Executables in a SunOS 4.1 into which getwd is linked*

User:	–	Hey, what's wrong? Everything hung up?
SA:	–	Oh, we just took the Sequent down for backup, it'll be back in a few hours.
User:	–	The Sequent, what kinda machine is that? I never used it! Why #@!%$@ do *I* have to wait for it ???!?!
SA:	–	Hmm, say, what was the last thing you did?
User:	–	I didn't do nuthin', just typed *pwd* t'see where I was n'everythin' stopped!

This started a small investigation especially concerning what happens when typing *pwd*.

First, we thought that */bin/pwd* might be inaccessible, but since */bin/pwd* is located in the root file system, the server having *pwd* was easily found, doing its normal business. Therefore the problem must be in what pwd actually does.

Now, pwd is actually only a shell interface to *getwd(3)* which is used whenever the path to the current directory is needed. Getwd is actually quite a commonly used subroutine. Table 1 shows all the commands in a SunOS 4.1 into which getwd is linked.

This calls for an analysis of the algortihm of getwd. Fortunately this was easily done since source code was available. The algorithm[1] is shown is Figure 3.

The algorithm have the consequence that for each directory in the path *all entries* might be stat'ed. A stat call on a mount point will generate a request to the server that hosts the directory in question. Now, if that server is down we will either wait a long time for a timeout or wait for the server to come up again depending on the behaviour of the networked filesystem and the way mounting was done. The wait is commonly noninterruptible which means that if the user types `%pwd` (or something else uses getwd) the process will wait in the kernel and block at least the process or window used.

```
stat(".") and save the filesystem and inode numbers.

while( stat("..") not returns the saved numbers )
        /* i.e. we haven't reached the root */
{

        opendir("..")

        Use readdir() and stat() each entry until the saved numbers are found.
        When found, prepend the name of the directory to the path.

        In principle chdir("..");

        stat(".") and save the filesystem and inode numbers;

}
prepend "/" to the path.
return a pointer to the path;
```

Figure 3: *Algorithm of getwd(3)*

1 In principle. I don't dare to expose any source code.

From a technical point of view this behaviour is correct, but the user usually has another view, and if the server that is down is one that he/she *never* uses, things are getting a bit difficult for the SA to explain.

2.2. Cross mounting

This is a problem that doesn't show up in the small installations. When we have two servers, the following problem may occur:

1. *Kermit* has xy0g mounted on /usr
2. *Kermit* mounts /usr/src from *piggy* and wants to mount xy1g on /usr/src/X.

Now if *piggy* is down when *kermit* mounts /usr/src, and the mount times out, the mount of /usr/src/X will fail since the mount point doesn't exist yet. Note that *this* mount will *not* be put in the background.

When the third server is installed we really get the possibility of getting gray-haired; for instance:

1. *Kermit's* xy2g contains what should be mounted on /usr/src/X/contrib.
2. *Piggy's* xy1g contains what should be mounted on /usr/src.
3. *Gonzo's* xy1g contains what should be mounted on /usr/src/X.

All servers cross mount in such a way that they all see the same tree structure under /usr/src.

Let's see what happens after a power failure:

1. All servers boot and mounts their local /usr file systems, all well so far.
2. Kermit wants to mount /usr/src from *piggy* and waits for *piggy* to come up.
3. *Piggy* wants to mount /usr/src/X from *gonzo* and waits for *gonzo* to come up.
4. *Gonzo* either wants to mount /usr/src from *piggy* or /usr/src/X/contrib from *kermit* and waits for one of them to come up.

When will the system be available?

The servers will eventually come up, but some mounts will fail and in the worst case a complete deadlock will occur.

3. Another way of mounting file systems

Some of these problems will surely be fixed by changes in the operating systems. Rumors say that SunOS 4.1 uses a mount point cache to fix the getwd problem above. However until these fixes has found their ways into all the OS'es of different vendors that we use, we've tried to attack our problems not by some obscure hacking[2] but by reorganizing the way we mount filesystems guided by a few rules described below.

3.1. Solving the getwd problem

Even if the getwd problem is the least serious of the two, it turns out that the solution of this problem also solves the cross mounting problem. The algorithm of getwd suggests that a directory should only contain mount points from one server.

Rule 1A: Only one server's mount points in each directory and no mount points directly in the root directory.

This way, when getwd searches its way up to the root it won't try to stat any mount point on another server than the one involved in the path.

This suggests a structure as in Figure 4 (but note that the file tree must be tied together using symbolic links).

This structure unfortunately has a serious drawback: We are entering server names into the structure. This means that the structure and the symbolic links must be changed if a filesystem is moved to another server or (God forbid) a server is renamed. So we also must have rule 2:

Rule 2: No server names in the file system structure.

2 Source code are becoming a scarce resource.

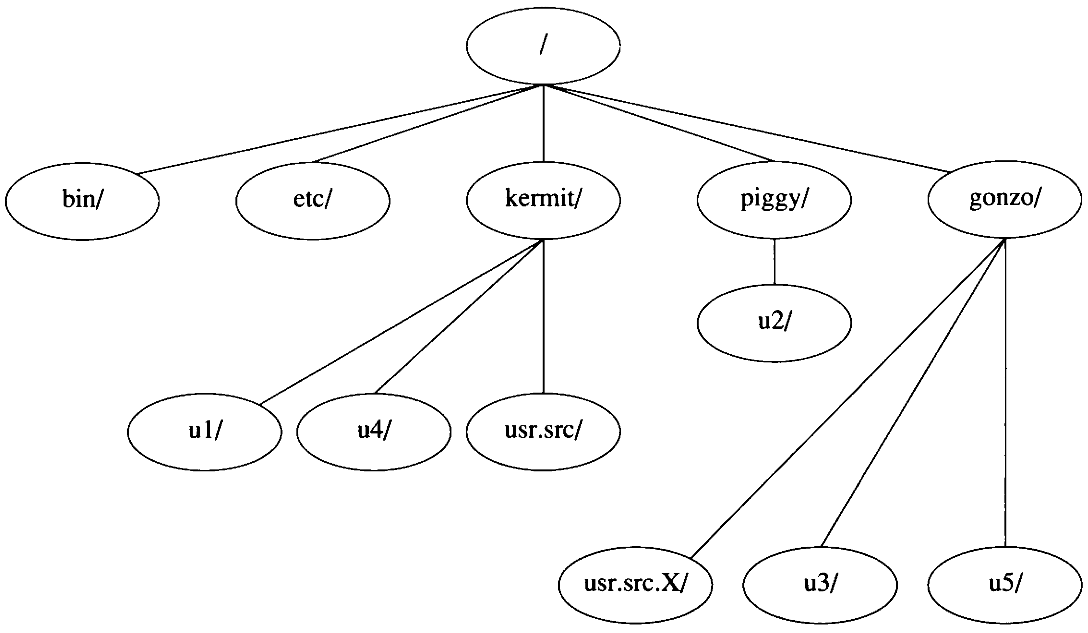


Figure 4: *New organization, first suggestion*

The next suggestion would of course be to replace the machine names with some arbitrary bogus names which will not have to change if a server is renamed. However, this will only solve the problem of server renaming which doesn't happen very often, but if a filesystem is moved to another server, the same problem persist.

If we want to make movement of filesystems easy, and adhere to Rule 1A and 2, we need a more restrictive rule:

Rule 3: Only one mount point in a directory

This rule also implies rule 2 when more than one filesystem is involved on each server.

This at first suggests the structure in Figure 5:

This has the minor drawback that we will have to invent meaningless names (A, B, C etc) whenever a filesystem is added. The most meaningful thing to do is to use the filesystem names again instead of A,B,C etc. This will yield:

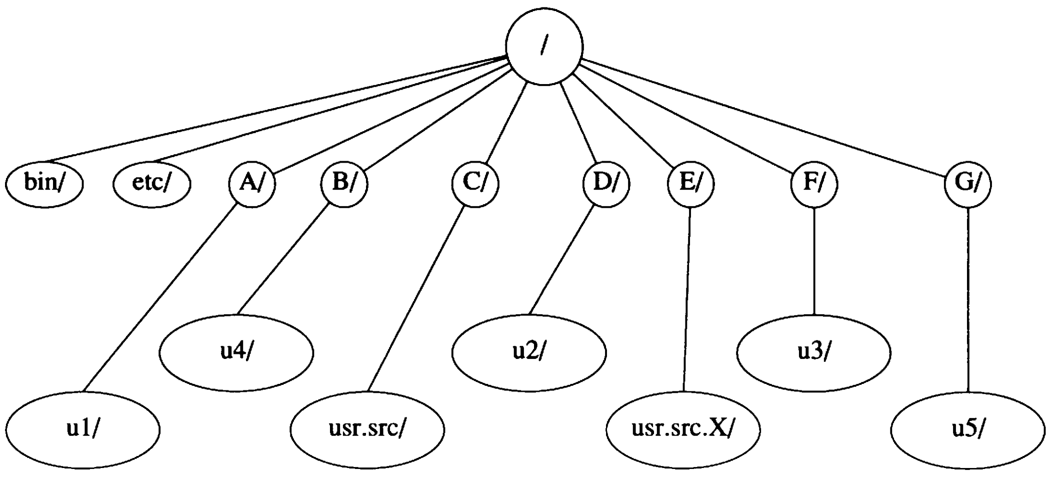


Figure 5: *New organization, second suggestion*

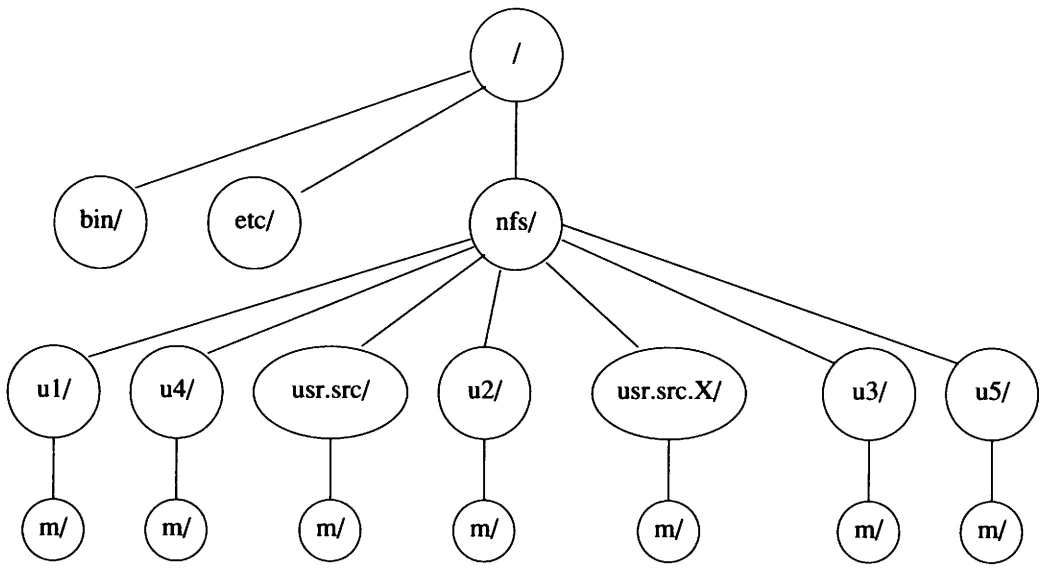


Figure 6: *New mount organization for NFS file systems*

```
/ul/ul
/u2/u2
/u3/u3
/usr.src/usr.src
/usr.src.X/usr.src.X
/usr.src.X.contrib.toolkits.andrew/usr.src.X.contrib.toolkits.andrew
```

Which also have the advantage of showing, among other things, all the filesystem names when listing the root. The last example however illustrate the big drawback with this structure: paths may get *very* long. We also see that now the names at the second level (the real mount points) are really superfluous. They can be replaced with something more simple and since they are all alone in their directory they can have the same name. Since it is a mount point let's choose "m":

```
/ul/m
/u2/m
/u3/m
/usr.src/m
/usr.src.X/m
/usr.src.X.contrib.toolkits.andrew/m
```

Now, mounting this way, filesystems can be moved between servers just by editing the mount table (and moving the data of course). Server renaming creates no change in the structure and getwd won't block on uninteresting crashed servers.

When this is applied to a large system, we tend to have quite a lot of these mount directories cluttering up the root directory, Therefore we chose to collect them all in a subdirectory. Since the structure is tailored for NFS mounts we chose /nfs, and thus we have the final structure in Figure 6:

If we refrain from mounting local filesystems here, this structure also gives us a spinoff effect which may be good or bad, but is usually good for the SA:

> It is now very easy to see whether a directory is local to the machine or if it is mounted from another server. Just type `%pwd` and the answer will be printed. Also a `%ls -l /nfs` will give a list consisting only of the (mount points for) networked file systems.

3.2. Solving the cross mounting problem.

To solve this problem we need to adhere to another rule:

> **Rule 4: The mount point should always exist.**

This is because we want to avoid the effects of cross-mounting described above. If this rule is obeyed, the order of which mounting is done will be insignificant and thus *much of the mounting can be put in background* when a machine is booted. This means that a server will come up quicker and be ready to export file systems to other servers.

The structure described in section 3.1 fulfills this rule since all the mount points are statically created in the root partition which is guaranteed to exist or otherwise the machine wouldn't be up.

3.3. What about local filesystems

So far we have only dealt with networked file systems. What do we do with local file systems? Do we really need to touch these. Aren't they still good old UNIX file systems?

Well, yes, but the environment where local file systems are mounted might have changed. As long as we are sure that all of the tree above a local mount point is local to the machine we can mount filesystems directly in the tree. This is however nowadays almost only true for the /usr and the /tmp filesystems.[3] For other filesystems we can use a similar approach and we can have:

```
/usr.local/
/usr.bin.X/
```

Stuffing this into another directory (not /nfs, then we loose the spinoff effect above) say "/ufs" we have

```
/ufs/
     usr.local/
     usr.bin.X/
```

Here we don't need the /m extension since the getwd problem doesn't exist for local filesystems, although it can be included for consistency.

3.4. Tying up the file tree.

Now we have to make the file tree look (almost) as before again. This is of course done using symbolic links to the new mount points. Unfortunately it isn't as simple as that because a **big** trap is now hidden here. On the workstations the /usr/src filesystem should be mounted on /nfs/usr.src/m but the server that actually have the disk containing /usr/src mount this on /ufs/usr.src. Now the /usr file system shall contain a symbolic link to the mount point of /usr/src. This name isn't the same on the server and the workstations, but the contents of the symbolic link is since the /usr file system is shared between the workstations and the server. Thus we cannot put in links in the tree pointing to the /nfs and /ufs directories directly since the symbolic links in the file system tree will most probably be the same for all machines. Therefore we have to create two link chain where the second link is allowed to vary between machines. These are collected in a third directory under each machine's root. Let's call this `/lfs`[4]. This directory doesn't contain any mount points but only a collection of symbolic links to the actual mount point on the machine in question. Figure 7 shows the setup of a workstation and its server.

Now the directory /usr/src can be removed and be replaced by a symbolic link to `/lfs/usr.src`.

If all of the mount points, both networked and local, are collected under (but not within) one directory this is unnecessary, but then, the same structure must be used for both local and networked file systems.

4. Summary of the new organization:

File systems are named after their ordinary placement in the file system tree with the first "/" deleted and the other ones replaced by dots, e.g.

```
/usr/src/X -> usr.src.X
```

Networked filesystems mount at `/nfs/<fsname>/m`.

Local filesystems mount at `/ufs/<fsname>`.

All *links to filesystems* go through `/lfs/<fsname>` which point out the real mount point. Figure 8 shows the new organization with all details.

5. Pros and cons

Now the interesting question arises: What do we gain and what do we lose?

3 At least on file servers.

4 The name `/lfs` is grabbed out of empty space, maybe something more informative should be chosen like `/fslinks`.

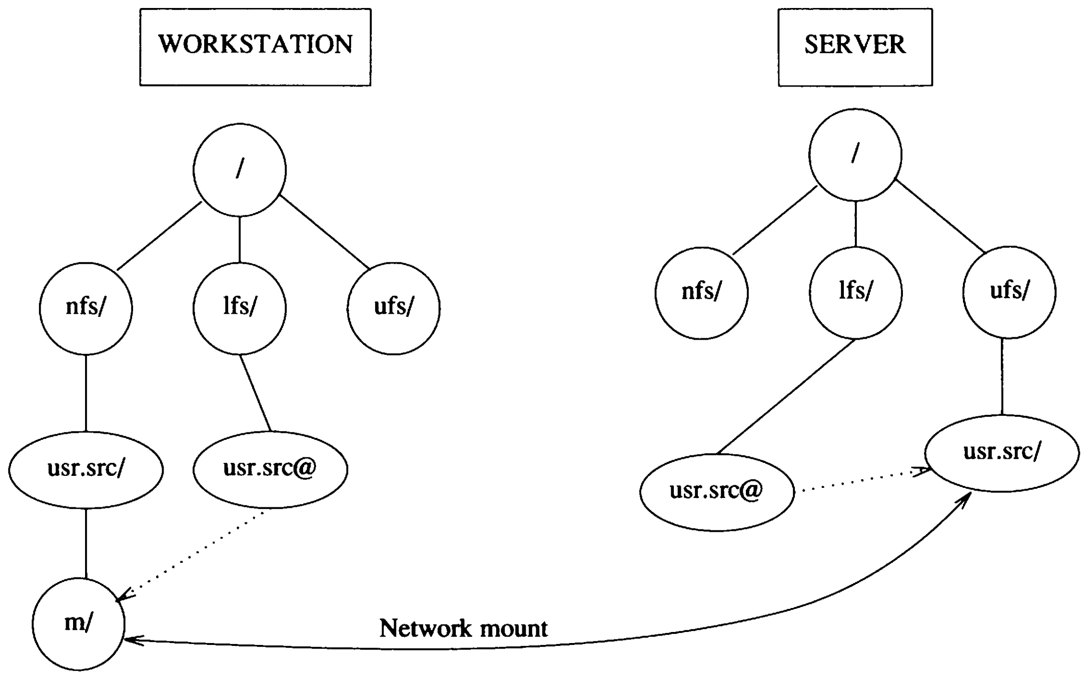


Figure 7: *Final link of the filesystem link chain*

5.1. Advantages

1. We do not have any mounts that depend on each other. Most of the mounting can be done in any order which means that the NFS part of the mounting at startup can be put in background immediately and each server can be brought up *much* faster especially when the other servers are down.
2. When a new filesystem is to be installed, its placement in the file system tree doesn't restrict us when we select which machine that shall host the filesystem.
3. When a server is down, the probability of hanging up the workstations has decreased substantially, at least when the crashed server is unimportant to the user.
4. *Getwd* now works faster and generates less network load since a maximum of 1 (one) server is contacted. Formerly it wasn't uncommon to contact ALL servers when getwd was called.
5. The users can now easily determine if they are on a networked or a local file system.
6. No kernel hacks or source code is needed. This can be applied to *any* UNIX system that supports symbolic links.

5.2. Disadvantages

1. If anyone in some way depends on the path given by getwd the change may cause trouble. We have experienced no trouble at all (yet). This also means that the result of getwd should be consumed immediately and not be stored anywhere for later use or usage on another machine.[5]
2. Care must be taken not to split a file tree where one depends on the correct semantics of "..". This means that splitting e.g the X-windows source tree in this way will cause you trouble.
3. The increased usage of symbolic links will contribute somewhat to the system load.

5 This might seem to be a severe restriction, but doing this could be considered bad practice anyway since users might want to use their home directories networked to some other machine that could have another path to home directories. I have that problem: At CS dept my home directory is `/u/bernerus` (or really /nfs/u4/m/bernerus), but at Computer Engineering it is `/afs/u/utc/bernerus` If I use something that relies too much on what getwd returns, I will be in trouble.

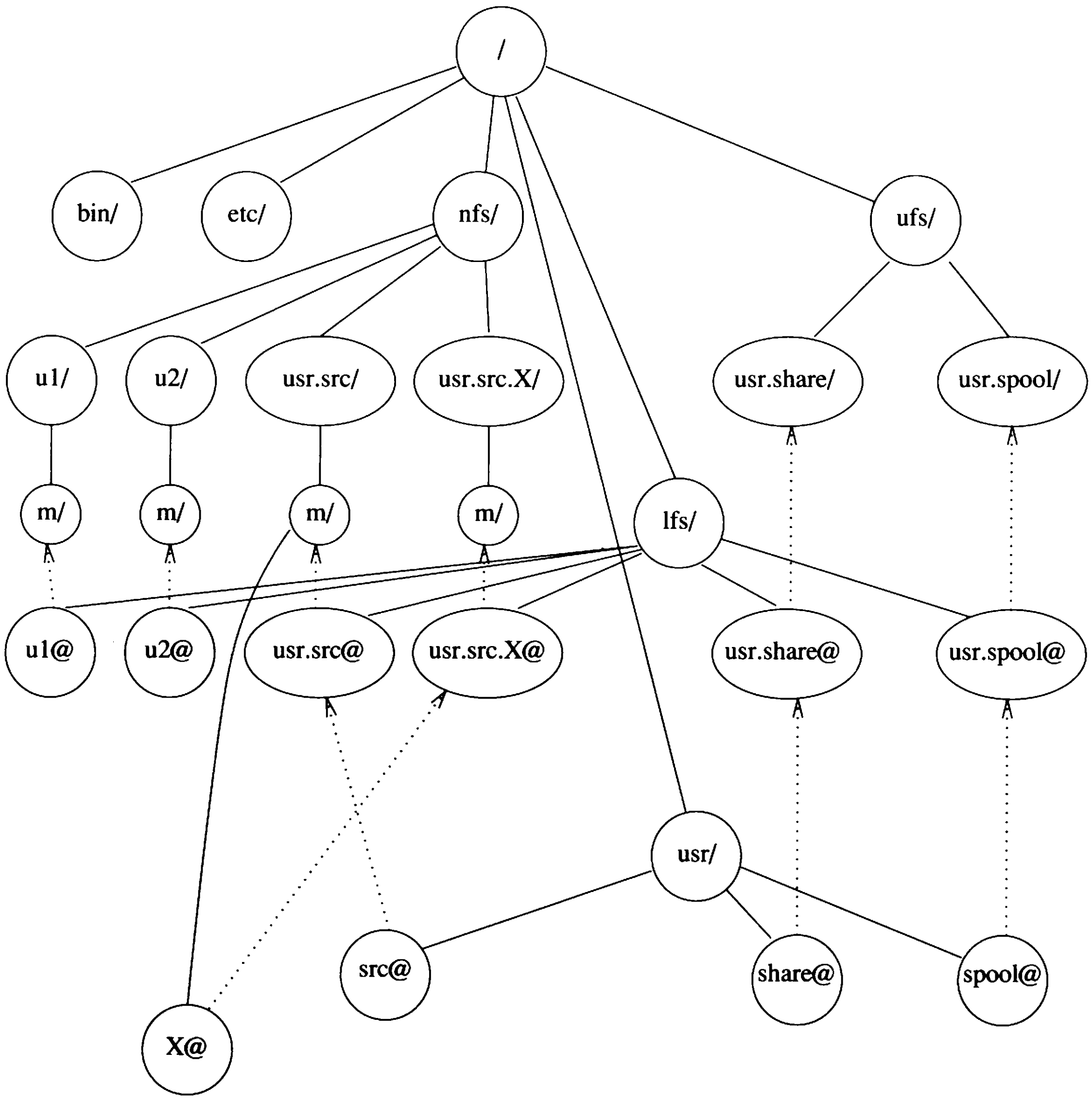


Figure 8: *New mount point organization*

6. Conclusions

An alternative scheme for mounting NFS and local filesystems has been implemented which has proven to solve some problems encountered when expanding the simple one-server systems to multi-server and multi-vendor systems. Server crashes no longer create as much disaster as before. Network load has decreased and faster responses has been noted. We now immediately can create a mount point for a new file system without having to think twice before deciding where mounts should be done. We also have more freedom of choice between servers when a new filesystem should be installed. Servers and workstations start up *much* faster since no mounts depend on each other and we no longer have any risk of creating mount deadlocks.

7. Acknowledgements

I wish to thank the following people who have contributed to the completion of this article:

- **Urban Lane** for patient listening, and good points while the first thoughts on this scheme came to our minds.
- **Gunnar Lindberg** for pointing out the dangers of having server names in the directory structure.

- **Mark Moraes** for writing *xpic* which made it possible to finish this article before deadline.
- **Borje Lindh** and **Gorgen Olofsson** for encouraging me to write this article at all, and last
- **Uwe Untermarzoner** whose enthusiasm gave me the final kick to finish this paper.

Developing Documentation to Meet the Changing Needs of UNIX Users

Linda Branagan
Marilyn J. Lutz

Convex Computer Corporation
3000 Waterview Parkway
Richardson, TX 75080 USA
branagan@convex.com

ABSTRACT

In recent years, the UNIX operating system has found its way into fields ranging from medicine to mechanical engineering. Some UNIX users are no longer necessarily computer scientists – often they are educated and employed in other fields. To these users, UNIX is simply a tool through which an important set of tasks is accomplished. As a result, they apply a different set of strategies when they use software and the documentation that accompanies it.

Traditional forms of UNIX documentation do not serve the needs of these users particularly well. This paper discusses methods for developing documentation that does meet their needs.

1. Introduction

In recent years, use of UNIX has expanded greatly. It is no longer reserved for use only in computer science. Instead, it has found its way into dozens of fields, ranging from chemistry to geology. The people who use it are not necessarily computer scientists – often they are educated and employed as scientists or engineers in some other field.

To these users, UNIX and the hardware it runs on is not the object of their work, instead, it is set of tools they use to accomplish other goals.

2. The "Non-Expert" User

"Non-expert" users are people who are educated and employed in some field other than computer science, who use UNIX as a tool to get their jobs done. The term "non-expert" is used to distinguish these users from novices. Novice users are beginning users, they are expected at some point to become experts. Non-experts often have years of computing experience, and they often develop computing expertise in one or more limited areas. However, they don't normally possess a broad base of background knowledge simply because they don't need to, they can typically rely on system administrators or other local experts when they need to.

The strategies that non-experts employ when using system-level software and its associated documentation are different than those of more traditional, computer-oriented, expert users.

3. Accessibility

Non-expert users are very task-oriented users; they use documents long enough to figure out how accomplish a necessary task are rarely read further. As a result, they need documents that are very accessible. They must be able to pinpoint and absorb the information they are seeking an a very short period of time. There are several methods that can be used in increase the accessibility of a document.

3.1. Titles

The title of a document can go a long way in indicating its audience and contents. "User Guides" are different from "System Manager's Guides" which are different still from "Installation Procedures".

Several small documents, divided and titled according to the types of tasks they describe, are often much more effective than one large manual. Specifically, user information should be strictly separated from system administration information. Users are rarely interested in installation, configuration or maintenance procedures. Likewise, system administrators rarely require explanations of every command line option offered by the software. It will take less time for both types of readers to find the information they need if the documents are small and the type of information in each book is clearly indicated by its title.

Also, large documents are unfriendly and uninviting. By dividing information by audience, smaller manuals are generally produced.

3.2. Chapters and Headings

Like titles, useful chapter names and subject headings can greatly increase the accessibility of a document. They make scanning the Table of Contents a useful method for locating information in a document or determining that the desired information in *not* in that document.

Useful chapter names and headings also aid in the user's ability to scan pages of text to find the location of the desired information, and they indicate which chapters or sections can be skipped.

4. Concepts Guides

There are times when it is important to convey background information to an otherwise task-oriented user. At CONVEX, we have found brief, high-level descriptions of software packages, called Concepts Guides, to be extremely useful. These books are specifically designed to provide users with the minimal amount of background information required for efficient use of some piece of software.

Concepts Guides have been written for several different software products including a distributed batch system, a high speed networking package and a process scheduler.

Since users are often impatient when it comes to reading manuals, Concepts Guides emphasise the effect the software will have on the user. Particular attention is paid to discussing how the user's working environment will be improved or altered by the software. Users often become very interested in a piece of software if they know that their working environment will look different or become more efficient after it is installed. As a result, they may pay closer attention to explanatory documents.

Concepts Guides are usually less than fifteen pages long and are written in an informal style. They are often organised in a question-and-answer format, which makes it very easy for users to determine if this document will suit their needs. If the user doesn't care about the questions that appear in the headings, she doesn't need to read on.

Also, in cases where the user is completely clueless about a piece of software, question-like headings can help a user identify their own concerns.

For example, the following headings appear in the *CONVEX Share Scheduler Concepts* document:

What is Process Scheduling?

What is Share and Why is it Useful?

How are Shares Divided Among Users?

What Happens if I Use Up My Share?

The answers to the first two questions convey background material in a brief, easy-to-digest form. The second two questions address the way in which the user's working environment will be affected.

The last question, "What Happens if I Use Up My Share?" indicates to the user that using up his share is indeed possible and is something he should take care to prevent.

Since these documents convey high-level concepts, diagrams and figures are also used extensively.

5. User Guides

User manuals require an increasingly cookbook-oriented approach. Users are not typically willing to learn about all the functionality available from some piece of software so they can determine which subset suits their needs. Instead, they tend to look for and follow examples. Therefore, examples that might conceivably match or come close to what readers want to do are designed and included. Users are much more content to follow instructions that say "type this" rather than "figure this out." Nothing makes a user quite as happy as when she realizes that the results she wants from a given piece of software can be produced exactly by following the procedure described in the User's Guide.

An example of this can be drawn from the CONVEX Share Scheduler software. Share allows users to be divided into scheduling groups. The priority of each user process is based on (among other things) the scheduling group the user is in. It is useful for users to be able to display a list of all the scheduling groups in the current configuration. This is done with the following lengthy command:

```
    pwintf 1 '%s\n' sgroupname | sort | uniq
```

pwintf is a *printf*-like utility that is used to display data from the shares database. It isn't very likely that a non-expert user will find its syntax intuitive. Likewise, *sort* and *uniq* are standard UNIX tools with which the user may or may not be familiar. The chances are good that a non-expert user won't think to combine all three of these programs to produce the desired results. Instead, the user may decide that it simply isn't possible to extract a list of scheduling groups.

However, if this command line is documented under the heading

Displaying a List of All Scheduling Groups

the user will be happy to type it in and use the results.

When the software is too complex to describe largely through examples or when a decision-making process absolutely requires input from the user, step-by-step lists of instructions or flow diagrams are often provided.

6. Reference Documents

Reference documents are often provided to supplement Concepts Guides and User Guides. They are not designed to be a sole source of information. Instead, they provide a way for users to find details that they won't normally commit to memory. CONVEX provides three different types of reference materials: Quick References, Reference Pages and traditional UNIX man pages.

6.1. Quick References

Quick References are small booklets or cards that contain extremely brief command examples. As with other documents, it is useful to organise information in a Quick Reference by task. For example, commands for submitting jobs to batch queues should be separated from command for deleting jobs.

6.2. Reference Pages

Reference Pages contain information that is very similar to the information presented in a man page. However, since they only exist in hardcopy, graphic design techniques are used to make the page appear more inviting than a man page.

6.3. Man Pages

Like other reference documents, traditional man pages are not particularly effective when they are a user's only source of information on some topic. However, they can be an extremely useful reference for non-expert users, particularly because they are always available online.

7. Expert Users

Often, the same writing strategies that are used to make documents more accessible for non-experts apply directly to documents written for experts.

7.1. Concepts Guides for Experts

For example, CONVEX has produced a *POSIX Concepts* guide. Since you have to be working fairly close to the operating system to be severely affected by POSIX compliance, this document is obviously not designed for the non-expert user.

It does, however, take advantage of many of the strategies that improve accessibility in documents for non-experts. It has some headings in the form of questions, including

What Is POSIX?

Why Use POSIX?

A reader who is already familiar with this standards movement will probably skip both these sections. However, a reader who has heard very little about POSIX will probably choose to read them.

7.2. System Manager's Guides

The cookbook-style approach that is used in User Guides can be appropriate for System Manager's Guides, particularly when they are used to guide the reader through an important decision-making process. Often, system managers need to know not only how to accomplish some task, but how to decide when to do it as well.

For example, the *CONVEX System Manager's Guide* contains instructions on creating striped file systems. Until recently, however, the book contained very little information that helped system managers decide which file systems on their machine, if any, were good candidates for striping. This is a particularly dangerous omission, since bad decisions about disk striping can often drastically impede I/O performance.

A step-by-step approach to a decision making process in this example might include questions like:

> Is I/O activity for this file system normally very high?
>
> Is it necessary for you to exceed normal limits on file system size?
>
> Do you have the appropriate hardware?

This format tells system managers what kind of information they will need to gather to make a good decision about disk stripes.

8. Conclusions

Many modern UNIX users are skilled, educated scientists and engineers who work primarily in some field other than computer science. Three types of documents can be used to fulfill their needs:

- Concepts Guides
- User Guides
- Reference documents

Concepts Guides present high-level overview information, and contain

- question-like headings
- descriptions of the changes that may take place in the user's work environment
- descriptions of improvements to the user's environment that may take place
- extensive diagrams and figures
- fewer than fifteen pages of text

User Guides present detailed descriptions of the cabilities of the software and

- task-oriented headings
- many examples

Reference documents are used to look up details and can be in the form of Quick References, Reference Pages or traditional UNIX man pages.

Many of theses strategies, which are designed to make documentation more accessible for the non-expert, can be used in expert-level documentation to improve its accessibility as well.

DIAMOND – An Object Oriented Graphics Library

for Software Development Environments

Ralph Zainlinger
Gustav Pospischil

Technical University Vienna
Treitlstr. 3/182/1
A-1040 Vienna
Austria
ralph@vmars.uucp
gusti@vmars.uucp

ABSTRACT

The paper presents an object oriented graphics library that is particularly suitable for the construction of user-interfaces for Software Development Environments (SDEs). The library is based upon an object oriented interaction model that has been specifically tailored to meet the requirements of software engineering and human-computer interaction.

The library along with tools that support interactive construction of user interfaces forms a toolkit that can be used as part of the infra-structure of any UNIX based SDE.

1. Introduction

Software Development Environments (SDEs) are characterised by a number of integration mechanisms among which management integration, data storage integration, and interaction integration constitute the most important ones. One challenge for today's builders of SDEs is the preservation of integration even if the environment is enhanced with novel tools. Particularly critical is the design of a tool's user interface, as the user interface forms naturally that part of the tool the user is most confronted with.

The generation of user interfaces is usually assisted by (a) User Interface Management Systems (UIMSs) or (b) toolkits [Mye89a].

UIMS are based on the idea of separating the design of the user interface from the development of the application. Since this separation is to some extent "artificial" it gives rise to various problems which are not yet satisfactorily solved.

Toolkits usually comprise various (graphical) tools that help the designer to interactively construct and arrange the graphical elements. Along with these tools a graphics library is provided that represents the interface to an underlying programming language.

Both kinds of assistance (UIMSs as well as toolkits) are well suited to support specifying "what" is allowed to appear on the screen (windows, buttons, icons, etc.) but they do not say "how" and in which combinations these objects should be effectively used. Particularly UIMSs are designed to cope with a variety of different interface styles, i.e. a tool designer is not prohibited from producing inconsistent interfaces within an SDE.

To cope with these apparent problems of user interface design we developed:

- An object oriented interaction model that is particularly suitable for SDEs [Zai90a].
- DIAMOND, a graphics library specifically tailored to support the object oriented interaction model [Pos90a].

The graphics library is part of a (still growing) user interface toolkit that can be used to generate interfaces for tools of SDEs.

The user interface toolkit currently consists of an interactive form editor, a menu editor, and an icon editor.

This work was supported in part by the ESPRIT Basic Research Project 3092 "Predictably Dependable Computing Systems".

The main focus of this paper is on the particular graphics library that is built on top of *Xlib*, the basic library of the X Window System [Sch86a, Jon89a].

The structure of this paper is as follows. Section 2 presents the object oriented interaction model. Section 3 describes the basic concepts of the graphics library and how they relate to the interaction model. Section 4 gives a more detailed description of the library and focuses on some implementational details. Section 5 presents a sample DIAMOND application.

2. The Object Oriented Interaction Model

2.1. Objects, Actions and the "Natural Way" of Human Thinking

Many – even graphical – user interfaces are dominated by an interaction scheme based on a sequence *<command><options><object>*. This scheme has formed the basis of the line-oriented command-driven interfaces (e.g. UNIX) for a long period of time and has been adopted by most of the menu-based graphical interfaces (e.g. Apple Macintosh).

We feel that command orientation does not always reflect the "natural way of human thinking", which corresponds rather to a sequence *<object><activity><tool>*. Humans first select an object, then determine an activity and finally choose an appropriate tool, e.g. *<orange><peel><knife>*. Since the determination of the activity and the selection of the appropriate tool are blended in the human mind, the sequence can be reduced to an *<object><tool>* relation whereby the activity is implicitly established by the object-tool combination (e.g. *<orange><knife>* implies "to peel").

If we consider a tool as an object the whole process can be described by interrelated objects. Thus,

> *a graphical object oriented user interface can be determined exclusively by objects, where the relation between objects implies an action.*

By this simple abstraction (tools are treated as objects) the model is capable of supporting both interaction schemes. Performing an activity can be established either by first identifying the tool (corresponding to the "command first philosophy") or by first selecting the object (corresponding to the "object first philosophy").

Unfortunately this rigorous approach is not practical in general for the following reasons:

1. The relation between two objects does not necessarily imply one unique and meaningful action: on the one hand several object relations may yield absurdities (e.g. *<orange>* and *<vacuum-cleaner>*), on the other hand one object combination can result in several plausible but ambiguous interpretations (e.g. *<orange>* and *<hand>*).
2. A large number of objects would be required to describe a large amount of functions and tools (i.e. most of the operating-system interfaces).
3. A command is executed by the relation between two objects. How can a novice get the information which combinations are possible and which actions will be triggered?

2.2. Adapting the Rigorous Object Oriented Approach

This subsection shows how the previously mentioned disadvantages are overcome by three general improvements.

The first disadvantage can be eliminated by graphical techniques. The moment an object is selected, the objects that cannot be combined in a useful way will be specially indicated. Thus, the user can easily recognise all valid combinations. If a chosen object combination defines several valid actions (problem of ambiguity), the user selects the intended action through a dynamically displayed pop-up menu.

To avoid information overflow for the user by representing too many different objects we decrease their number by not displaying seldom used objects. Those objects which are only used in a few combinations will be presented to the user in dynamical pop-up menus, whenever a combinable (more often used) object is selected. As argued in [Zai90a] this reduction is supposed to be negligible within the area of SDEs.

The third disadvantage can be overcome by associating the different available mouse buttons with specific functionalities. During normal interaction one button triggers the user's operations. Another button is reserved for system exploration (for one-button mouses a key-button combination is used). In this way a help-function will be executed to present the relevant help-information.

3. Diamond Concepts

3.1. Policy Free vs. Simplicity

Looking at many of the currently available graphical libraries running on top of the X Window System it turns out that their usage is often difficult and error prone. The reason is that these libraries are designed to be as flexible and as powerful as possible. The "look and feel" of the user interface under design (i.e. the graphical appearance of the various objects, and the way they can be manipulated) shall not be restricted, i.e. in principal the programmer is provided with a rich set of alternatives.

As a consequence, the huge amount of supported functionality is reflected in an equally large amount of library functions. These functions tend to have dozens of parameters, each of which has to be separately specified. Interdependencies between these functions (e.g. "function A has to be called before function B") as well as side effects are often vaguely specified and constitute a major source for programming errors.

In contrast to this "policy free" philosophy, DIAMOND is designed to offer merely a selected amount of functionality. This is particularly desirable as DIAMOND is designed to preserve user interface integration within SDEs. Decreasing the variety of interaction alternatives helps the tool builders to increase the optical as well as the syntactical consistency of the various tool interfaces.

3.2. Call-back Style of Programming

According to the presented object oriented interaction model DIAMOND is itself object oriented. DIAMOND is based on a (limited) number of different graphical objects each of which is associated with a set of functions activated as soon as the corresponding events for that object are encountered. This also results in a different programming style.

In contrast to the conventional style of programming where each event has to be explicitly processed by the application (which typically leads to deeply nested if-then-else cascades) DIAMOND can process all occurring events autonomously, since each object is itself associated with the appropriate event handling functions. This helps both to decrease programme complexity and to increase readability and maintainability.

The resulting "call-back style" of programming is reflected in Figure 1.

3.3. DIAMOND Objects

DIAMOND comprises two kinds of graphical objects, i) *elementary objects* and ii) *composed objects.* Elementary objects are icons, buttons, lines, text and windows, whereby windows correspond to the simple windows as provided by the X Window System, i.e. rectangulars with an optional border.

Composed objects are constructed using elementary objects. Currently DIAMOND supports three kinds of composed objects:

(a) DIAMOND windows (Dwindows)

(b) menus and

(c) forms.

Dwindows correspond to the wide spread quasi-standard for windows. They are provided with scroll bars, scroll arrows, a close box, etc. All attributes of a Dwindow are realised by elementary objects, e.g. the close box is implemented by a tiny icon, elementary windows are used to construct scroll bars. Figure 5 contains a typical Dwindow. The major functionality of Dwindows is to provide the parent object for other objects (e.g. forms and icons).

Forms constitute a collection of editable and non-editable text objects and buttons within an elementary window. Forms are used to represent textual information and to collect input data from the user.

The major reason why forms are adopted as standard objects within DIAMOND is that particularly tools for software development require a mechanism to collect data, typically in the form of element descriptions.

Menus are composed of a sequence of non-editable text objects presenting selection alternatives. According to the proposed interaction model menus are required for two purposes: First, in the situation where an object is combined with another object resulting in more than one possible action. Secondly, in order to implement the simplified variant of the interaction model, where selecting a single object results in the display of a pop-up menu that contains all activities applicable to that object.

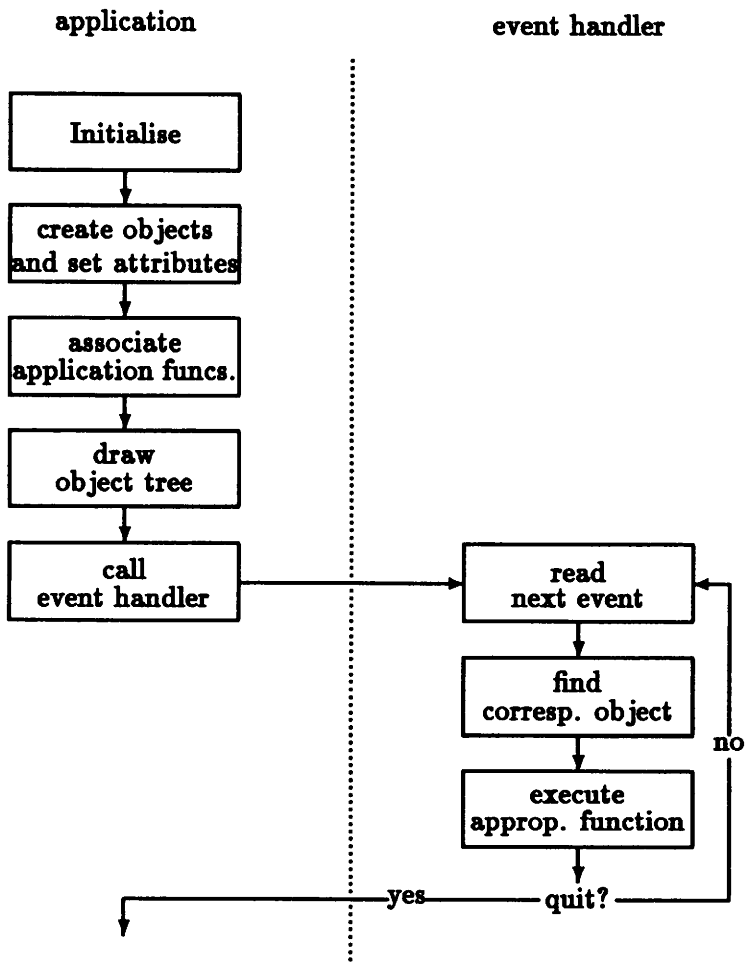


Figure 1: *Call-back Style of Programming*

3.4. Object Hierarchy

The DIAMOND object hierarchy and structure originated from the concepts applied in the Graphics Environment Manager (GEM) [Aum87a] which is well known in the PC world. The benefit of this structure is that it can be easily combined with the underlying hierarchy of the X Window System.

Each object (except the root object) has exactly one parent object and may have an arbitrary number of successors and siblings. Figure 2 exemplarily shows the principal tree structure of one parent object with three successors.

In contrast to GEM, where windows are not embedded into this general object structure, DIAMOND does not distinguish between windows and any other graphical object.

3.5. Grouping Objects

Each object is characterised, apart from its elementary type (e.g. button), by the so-called *xtype* (extended type). Generally this type is set by the application in order to easily identify particular objects. Related objects may have the same xtype which then corresponds to the "class semantic" as known in the domain of object oriented programming.

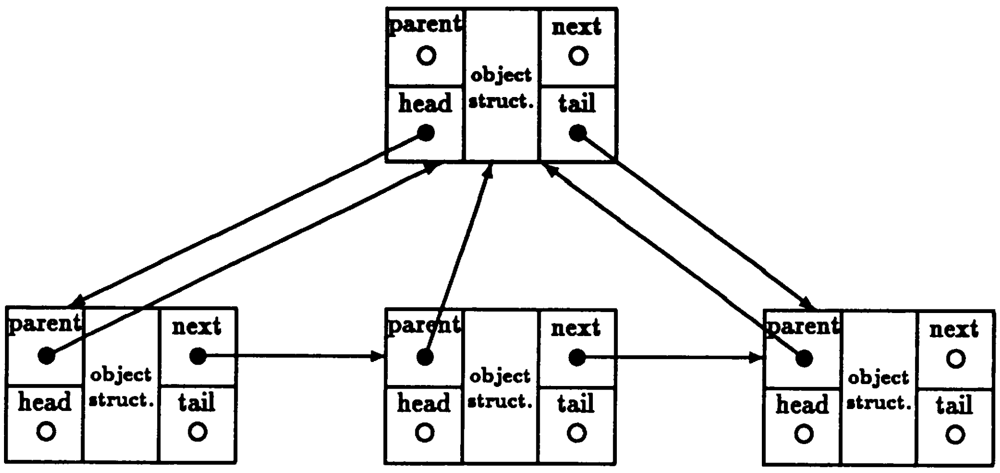


Figure 2: *DIAMOND Object Hierarchy*

3.6. Object States and Flags

Objects may have different *states* usually depending on the application context. Typical states are *SELECTED, HIDDEN,* and *DISABLED.* An object's state is reflected by its graphical representation, e.g. a selected icon is displayed inverted.

State changes occur either as a consequence of certain user actions (and can then be autonomously performed by DIAMOND, e.g. an object's state becomes *SELECTED* as soon as the object is entered with the mouse cursor) or in the light of a state change in the application (e.g. the application observes a certain condition and decides to remove an object from the screen by setting its state to *HIDDEN).*

Furthermore, each object is characterised by a set of flags describing the object's current behaviour. Typical flags are *MOVEABLE, SELECTABLE, COMBINABLE, EDITABLE.* The set of flags an object is actually associated with depends on the application context. Flags are either altered explicitly by the application or implicitly by DIAMOND.

3.7. Event and Combination Functions

DIAMOND knows a number of events that are closely related to the original X Window events. Each object is associated with particular event functions. As soon as an event for a certain object is encountered the corresponding event function is automatically invoked. For events that usually do not require intervention of the application, DIAMOND provides default functions. The standard DIAMOND reaction for entering an icon with the mouse pointer is for example to display the icon inverted. However it is possible to make any application defined function the actual event function.

In correspondence with the presented interaction model, each object may be *combined* with any other object on the screen (provided that the combination makes sense in the application context). Functions that should be invoked upon object combination can be set similarly to the event functions.

3.8. DIAMOND Look and Feel

In order to guarantee optical as well as syntactical consistency within a single user interface and among user interfaces of different tools populating the same environment, a toolkit needs guidelines apart from the numerous library functions that determine "how" the user interface has to be designed. These guidelines should

(a) describe the standard graphical objects available.

(b) determine how and in which combinations different graphical objects have to be used. Questions such as "is a menu appropriate in this situation or is it better to use a set of buttons" should ideally be answered.

(c) determine the basic principles of the interaction style, i.e. describe the underlying interaction model.

(d) determine the command syntax and the reaction of the interface. This typically relates to questions such as "is an object selected by a double or simply by a single mouse click?".

In the above enumeration, (a) specifies the "look" of the interface whereas (b) to (d) particularly deal with the "feel".

The first such guidelines have been published in connection with the Apple Macintosh [Ros85a]. (a) and (d) were mainly realised by the available graphical routines (i.e. it was nearly impossible to create other than the standard graphical objects, and the command syntax was also predetermined by the system), (b) and (c) were explicitly specified in the guidelines and explained through numerous examples.

This approach has turned out to be very useful. The Macintosh is famous for its tools that all follow the same philosophy, have the same graphical appearance and are characterised by the same "look and feel".

DIAMOND has thus in principal adopted this philosophy. The standard graphical objects (see Section 3.2.) are provided by the DIAMOND library and the command syntax is predetermined by a set of standard event functions that are associated with the graphical objects. The exact command syntax determining "how" objects are selected and combined and what the corresponding reaction of the interface will be, is described in Section 4.3.

The underlying interaction model is described in the DIAMOND guidelines [Pos90a] and supported by the specifically tailored inherent mechanisms for object combinations. The utilisation of the different graphical objects is limited due to the implications of the interaction model (e.g. a dynamic menu that merely appears upon object selection or object combination).

4. Diamond Details

4.1. Object Structure and Attributes

Each DIAMOND object is characterised by a number of attributes. Some of these attributes are common to all DIAMOND objects (e.g. parameters that specify the location of the object on the screen), other attributes are specific to a particular object type.

The general object structure is shown in a C like syntax in Figure 3.

Apart from various pointers maintaining the object structure each object is provided with a pointer to a sub-structure that contains object specific information *(spec)*. The *xspec* pointer is for free use and allows to store application specific information related to a particular object.

Furthermore, each object is associated with a drawing, a dragging, and various other event handling functions. These functions are called by the DIAMOND event handler (see Section 4.2.) if required. Finally, each object is provided with a list of all combinable objects (more precisely with a list of xtypes) and correlated combination functions.

4.2. Diamond Event Handler

The DIAMOND event handler *dnext_event()* provides the following functions:

- reading and processing events
- controlling object movement
- controlling object combinations

dnext_event() reads all pending events, searches the object tree in order to locate the appropriate DIAMOND object and executes the corresponding event function.

Furthermore, dnext_event() controls the movement of the various objects on the screen, such that they remain within their dragging areas as specified by the dragging parameters.

According to the object oriented interaction model as presented in Section 2 it is desirable that all useless objects are graphically indicated as soon as an object is selected and dragged around in order to be combined with another object. Since each object knows exactly with which type of objects it may be

```
struct object
  {
  struct
  object   *next,         /* object's next neighbour */
           *head,         /* object's first child */
           *tail,         /* object's last child */
           *parent;       /* object's parent object */
  int      type;          /* object's type */
  int      xtype;         /* for free use */
  int      flags;         /* object's flags */
  int      state;         /* object's state */
  long     *spec;         /* pointer to more detailed object structure   */
                          /* depends on object's type                    */
  long     *xspec;        /* extended specification pointer for free use */
  int      x;             /* object's coordinate in parent's window */
  int      y;             /* object's coordinate in parent's window */
  int      width;         /* object's width */
  int      height;        /* object's height */
  int      min_width;     /* object's minimum width */
  int      min_height;    /* object's minimum height */
  int      max_width;     /* object's maximum width */
  int      max_height;    /* object's maximum height */
  Pixmap   background;    /* object's background */
  Window   window;        /* object's X window */
  int      cursor_shape;  /* object's cursor shape */
  DRAGPAR  drag;          /* object's drag parameters */
  ROUTINE  draw_routine;  /* routine used when object is drawn  */
  ROUTINE  drag_routine;  /* routine used when object is dragged */
  ROUTINE  select_routine;/* routine used when object is selected */
  ROUTINE  eventaction[D_OBJC_EVENTS]; /* routines used when diamond */
                                       /* event occurs for this object */
  COMBINE  *combinables;
  };
```

Figure 3: *General Object Structure*

combined (i.e. all objects for which a combination function is specified), dnext_event() can easily determine all useless objects and autonomously put them into a *DISABLED* state until the mouse button is released.

4.3. Command Syntax

This section describes the basic principles of the DIAMOND command syntax. The process of object selection, combination, and movement is detailed.

To represent the different system states and state transitions we use a graphical notation which is taken from [Gre86a].

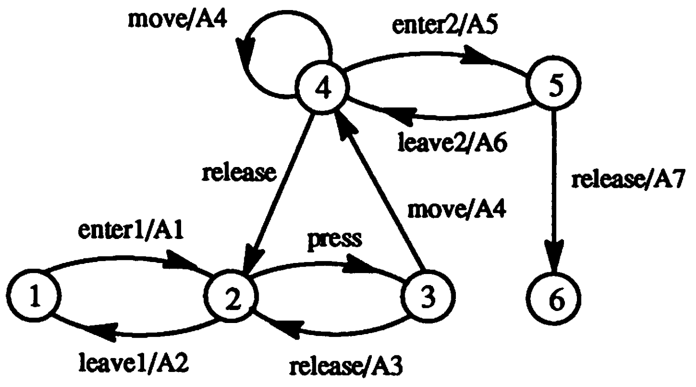
In this notation the man-machine dialogue is represented in form of a directed graph, where the nodes represent the system states (in relation to the man-machine dialogue) and the arcs describe the state transitions. Since state transitions are caused by user activities, the arcs are labeled with user actions expressed by events. Furthermore, each arc is labeled with the triggered machine reaction.

The graph comprises six nodes and is displayed in Figure 4. Events that do not cause a state change have been omitted.

Nodes (1) to (3) reflect the situation of object selection. The first object is entered with the mouse pointer resulting in an *enter1* event. The reaction of the system is to display the entered object inverted *(A1)*. If the object is now left before any mouse button was pressed, we get back to the initial state and consequently the object is reset to its original representation *(A2)*.

If on the other hand a mouse button is pressed *and* released inside an object without moving the mouse between these two events then the object has been selected which results in an application defined action *(A3)*. Typically this action is to display a dynamic pop-up menu presenting a list of actions that can be applied to the selected object.

Entering an object, pressing the mouse button and then continuing to move the mouse with the depressed mouse button will start moving the entered object such that it follows the mouse cursor *(A4, node 4)*.



A1: draw object1 in selected state
A2: draw object1 in normal state
A3: perform application defined action
A4: move object1 to current mouse cursor location
A5: draw object2 in selected state
A6: draw object2 in normal state
A7: move object1 to original position;
perform associated action

Figure 4: *DIAMOND Command Syntax*

If – during this movement – another (combinable) object is entered (event *enter2)* and the mouse button is released inside the second object (transition from node (5) to node (6)), object1 and object2 are considered to be combined which again results in a corresponding application defined action *(A7).*

If no other object is entered and the mouse button is released (transition from state (4) to (2)) nothing special happens, i.e. the first object is moved to the actual mouse cursor location.

5. A Sample Programme

Figures 5 and 6 describe a sample DIAMOND application. Two icons, one representing glasses, the other representing a document are contained in a Dwindow. If the two icons are combined a file (which is represented by the document icon) shall be displayed. The application terminates as soon as the Dwindow's close box is selected.

The programme structure conforms to the characteristic structure as sketched in Figure 1. After initialisation a Dwindow is created along with two icons. The icons are configured such that they can be selected and moved. Afterwards the application dependent functions are set. The select function for the close box is set to the DIAMOND default exit function and the combination functions for the two icons are set accordingly. Both combinations *<document><glasses>* and *<glasses><document>* are associated with the application function *combine*. Finally, all objects are displayed and control is transferred to the DIAMOND event handler.

6. Conclusions

In this paper we presented the object oriented graphics library DIAMOND that is particularly suitable for the construction of user interfaces for SDEs. The library is centered around a specifically tailored object oriented interaction model.

DIAMOND along with tools that support interactive construction of user interfaces forms a toolkit that is part of the infra-structure of MARDS [Sen89a], a computer-aided design environment for distributed real-time systems.

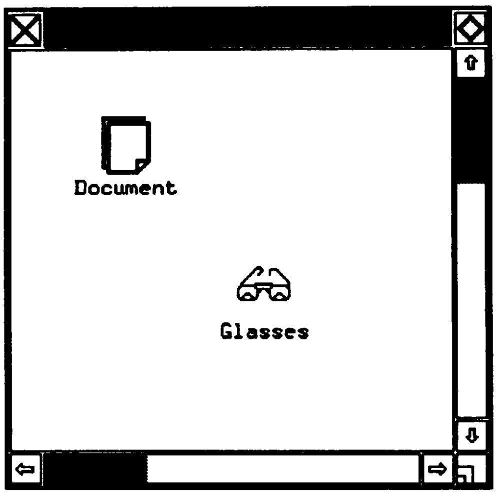


Figure 5: *User Interface of the Sample DIAMOND Application*

The major benefit of DIAMOND is that it helps to build consistent graphical interfaces and to preserve user interface integration within the overall environment.

DIAMOND is built on top of the X Window System and currently runs under *Ultrix-32 V3.0* with *X11R2*.

DIAMOND has been used at our institute for more than two years now. During this time our design environment has been populated with numerous tools all of which have been implemented using DIAMOND.

7. Acknowledgements

We wish to thank Thomas Brustbauer who helped initiating the development of DIAMOND and who significantly contributed to its implementation. We are also grateful to Werner Schütz for valuable comments on an earlier version of this paper.

References

[Aum87a] R. Aumiller, D. Luda, and G. Möllmann, "GEM-Programmierung in C," *Markt und Technik* (1987).

[Gre86a] G. Green, "A Survey of Three Dialogue Models," *ACM Transactions on Graphics* **5**(3), pp. 244-275 (July 1986).

[Jon89a] O. Jones, "Introduction to the X Window System," *Prentice-Hall Inc.*, Englewood Cliffs, New Jersey (1989).

[Mye89a] B. Myers, "User Interface Tools: Introduction and Survey," *IEEE Software* **6**(1), pp. 15-23 (Jan. 1989).

[Pos90a] G. Pospischil and R. Zainlinger, "The DIAMOND Programmer's Guide," *Research Report 8/90, Inst. f. Techn. Informatik, TU Vienna*, Vienna, Austria (Apr. 1990).

[Ros85a] C. Rose, "The Macintosh User Interface Guidelines," in *Inside Macintosh*, Addison-Wesley Publishing Company Inc. (1985).

[Sch86a] R. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics* **5**(2), pp. 79-109 (Apr. 1986).

[Sen89a] C. Senft and R. Zainlinger, "A Graphical Design Environment for Distributed Real-Time Systems," pp. 871-880 in *Proc. of the 22nd Annual Hawaii International Conference on System Sciences, Vol. II*, Hawaii (Jan. 1989).

[Zai90a] R. Zainlinger, "Building Interfaces for CASE Environments: An Object Oriented Interaction Model and its Application," pp. 65-80 in *Proc. of the IFIP International Conference on Human Factors in Information Systems Analysis and Design*, Schärding, Austria (June 1990).

```
#include "diamond.h"
#include <stdio.h>
#include "DocumentIcon"  /* includes for the      */
#include "DocumentMask"  /* icon images           */
#include "GlassesIcon"   /* similar to files      */
#include "GlassesMask"   /* created by bitmap(1X) */

#define DOCUMENT 1001    /* xtype of first icon   */
#define GLASSES  1002    /* xtype of second icon  */

main(argc,argv)
char *argv[];
{
OBJECT *root;
OBJECT *desktop,*desktop_tw;
OBJECT *icon1,*icon2;
int combine();

/* INITIALISATION */
   if((root = init_diamond("")) == (OBJECT*)0){
     fprintf(stderr, "init_diamond failed");
     exit();
   }

/* CREATE OBJECTS & SET ATTRIBUTES */
   desktop = dwin_create (root,0,0,1000,600,2,BlackPixmap,WhitePixmap,
                          WI_ALL,"SIMPLE APPLICATION");
   desktop_tw = dwin_get_text_win(desktop);

   icon1 = icon_create (desktop_tw,100,100,GLASSES_ICON,GLASSES_MASK,
                        "Glasses","","8x13");

       objc_add_flag(icon1,FL_SELECTABLE|FL_MOVABLE);
       objc_set_xtyp(icon1,GLASSES);


   icon2 = icon_create (desktop_tw,200,200,DOCUMENT_ICON,DOCUMENT_MASK,
                        "Document","","8x13");

       objc_add_flag(icon2,FL_SELECTABLE|FL_MOVABLE);
       objc_set_xtyp(icon2,DOCUMENT);

/* ASSOCIATE APPLICATION DEPENDENT FUNCTIONS */
   objc_set_slct_func(objc_fdn_xtyp(desktop,CLOSE_BOX,1),exit_diamond);
   objc_add_comb_xtyp(icon1,DOCUMENT,combine);
   objc_add_comb_xtyp(icon2,GLASSES,combine);

/* DRAW ALL OBJECTS */
   objc_draw(desktop,MAX_DEPTH);

/* CALL EVENT HANDLER */
   dnext_event();
}
```

Figure 6: *A Sample DIAMOND Application*

Why Open Systems are important to the Research Community

Jaana Porra
Johan Helsingius

Oy Penetron Ab
Espoo
Finland

ABSTRACT

Once upon a time...

The old, much-reiterated story about Ken and Dennis and their friends creating UNIX back in the labs is all too familiar to us by now. But maybe we should stop for a second to reconsider the effects of their work upon the whole computing community. As an operating system, UNIX took a bunch of good ideas, and pushed them to the maximum. But the major impact of UNIX on the community is not due to the features of the operating system per se, but to the fact that it was distributed widely both amongst the academic community and amongst commercial vendors, all sharing a couple of key concepts:

- A common platform and environment for running programs in, and a common development environment and toolset to be able to easily share both code and experience.
- A set of guidelines for designing systems based on modular, toolkit-based principles of combining components, often in an ad-hoc fashion. A fairly universal adaptation of these principles of "small is beautiful" made it possible to integrate newly-developed components from various sources into the familiar environment.

Another crucial development, closely tied to the UNIX community, was the birth of the global, interconnected networks. The traditional publications, conferences and occasional tape of shared software were supplemented and increasingly replaced by the networks as a means for communicating ideas, opinions, news and software. Communication took a much more real-time flavor, and geographical distances have become secondary.

Thus we can see the birth of a networked community of "UNIX People", spread around research institutes, universities and the industry. Yes, there have been quite a lot of other fairly similar sub-communities forming in the past, especially in the academia. They have been centered around languages (Lisp, Algol, Smalltalk, ADA...), computer systems (Dec-10/20, TOPS-20) etc., but none of these has achieved the widespread acceptance in the whole of the computing industry and academia that UNIX has gathered.

We are now going to look more closely at three key areas of the community: The universities educating the future computing professionals, the research institutions developing new technologies, and the industry developing and using products based on these technologies.

Universities

In the good old days of UNIX, every computer science department had their VAX (and a couple of old PDP-11:s), running BSD UNIX (or even plain old V7). Of course DEC didn't support UNIX, and Berkeley didn't (officially) support BSD, so everybody had to provide their own support. And this lead to the emergence of local UNIX gurus, who knew the ins and outs of (their local) UNIX kernel and configuration, and patched device drivers on the fly.

Of course, these people were often some of the brightest and most promising young people at the university, who instead of doing original research, spent their days (and nights) making clever hacks to the operating system or essential utilities such as emacs and rogue. An very easily UNIX hacking became the main activity, instead of being a way to greater productivity.

And when commercial vendors turned towards UNIX System V, and then to the elusive "open systems", the people responsible for computer procurements at the universities discovered that the computing people did not use UNIX. No, they used VAXen and Suns to run BSD. And they couldn't switch. Because BSD was "right" and System V was "wrong". Because the UNIX gurus knew BSD and VAX hardware. And because the systems developed at universities required hacked kernels, and assumed awful things about the OS and the hardware.

But often the procurement people won, because of very convincing financial reasons, and suddenly there was a wealth of systems from various vendors, running different flavors of UNIX. And the software wizards had to learn to program to an interface specification, not to an implementation (that they could hack at will; "Use the source, Luke"). And they discovered, to their dismay, that a great deal of the public software floating around didn't run on half of the machines because of lack of portability.

Research institutions

How about the orginators of the technology? The research facilities at the major universities, and the research institutes and laboratories of big companies? Invariably we seem to come across the not-invented-here syndrome. New, original and interesting development work is based on local, esoteric and proprietary technologies and tools. Interest in research done by other people is mainly academic, and attempts to re-use wheels invented by others is minimal, even when freely available standard solutions would provide superior functionality. A typical attitude is "well, I don't need all the bells and whistles of Y, and besides, I don't have time to learn the damned interface anyway...".

Often the research and development work is very pragmatical, and is driven by local, specialized needs. Thus no effort is put into making the results more general or widely usable.

All this is not necessarily a bad thing. It is important that researchers are allowed to go on innovating and developing new technologies unencumbered by standards, conventions and unnecessary "product-oriented" thinking. But the current situation generates a number of problems.

The dilemma

If we look at the research and development work done at universities, we quickly notice that the results are not utilized very effectively. Even if most projects don't attempt to create a commercial product, the developers would in most cases like to see the results of their work gaining a wide distribution and being put into use as widely as possible. To this end, software is often posted on the net or distributed as public domain tapes. But in most cases the software ends up being used little or not at all. Why is this? Why does academic software have such a bad reputation among the industry? Why do most academic research projects involving software development end up producing a couple of learned treatises and some expertise, moving over to the industry in the minds of those graduating or switching over to the "commercial" world?

We have to keep in mind that the academia and research community has a very tangible influence on the industry, on the vendors and the end-user organizations. One of the overwhelming problems in the industry is the lack of people who know how to do open, modular systems. Not surprising, as most universities don't teach their students to do this. They might have courses on good design, portability, and modularity, but do they live the way they preach? And why not? Here are some of the reasons:

- Researchers and educators don't perceive openness as an important issue.
- Universities are stuffed with donated or cheap proprietary system environments, either supported by major vendors such as IBM and DEC, or by the major national supplier (ICL, Bull, Siemens, Nokia...). The guru cult quickly leads to competition pressure to utilize special features (preferably undocumented ones). An even worse alternative is using locally-developed, esoteric systems as a base.
- Portability and generality are seen to encourage commercial use, and the commercial world is often perceived among the academia as the evil empire, to be resisted to the bitter end.

So what?

The nature of research has changed. Things are not developed in vacuum, for internal use only. Development is increasingly becoming a distributed, co-operative effort, producing results that are immediately and widely usable. Thus the researchers get immediate feedback and input on the applicability of their ideas, and are able to draw on the work of others more easily.

This development also helps erase the wall between the research and academia on one side and the commercial vendors and users on the other. Researchers rely increasingly on standard products, and research results can be applied much more quickly and uniformly into new products.

To facilitate this, universities have to realize the value of abiding to open standard specifications and interfaces, and encouraging portability, modularity and generality instead of fostering closed sub-cultures of gurus and wizards. Research and academia also have to put enough resources into monitoring and participating in standards efforts, such as ISO, IEEE and even X/Open.

Why Factionalism?

Pamela Gray

Marosi
UK

ABSTRACT

Faction:- "A group or clique within a larger group, party, government, organisation or the like; party strife and intrigue; dissension."

The UNIX market is growing fast on all fronts, both technical and geographical. Nevertheless, the press still seems to delight in reporting the so-called "UNIX Wars", and to concentrate on the differences in the strategies of product suppliers and the flavours of their UNIX implementations, rather than the similarities that exist amongst them.

Why is there factionalism in the UNIX market today? We will address this by breaking the question into three parts:

1. Is factionalism today stronger than it was in the past?
2. Are there real reasons for concern?
3. Are there any positives in the present situation?

We will address each of these questions in turn.

1. Is factionalism today stronger than it was in the past?

Five years ago, it was difficult to identify the many different versions of UNIX that were available from the various vendors of both software and hardware, or to understand their relationship one to another. From the procurement agency's point of view, it was therefore hard to define rules for purchasing that would guarantee a reasonably high level of software portability. The standards were in early stages of definition, and it was not clear if, or which, vendors would ultimately support them.

Most computer suppliers were only just beginning to grapple with the issues of supplying "open systems" that would compete in the marketplace with their own proprietary technologies, and none knew how to sell against competitors on a platform of compatibility. Past experiences had led them to believe that product differentiation had to be made on the basis of technological differences, so that even those suppliers who entered the UNIX market early often believed that their own UNIX implementations had to be significantly different from any other suppliers to enable them to claim superiority. These differences were called "the manufacturers' added-value", and led to innumerable variations from the UNIX product originally supplied to them by AT&T.

A notable exception to this thinking, at least in the past, was SUN Microsystems, which, although consistently enhancing UNIX for its own purposes, has always been willing to supply the enhancements back to the marketplace for incorporation into standard product.

Today, this situation seems to have changed dramatically. Rather than focusing on standards for the operating system itself, international standards are emerging, through the POSIX work, for the operating system interface definition. These are supported in principle by the whole of the computer industry. (This may be taken as an indication of uniformity in support of standards, or as a statement about the weakness of the standards definition.)

Three distinct and important versions of UNIX existed in the marketplace until recently, coming from major software suppliers; AT&T (UNIX System V); Berkely Software Distribution (BSD); Microsoft via its licensee, the Santa Cruz Operation (SCO XENIX/UNIX). Each of these has had a devoted band of followers in its OEMs and other licensees – a situation that could have been described as a minor example of factionalism. While these variants are now converging rapidly into a single, standard UNIX version (UNIX System V.4), many hardware manufacturers are still adding their proprietary "enhancements".

In spite of this, there is little evidence that such variations have been a cause of "strife and dissension", although they have undoubtedly caused a certain amount of confusion in the user base. Nor is there much cause to believe that they are a major obstacle to application software portability – and subsequent people and skill portability – the main issues of concern in the user base. However, each variation that exists gives rise to additional costs for porting and support for suppliers of applications software, and these are ultimately passed on to the users, directly or indirectly. If these increased costs were better identified and the reasons for them understood by the purchasers, pressure would surely be brought to bear to reduce the unnecessary variants i.e. those that are there for the sake of being different, rather than for good technological reasons.

The UNIX situation changed dramatically with the formation of the Open Software Foundation in 1988. With founder members that included Digital Equipment Corporation, Hewlett Packard and IBM, and a declared strategy of developing software technologies competetive to AT&T's, factionalism definitely reared its head. Here clearly was a "group (the OSF members) within a larger group (the UNIX licensees)" that was likely to lead to "strife and dissension" within the computer industry.

Subsequently, this has proved to be the case. The OSF/UNIX International divide has provided more stories and analyses within the computer press than any other issue in the history of the computer industry. And as competetive technologies for the operating system and its extensions to user interface, distributed applications, multi-media support and so on, emerge from these two organisations (and others), the discussions, and confusion in the user base, can be expected to continue.

So factionalism does exist in the UNIX marketplace today, caused principally, but not entirely through the formation of the OSF, and is certainly more identifiable and focussed than it has been in the past.

2. Are there real reasons for concern?

By far the biggest area of concern today that results from significant differences in the standard applications software environments existing within machines is that of interoperability – the ability of applications to be run in a distributed fashion over a network. For the applications of the future, likely to be based on client-server architectures, and running on networks of multiple vendor computers, applications will be partitioned into modules that will run on whatever machine is appropriate at the time, with the user not needing to know where this might be. Applications running on different machines will then need to communicate effectively.

For interoperability to be accomplished, standards need to be established for the applications environment, for data representation, for communications, and for the combination of technologies at the applications layers of the communications protocols. True distributed computing will be very difficult to accomplish while there are many variants of applications environment to deal with. Although it is probably not realistic to advocate one and only one overall – there is, after all, IBM's own SAA environment to deal with (itself an uneasy union of incompatible architectures) – it does not seem unreasonable to hope for only one in the UNIX/Open Systems market, and to advocate a single standard for distributed computing overall.

A second issue concerns the user interface. Although it is true that a user familiar with one graphical user interface of the Apple Macintosh type can easily learn to use another, the fact is that most users do not want to have to stop to think about such things. The reason, after all, that they have computers is to use them to increase productivity within their own businesses, and anything which takes away from this is therefore to be avoided. Likewise, no software developer wishes to port and support software on any more than the absolute minimum number of graphical user interfaces, particularly since it is clear that, with multi-media advances, there are many more interesting and potentially useful ways to deploy scarce development and support resources.

So there are real reasons for concern as both the Open Software Foundation and UNIX International continue to move to develop competing technologies in a number of vital areas.

3. Are there any positives in the present situation?

Compared to the state in the past, at least today it can be said that the factions in the UNIX market are clearly identified, the market is fairly orderly and the technical issues are becoming clearer. In addition, there are more issues on which the factions agree than there are on which they disagree – both groups support the POSIX and X/Open XPG standards, for example.

The debate is taking place fairly openly. Both the OSF and UI publish detailed documentation on their current and future development plans, and allow the components of the market to make contributions to them. This means that users who wish to can make their views known, contributing to and influencing developments. both technical and political, if they so wish. Many are now doing this in various forums around the world.

Competition is almost always thought to spur people and organisations into better performances and products than they would otherwise produce. In this sense the present state would seem to have the potential for good.

The current "open" factionalism, often taken to be a symptom of the vendors' desire to manipulate the market, may well end up by the market controlling the vendors. Exposure of the issues constantly and publicly means that slowly but surely, users are understanding what this all means to them. Since the open systems market is driven primarily by the users needs to rationalise their information technology resources, in the end, the users will decide whether factionalism in the UNIX marketplace can be supported, or not.

UNIX International's

System V Roadmap

UNIX System V Strategy for the "90s

Andrew Schuelke

UNIX International – Europe
Ave de Beaulieu 25
1160 Brussels, Belgium
a.schuelke@uieu.ui.org

ABSTRACT

One of UNIX International's primary tasks is to define requirements for future releases of UNIX System V. The Roadmap is UNIX International's top-level strategic plan and contains UNIX International's requirements for approximately the next five years of System V releases.

The first half of this paper explains the significance of the Roadmap's existence, the process by which the Roadmap was created, and the selection criteria used to select System V release features. The second half presents the Roadmap System V requirements, defining four major groups of functionality centered around the following themes: Unification, Enhanced Security, Multiprocessing, and Network Computing.

1. Introduction

The Roadmap represents the top level, approximately, five year strategic plan for UNIX System V and is the foundation requirements document for the development of future System V releases. It defines several general areas of enhancements by grouping requirements and functionality based on market priorities, technical maturity and functional interdependencies.

The Roadmap is a result of the UNIX International open decision process which ensured that this strategic plan for UNIX System V reflects the industry's needs and desires. As the foundation document in the UNIX International process, the Roadmap will be used as input to UNIX International work groups and UNIX System Laboratory (USL) where Roadmap requirements will be translated to detailed requirements, specifications, designs and finally the product.

This paper presents the **1989** Roadmap, UNIX International's first Roadmap. The Roadmap is updated annually with the 1990 version to be released late 1990 or early 1991.

This paper presents both the Roadmap's development process and its content. Section 2 begins by explaining the significance of the Roadmap's existence. Section 3 then describes the process by which, and the environment in which, the Roadmap was written so as to assure the reader that the Roadmap is more than "just another spec."

Section 4 contains the heart of the feature selection criteria, that is, the general requirements, constraints, engineering tradeoffs, and thought processes used by the UNIX International Steering Committee to discuss and formulate the Roadmap.

Section 5 contains a summary of the Roadmap itself. This section begins with the Roadmap's mission statement, goals, and strategies for System V followed by a description of the Roadmap's requirements for each upcoming System V release.

2. Significance of the Roadmap

"Open Systems" has in the past two years become a major movement and battle cry with essentially all major hardware vendors jumping on the bandwagon. This awakening is due primarily to end-users who have realized the benefits of a long term strategy based on open systems and who are now demanding open system solutions. However, many organizations do not follow through on their newly stated commitment to open systems. Some vendors are promising but not delivering open systems products and various industry groups are promising and not delivering open system decision processes. The proof of the pudding, as usual, must be in the results.

The UNIX System V Roadmap *exists* and defines the next several years of progress for the open system *product*, UNIX System V. In addition, the Roadmap represents a true open system decision *process*, being the first time that such a large and diverse body of market participants have worked together in a product planning capacity. As will be described later, the Roadmap represents input from X/OPEN and its sources, UNIX International members, and other UNIX data sources.

Although this sounds like typical marketing hype, this credible message has real implications for all product- and technology- planners. For example, independent Software Vendors (ISVs), knowing System V's future, can reduce functionality overlap between applications and the operating system as they rely on increased support from the System V environment and can concentrate more on their specific value-added applications.

Similarly, system integrators and hardware vendors can better plan when to bring which system products to market, adjusting their strategies to provide value-added more efficiently and where most needed.

End-users too can better plan their system and application procurements. Overall, the Roadmap gives the various industry players the information necessary to efficiently plan their future and the confidence to proceed in the open systems direction.

The bottom line, of course, is that these increased efficiencies in planning translate to cost savings and decreased corporate risk. Open systems, in general, offer savings to the end-user as the end-user is given more hardware/software choices at competitive prices.

To further understand the significance of the Roadmap's existence, the reader need only be reminded of the situation two or three years ago when industry players had little or no knowledge of, or input to, AT&T's plan for the upcoming System V release, let alone several releases ahead. An even more timely comparison can be made with the world of proprietary operating systems whose vendors have not and will not publish multi-year product strategies.

3. The Roadmap Process

The UNIX System V Roadmap was developed over nine months in 1989 as the primary responsibility of the UNIX International Steering Committee. The Steering Committee consists of delegates from the UNIX International membership supported by various UNIX International work groups, special interest groups, and UNIX International staff.

The Steering Committee members are in senior positions in their respective companies and are well versed in the industry, UNIX technology, market requirements, the software development process, and associated business issues. The Steering Committee members take their task seriously since a Roadmap that is not implementable or that does not meet market demands will be useless and will undermine the credibility of System V in the market.

The Steering Committee received and analysed input for the Roadmap from many sources in addition to that from UNIX International members. In particular, X/Open's Xtra program provided a significant source of end-user and ISV requirements. In addition, USL provided necessary input throughout the process to ensure that the Roadmap feature and timeframe requirements could be implemented.

The Roadmap represents just the beginning to the UNIX International process. As defined in [Com89a] feature areas for upcoming releases as defined in the Roadmap are assigned to UNIX International work groups to define detailed requirements and specifications. Several hundred senior computer scientists participate in the following UNIX International Work Groups and Special Interest Groups:

Distributed Computing	Operation, Administration, and Maintenance
Documentation	OSI Networking
File System	Performance Management
Internationalization	System Interfaces
Licensing and Conformance	Transaction Processing
Multiprocessing	User Interfaces

USL steadily assumes more responsibility for the feature set as the development process moves toward detailed specification, design, and implementation.

The Roadmap is a living document and the Roadmap process requires that the Steering Committee update the Roadmap each year. The Steering Committee's goal is to maintain continuity from one year's Roadmap to the next yet be flexible and responsive to changes in market requirements and/or technology that would necessitate Roadmap changes.

4. Selection Criteria

Defining the Roadmap is essentially a selection and prioritization process. Given the constraints presented in this section, the Steering Committee had to select from their ideal and overly-long "wish list" those features which would most effectively further System V in the marketplace.

The Steering Committee's first step was to define a framework of basic principals within which to prioritize and evaluate possible features. These principals are:

Compatibility	Applications running on a previous version of UNIX System V, XENIX, SunOS, or Berkeley systems must run on UNIX System V Release 4 (SVR4) and future releases.
Portability	Applications must be binary compatible across all machines within the same architecture (e.g. 386/486). Applications must be source level portable across different architectures running SVR4 and future releases.
Scalability	System V must continue to be a viable system across all classes of machines.
Interoperability	SVR4 based machines must be able to communicate in heterogeneous environments using industry-standard protocols.
Standards compliance	SVR4 must fully support the creation of and must comply with relevant standards, particularly X/Open's Common Applications Environment.
Open process and product	System V must continue to be an open operating system product and UNIX International's process an open process.

Secondly, the Steering Committee, in following a top-down approach, spent a significant amount of time defining the Roadmap's Mission Statement, Goals, and Strategies. These are presented in Section 5.

The Steering Committee also had the following constraints and tradeoffs to consider:

4.1. Technology Adoption Criteria

The UNIX industry has always considered the UNIX system as a core operating system which was enhanced by a wide variety of value-added features from various outside sources. Keeping this heritage in mind, the Steering Committee needed to consider when technology should be adopted for inclusion into System V versus when the market requirement should be left for outside sources to provide.

One possible approach is that USL incorporate new computer technology into the UNIX System V base without waiting for hardware vendors to implement the technology as value-added. The other approach is that new features and technology should only be added into the System V base once the technology had been proven to be a stable and well understood technology and had been accepted by the market as a generally useful feature.

The early adoption approach provides features earlier but at a higher risk of unacceptable system stability. The "only when proven" approach provides for a stable and well understood base but leaves System V open to criticism of "that feature is too late!"

The Steering Committee adopted the latter approach, recognizing that a base product with such wide dissemination must be absolutely stable and should contain only mature and proven technology. System V releases cannot be subject to later retraction due to premature and defective designs. With this approach, the Steering Committee also acknowledges and encourages the critical role of hardware vendors, ISVs, and system integrators in adding value to the base. Using this wide variety of value-added, the market also provides an automatic selection process which highlights the best technology.

4.2. Feature Grouping

A significant and very practical consideration was how to group the features in a particular release. For example, should a full feature set for a particular market (e.g. multiprocessing) be included in a System V release or should the features be phased in over several releases? A phased approach would provide initial pieces of more feature groups earlier but total functionality for each group would arrive later.

The Steering Committee selected an approach of defining *groups* of functionality which can easily be considered and implemented as releases. USL defines the *specific format* of their releases, updates, add-on modules, etc. For the sake of this paper, the Steering Committee-defined groupings are referred to as "releases."

From a software design approach, implementing all parts of a feature group at once also makes more sense since the entire feature group (e.g. multiprocessing) can be designed and implemented as a coherent piece avoiding the dangers of design philosophies changing between phases. As presented in the next section, the themes for the four releases are Unification, Enhanced Security, Multiprocessing, and Network Computing.

However, some technologies are more conducive to a phased implementation (e.g. internationalization). Each major release may contain new features for such a particular phased-in technology group. As mentioned above, USL may package their deliverables in various ways. For example, USL may provide functionality modules which can be be added to a system without needing to update the kernel.

The Steering Committee also discussed how frequently major releases should be scheduled. Frequent releases would allow smaller sets of features to be implemented whereas larger feature sets would require longer time periods between releases. In general, the Steering Committee set a guideline that system releases should come no more frequently than every 12 to 18 months but at least every 24 months. Releases that appear too frequently require too much effort from the hardware vendor and frequent upgrades can confuse the end-user.

4.3. Dependencies

The following are dependencies which the Steering Committee also considered in their selection and prioritization process:

Technology	Certain features are dependent on the availability of stable technology or the completion of other features. For example, the Network Computing Plus release will support loosely-couple multiprocessor architectures which is clearly dependent on the tightly-coupled features of the Multiprocessing Plus release.
Standards	Standards are moving targets as they are being defined and implementation of associated features must often wait until the standards definition is stable. For example, real-time features are required for the Network Computing release because P1003.4 standards work is expected to be stable by this time.
Market Acceptance	At times, a feature is dependent on what the market endorses. Although OPEN LOOK is included with the System V Release 4 product to meet the graphical user interface requirement, the Multiprocessing Plus release will include GUI enhancements which will reflect the GUI solution endorsed by the market.

5. The Roadmap

5.1. Roadmap Statement of Direction

As mentioned earlier, the Steering Committee used a top-down approach in its selection process. Toward this end, they spent a large proportion of their time defining a Mission Statement, a Long Range Goal, and a Strategic Direction.

The following Roadmap Mission Statement may seem somewhat obvious but such a statement is required to lay a solid foundation.

> **Mission**: Define forward compatible evolution of the UNIX System V software platform to further broaden the open systems market by making computing accessible to more users and applications while maximizing the System V market share.

Likewise, the following Long Range Goal may seem obvious but needs to be stated. Note the emphasis on interoperability in heterogeneous environments:

> **Long Range Goal**: Produce a universal software platform that supports and facilitates coexistence of diverse, heterogeneous computer systems using high speed communications in an open systems environment. System V must provide a consistent system architecture across

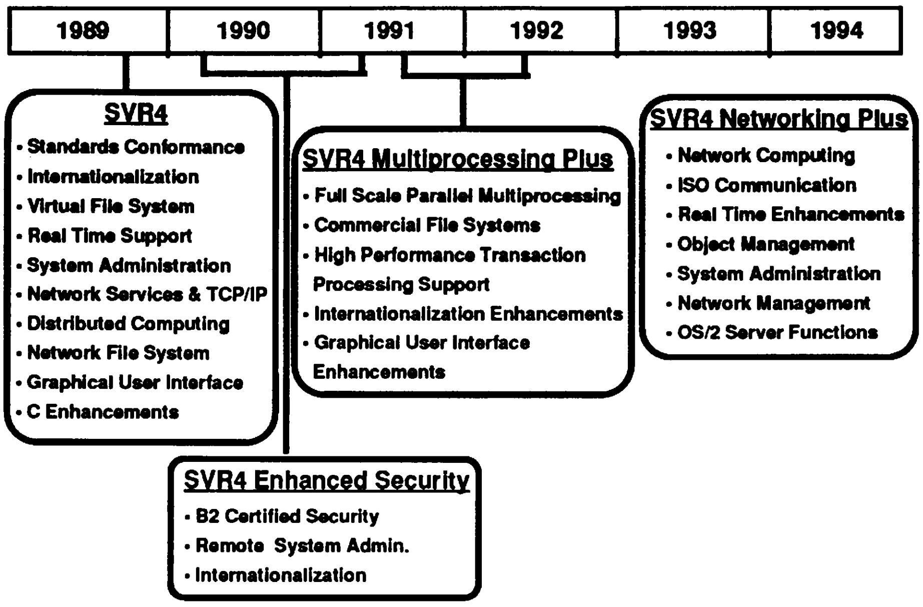


Figure 1: *System V Release Feature Timeline*

all environments, small and large, and support interoperability with other prominent proprietary systems.

Less generic, however, is the Roadmap's Strategic Direction which provides a specific direction which future releases should follow:

> **Strategic Direction**: Establish UNIX System V as a pervasive system in the network computing marketplace. This requires UNIX System V to capture a significant share of network servers and high-end desktops for which DOS is inadequate.

Network Computing is a major developing trend which, including client/server and cooperative computing, involve the connection of numerous desktop machines to department and corporate-wide computer systems. Such a configuration gives organizations more flexibility when expanding or changing computing resources; including permitting effective use of specialized systems for specific applications (e.g. database machines), while maintaining compatibility with existing software. But more than just distributively connected machines, network computing may lead to significantly different applications and a further broadening of the range of people who use computers.

5.2. Roadmap Release Overview

Figure 1 presents the 1989-defined Roadmap for UNIX System V in the "90s. This timeline presents the four releases, their themes, and their features. The dates are realistic estimates which may, of course, be adjusted based on further definition by UNIX International Work Groups and USL development staff.

Figure 2 presents a different view of the System V release feature progression. This tabular representation shows clearly which features are introduced as major pieces of functionality and which ones are phased in over several releases. To aid in orientation, the reader should refer back to this table while reading the balance of this paper.

5.3. System V Release 4

System V Release 4's primary theme is Unification as it represents the unification of the three major UNIX variants; System V, Berkeley BSD, and XENIX. Note that System V Release 4 (SVR4) was not defined by the Steering Committee since Release 4 development was well under way when UNIX International was formed. However, Release 4 is included in the Roadmap since Release 4 serves as the foundation release upon which the following releases are based.

The following lists SVR4 features. A more complete description of SVR features can be found in [Int89a].

	Pre-SVR4 UNIX Sys. History	System V Release 4	SVR4 Enhanced Security	SVR4 Multiprocessing Plus	SVR4 Network Computing Plus
International-ization	8-bit char. Support Nat. customs	Multi. Nat Language Support	Added multi-language and GUI support	Enhanced Mulit-byte Char. Support	
System Admin.	Basic Ops., Admin., and Maint.	Enhanced Local OA&M	Remote OA&M		Distrib. Resource Mgmt./Full Funct. Network Mgmt.
Multi-processing	Vendor value-added			MP Kernel Parallel Processing API	Distributed Multipro.
Network Computing	Network Architecture RFS, STREAMS	TCP/IP, NFS RPC			Full Distributed Appl. Support OSI Support
Security	Business/Mkt. Security	Security Ready	Certified B2 Security		
User Inter-face Funct.	Character based	Graphics based X Windows		Common API for Multiple GUIs	Object Management
Core Op. Sys.	Doc. & Prog. Tools	VM, VFS Real Time Capability		File Mgmt. & Trans. Process. Enhancements	Advanced Real Time

Standards Conformance — XPG3, POSIX 1003.1 — XPG4, POSIX 1003.2 — XPGn, POSIX 1003...

Figure 2: *System V Release Description by Feature Set*

5.3.1. Application Binary Interface (ABI)

ABIs provide ISVs the opportunity to write shrink-wrapped software which will run on any machine with a common architecture providing the required portability and upward compatibility. ABIs are now available for WE32000, SPARC, Intel 386/486, MC68000, and MC88000 architectures with ABIs for MIPS and i860 under development.

5.3.2. Conformance with Industry Standards

Release 4 conforms to: ANSI C, IEEE P1003.1, FIPS 151-1, and XPG3. As mentioned above, SVR4 resolves the major differences between Berkeley BSD based versions, XENIX, and System V.

5.3.3. Network Services

Expanding on the UNIX System V Networking Architecture introduced in System V Release 3.0, SVR4 provides further enhancements toward the Roadmap's network computing goal by supporting the development of distributed applications and the introduction of OSI networking.

SVR4 enhanced networking offers standard networking services which work independently of the underlying communications protocols. Programmers can, therefore, develop applications for distributed computing environments without becoming dependent on a particular family of protocols. Similarly, OSI can be introduced at the lower layers without affecting existing services.

More specifically, SVR4 Network Services introduces Network File System (NFS), Remote Procedure Call (RPC), External Data Representation (XDR), BSD sockets interface, and a streams implementation of TCP/IP.

5.3.4. Internationalization

SVR4 further develops the model introduced in Release 3.1. Applications can now choose appropriate national customs (e.g. dates) from sets of national data and can switch between locales by simply changing an environment variable. It supports and exceeds the internationalization capabilities defined in ANSI X3J11 C Language standard and fully supports all European internationalization and localization needs.

5.3.5. Virtual File System

SVR4 now provides a Virtual File System architecture with which multiple file systems are simultaneously supported. It is based on the File-System Switch (FSS) from SVR3 and the vnodes architecture first implemented in SunOS systems. Supported file systems are UFS, RFS, NFS, PROC, FIFOFS (for pipes), and SPECFS, a common-code interface to all device files. VFS allows system providers to provide additional file system types which furthers System V's interoperability goals and provides a strong base for future commercial file system enhancement work.

5.3.6. Real Time Support

New features include multiple schedulers, user-controlled process scheduling, high-resolution timers, and enhanced memory locking. These features are critical in applications ranging from industrial automation through database and transaction processing.

5.3.7. Operations, Administration and Maintenance

The following features have been enhanced or added; software installation and configuration management, backup and restore, a full and consistent OA&M interface, and message management.

5.3.8. C Language Enhancements

The C Software Development System now contains support for ANSI X3J11 C language standard, dynamic linking, and dynamic tables. It also supports a new object file format, Extensible Linking Format (ELF), which provides enhanced support for both C and other programming languages, particularly C++.

5.3.9. Graphical User Interface

SVR4 includes the graphical user interface, OPEN LOOK, and graphical user interface software, X11 and X11/NeWS. The SVR4 Multiprocessing Plus section discusses the Roadmap's GUI strategy in more detail.

5.4. System V Release 4 Enhanced Security

SVR4 Enhanced Security will provide a highly secure operating environment for those users requiring high levels of trusted operation. In addition, SVR4 Enhanced Security will contain new Internationalization support and will further the network computing strategy by providing a Remote Operations, Administration, and Maintenance (ROA&M) system.

5.4.1. Security Requirements

The Roadmap requires that SVR4 Enhanced Security meet the B2 level of trustedness with B3 features as defined in [Def85a] (TCSEC) and that it must pass the U.S. National Computer Security Center's B2 evaluation.

Note that SVR4 Enhanced Security is designed to satisfy the security requirements for those environments requiring a very high level of trust. The security provided by standard SVR4 is already quite robust and sufficient for 90% of standard environments, meeting the TCSEC C1 security requirements. Using modularized packaging, SVR4 Enhanced Security can also be configured as a C2 system. UNIX International's white paper [Int90a] describes and places into better context the security features of standard SVR4 and SVR4 Enhanced Security and which one should be used for a given operating environment.

The TCSEC defines B2 level requirements as:

Identification and Authentication	Features to identify and authenticate system users before allowing them access to the system.
Audit Trail	A comprehensive audit trail system which records all security relevant operations, providing administrators with a flexible tool to monitor user actions (also required at the C2 level).
Discretionary Access Control	A mechanism which allows a user to control access to their data using an Access Control List, a more finely grained extension of the user/group/other UNIX protection bits.

Mandatory Access Control and Security Labels	A feature set that labels each data file and process with a security label signifying its security sensitivity (e.g. top secret). Users and administrators are also assigned clearances which determine the data and processes to which they have access.
Trusted Facility Management	System Administrators are no longer fully trusted and their administration tasks must be divided up according to the principal of least privilege. Toward this end, the superuser privilege can no longer exist in its full privilege form.
Trusted Path	The system must provide a trusted path which guarantees to the user that they are communicating with the system itself and not with a trojan horse or other malicious program.
Other Assurances	The B2 level of trustedness requires extensive assurances that the system was designed properly and is executing properly. Toward this end, the software engineering cycle is particularly rigorous, design and end-user documentation must meet stringent requirements, and special analysis (e.g. covert channel analysis) must be done. In addition, the kernel must be significantly reorganized to make it modular, providing well-defined interfaces between major kernel components (e.g. memory management). Note that much of this modular kernel work has already been incorporated into SVR4.

5.4.2. Remote Operations, Administration, and Maintenance

With the steady move towards network computing, an administrator's task becomes more difficult as computers are distributed throughout an organization. SVR4 OA&M capabilities must be enhanced, allowing administrators to monitor, log, and track the status of both local and remote the systems and applications and to track the causes of system or application problems.

5.4.3. Enhanced Internationalization Support

Internationalization will continue to be enhanced to support all major written languages by providing hooks for supporting multi-national languages and GUIs, specific localizations of system messages and administrative functions, and multi-language documentation.

5.5. System V Release 4 Multiprocessing Plus

SVR4 Multiprocessing Plus is the first release fully defined by the Steering Committee in its definition of the Roadmap since SVR4 and SVR4 Enhanced Security were already under development when UNIX International was formed.

Multiprocessing Plus will meet the market requirements for an off-the-shelf, symmetric, shared-memory, fully-parallel, multiprocessing system. In addition, SVR4 Multiprocessing Plus will contain a consolidated GUI, Commercial File System and Transaction Processing Enablers, and further Internationalization enhancements.

5.5.1. Multiprocessing

Multiprocessing is a cost effective means to deliver better price/performance hardware as a result of improvements in microprocessor technology. Several System V vendors already offer multiprocessor systems, but in incompatible ways. The Roadmap defines a standard multiprocessor system with the following requirements:

> The choice of a specific MP implementation was made from the continuum of architectures from single processors to fully distributed message passing systems. The choice, a symmetrical shared memory architecture, is similar to models used by existing System V MP systems but must be integrated with the latest SVR4 functionality and implement a fully parallel kernel, I/O, and compiler functions. Areas of enhancement will include parallel

> execution of kernel, utilities, and libraries, as well as extension of the operating environment to support parallel execution of applications.
>
> The MP design should leverage existing MP work to provide an advanced system with negligible performance implications for non-MP applications. Operations, systems administration, and maintenance aspects of MP will also be addressed. While the implementation will focus primarily on symmetrical shared memory architectures, it will not preclude ports to non-uniform memory architectures, and will require only a modest amount of effort for these ports.

An additional requirement for the MP work is to create a multiprocessing Applications Programming Interface (API) specification. Application development to fully exploit a multiprocessor's performance capability is already a difficult task. The MP-API will allow ISVs to concentrate on developing only one version of their MP-capable application which will run on all architectures running SVR4 Multiprocessing Plus.

5.5.2. Graphical User Interface

The ISV community has very vocally called upon technology providers such as The Open Software Foundation and UNIX International to provide an API to support multiple "look and feel" services so as to ensure that ISV-developed software is portable between systems.

Toward this end, the Roadmap requires the following:

> It is the position of UNIX International to ensure System V will effectively support X windows-based GUIs. This means System V is a stable platform providing application investment protection for ISVs and end-users by ensuring GUI compatibility, portability, and heterogeneous interoperability exists for X windows-based GUIs which meet appropriate technical and market criteria. These criteria are: 1) Conformance to the X architecture; 2) Open availability of the GUI specifications to the marketplace; 3) Commercial availability of the product; 4) Absence of legal encumbrances; and 5) Existence on SVR4 and the ability to be forward compatible with future GUI requirements for the 1990s. The Roadmap requires System V to ensure that applications based on such popular GUIs will operate without modifications on all SVR4 systems.
>
> The requirements for GUI enhancements are for a specification first; then an implementation of a single, portable applications programming interface that addresses the major market-accepted GUIs.

Providing an accepted GUI to the market is critical for UNIX System V in its goal to provide the best operating system platform for the desktop.

5.5.3. Commercial File System and Transaction Processing Enablers

To further penetrate database and transaction processing markets a number of requirements have been identified. These include larger maximum sizes for system resources such as files and file systems, asynchronous I/O, direct I/O, and fast recovery of file system consistency at reboot.

Additional requirements for transaction processing are light-weight thread support for more efficient synchronization of processes, priority scheduling across multiple process scheduling policies, and error containment of error situations to increase system availability.

5.5.4. Internationalization

The Roadmap requires that SVR4 Multiprocessing Plus include an internationalized platform for both graphics and the GUI toolkit such that application binaries will operate in multiple locales without modification, and the graphics platform and operating system will support simultaneous use of several national languages.

In addition, the system will support multi-byte character sets primarily to address Asian languages, regular expression support and continued conversion of the System V command set to exploit the underlying internationalization capability.

5.6. SVR4 Network Computing Plus

SVR4 Network Computing Plus provides a powerful network computing base including distributed multiprocessing and network management features. Additional features of this release are OSI communications, object management features, real-time enhancements, and system administration enhancements.

5.6.1. Network Computing Enhancements

Network computing or client-server/cooperative-computing promise to enable a new generation of highly interactive information sharing applications. Such capabilities will allow more effective use of valuable corporate, or industry-wide information.

The Network Computing Enhancements will build on existing industry-standard base services such as remote files services (NFS and RFS), Remote Procedure Call, and External Data Representation to expand today's distributed applications into high levels of functionality and distributed applications. Functionality to be considered includes both X.400 electronic mail and X.500 network services.

In addition, this release will include tools, features, and interfaces such as naming services, location brokers, and stub generators. The goal is to provide an environment in which to create truly distributed applications. System administration and network management must also be enhanced to monitor and control the various pieces of a network computing environment.

5.6.2. OSI Communications

OSI support is critical toward System V's interoperability goal. This requirement is to enhance System V's already strong network features by providing layers 1-4 of the OSI communications standard. This implementation will be based on UI's OSI Work Group's definition of a common API not only for layers 1-4 but also for the full-stack which will reflect the merged GOSIP standards.

5.6.3. Distributed Multiprocessing

This requirement is to create an environment for the operation of a "single program" running on multiple "loosely-coupled" systems. This is a logical extension of the tightly-couple multiprocessing features of SVR4 Multiprocessing Plus and the distributed applications discussed above. These features provide for a real "network OS" in which an application can readily be split across a network among heterogeneous processors.

5.6.4. Network Management

With the new network computing environments comes an additional need for network management capabilities. This requirement is for service and application interfaces for full-function heterogeneous network management. The services supported should include dynamic configuration of the network, performance analysis on any part of the network, and problem identification facilities.

These features should be available from any network node, and information about changes should be automatically propagated throughout the network. GUI technology should also be employed in these network management applications.

5.6.5. Object Management Strategy

This requirement is for object oriented capabilities to increase software development and application operation productivity. The first requirement is for complex object exchange which will allow a user to share various types of data (e.g. graphics, spreadsheets) among applications. The second requirement is for an OS interface for C++ which will provide an object-oriented interface library to the System V kernel.

5.6.6. Real-time Enhancements

Real-time features will continue to be required by the market for applications ranging from industrial automation to transaction processing. This requirement is to evaluate the features of the POSIX 1003.4 "real-time" standard, implementing some features and providing hooks for others.

5.6.7. System Administration

With the complexity of System V ever growing, system administration features must be provided to aid the system administrator. Required features include a comprehensive resource monitoring and control system and autoconfiguration of the kernel. These features should, naturally, be GUI-enhanced.

References

[Com89a] UNIX International Technical Communications Committee, *UNIX International Structure, Processes, Products*, February 16, 1989.

[Def85a] U.S.A. Department of Defense, *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985.

[Int89a] UNIX International, "UNIX System V Release 4.0 Product Backgrounder," in *A UI white paper* (1989).

[Int90a] UNIX International, "UNIX System V Security; Today and Tomorrow," in *A UI white paper* (January 1990).

X.400 Server Toolkit on UNIX

Hans Georg Baumgärtel

Softlab GmbH
Zamdorferstr. 120
D-8000 Munich 80
West Germany
bat@softlab.uucp

ABSTRACT

The establishment of the X.400 as a new and far-reaching technology demands the adaptability of X.400 products to the needs of existing environments. There is a need for an electronic post-office in mayor environments using X.400. The X.400 Server Toolkit described in this paper is the toolkit to construct an X.400 server for this purpose. An important element of the construction of this toolkit is the interface definition in terms of abstract services.

1. Introduction

X.400 is the OSI standard for electronic mail. Since its acceptance in 1984 it has gained importance in the provision of services by many PTT's and the release of products by most manufactures of computer equipment. The X.400 standard in its 1988 version [CCI89a] brings new functionality as the use of X.500 directory, access to telematic services, message store and secure messaging.

On the other hand, X.400 is far from being a commonly available service or to become the dominant medium for message exchange. In the UNIX world unix-mail is established. Older technology such as Telex is reaching its zenith. Teletex has importance in some European countries. Facsimile (FAX) is experiencing enormous growth.

The establishment of X.400 as a new and far-reaching technology demands the adaptability of X.400 products to the needs of existing environments. This includes the interworking with the above mentioned telematic services and with manufacturer specific and system specific mail systems.

In data communications of large companies there exists already an enormous communication infrastructure. This infrastructure has to be incorporated in the introduction of X.400. An X.400 server is an instrument for this. But as this infrastructure is different from company to company, a new server has to be realised in every case.

This necessity is the starting point of the X.400 Server Toolkit described in this paper. It is a UNIX toolkit, designed to be portable to many hardware platforms. In this way it can be incorporated in many different environments. The X.400 Server Toolkit offers X.400 in its 1988 version with additions. These additions (see Figure 1) are functions that are outside the scope of OSI standardisation since they are considered as "local matter" and functions that are not yet completely defined by the base standards and the functional standards.

The X.400 Server Toolkit is designed to build servers. It contains no mail user interfaces. There are products for those interfaces available. It is desirable that user interfaces be an integral part of the users working environment (e.g. his office automation system).

The requirements on an X.400 server depend on the environment in which it is used. This concerns the functionality, the performance and the reliability of the X.400 server. The X.400 Server Toolkit allows configuring of the server gradually in this regard.

The X.400 Server Toolkit is a UNIX toolkit but designed to be portable to other operating systems.

2. Structure of the paper

In "Components of the X.400 Server Toolkit" the functionality of this product is described. The next section derives the requirements of the design of this toolkit. Then the concept of the abstract services in CCITT X.407 is explained and in an example it is outlined how it is used in the design of the toolkit. The

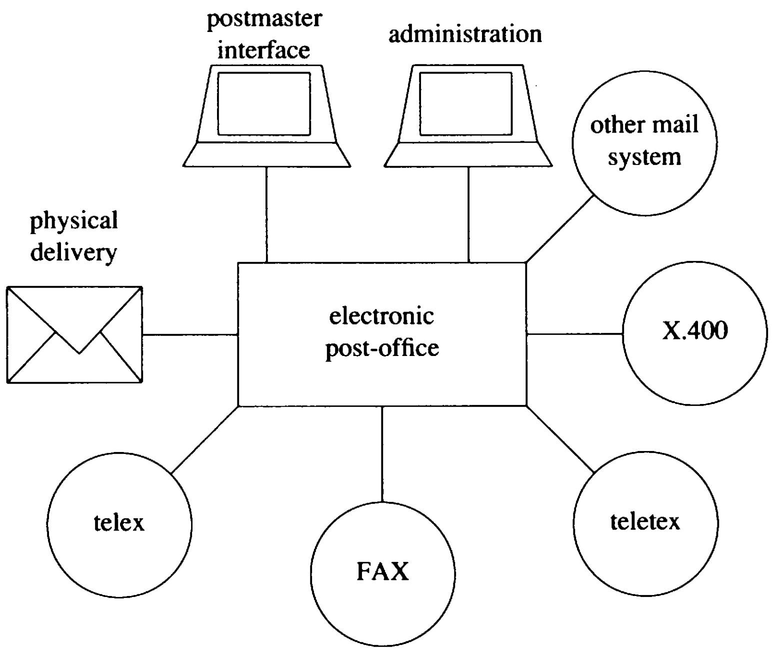


Figure 1: *Functionality of the X.400 server toolkit*

succeeding sections show the principles of its realisation in a UNIX system and the tools needed for its implementation.

3. Components of the X.400 Server Toolkit

The most important components of the X.400 Server Toolkit are the Message Transfer Agent, the User Agent and the Message Store of X.400 as defined in the 1988 standards.

The Message Transfer Agent is provided with a configurable routing function. This routing function allows special routing decisions as required in the target environment.

A User Agent is included, though the toolkit contains no mail user interface. It can be used to adapt user interfaces that contain no user agent themselves or to provide automatic user agents.

The toolkit contains the following additions that are outside in scope of the OSI standardisation:

- a comfortable administration
- a postmaster interface
- gateways to manufacturer specific mail systems

and functions that are not yet completely defined by the base standards and the functional standards:

- an access unit to telex, FAX and physical delivery
- a telematic agent for teletex
- use of X.500 directory
- secure messaging
- charging and accounting

4. Requirements of the Architecture of the X.400 Server Toolkit

The X.400 Server Toolkit is designed to be a toolkit. Therefore it is necessary that its components can be combined in many ways. It shall be used in projects with different requirements for the functionality, performance and reliability of the system.

To match different requirements for functionality, it must be possible that not all components need be used in all configurations.

Different needs in respect of performance and reliability can be covered by machines of different capacity and fault tolerance. The X.400 Server Toolkit must be therefore portable. This includes the use of such standard interfaces as TLI and the gateway API of the API association. A different approach is to distribute

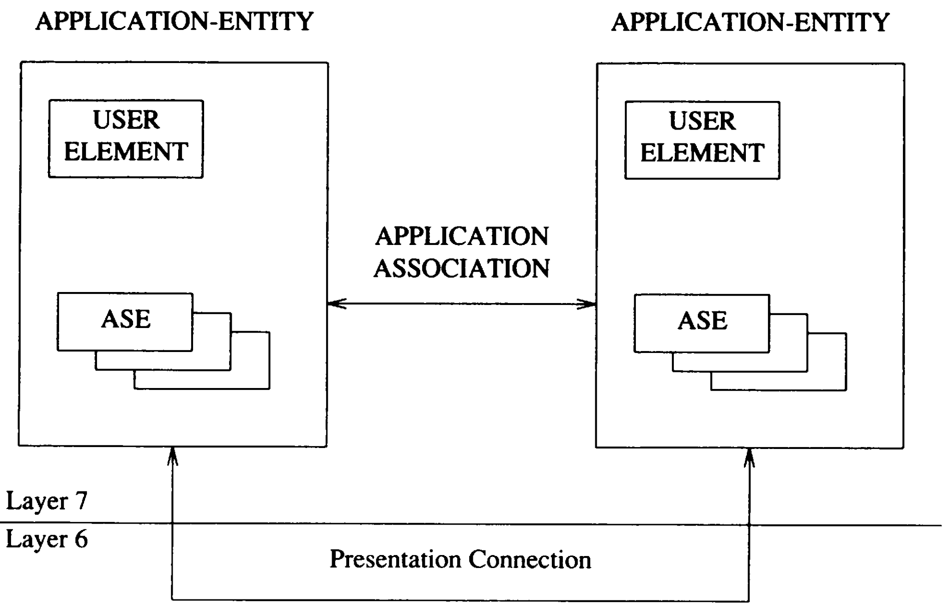


Figure 2: *The ASE Concept of X.402*

components of the server over two or more machines. The X.400 Server Toolkit helps here with interfaces that can be formed to operate over remote procedure calls.

It is an experience of everybody working in X.400 related projects that the expected time-scales are short. There is a need for functionality before it is defined by OSI and the organisations responsible for functional standards. The X.400 Server Toolkit contains pre-standard realisation. As soon as the standard is released it is necessary to change over to a conform realisation. Therefore an implementation is required that fits into the architecture of the X.400 standardisation. The fundamental description method is outlined in the CCITT X.407.

5. Abstract Models and Abstract Services of CCITT X.407

The X.400 recommendation in its 1988 edition follows the OSI application layer structure as it is defined in X.200. It avoids the introduction of sublayers in the application layer and the defining of 'sub'-layer-services. Instead it uses cooperating Application Service Elements (ASE).

A distributed application is modelled as application entities on two or more computer systems (Figure 2). These application entities establish an application association. The application entities themselves consists of a user element (UE) and one or more ASEs. The UE provides a service for other UEs. This is done with the help of the ASEs which use the lower OSI layers via the presentation layer.

The ASEs transform the requests and responses from the UE into an application protocol and protocol information in indications and confirmations to the UE. The AE provides its service to the UE by a transformation of service primitives into protocol information. This is what the X.402 recommendation means when it describes the role of the ASE as largely mechanical. Therefore the service definition and the protocol definition are partially redundant. This is an unnecessary overhead and a source of inconsistencies in definitions and of errors in implementations. The X.407 recommendation eliminates the redundancy of a separately defined service and protocol and introduces the concept of abstract services.

The X.407 recommendation "makes it possible to describe a total service in an object-oriented top-down way, and then refine that total service" [Man89a]. It introduces the description of the total service and the environment in which this service is provided by an abstract model. This abstract model consists of abstract objects. At the same time X.407 uses pictures such as Figure 3 to illustrate the abstract models. The abstract objects are represented by ovals, its ports by small squares on these ovals and the use of the ports by abstract services by lines connecting the squares.

The X.400 message handling service (MHS) and its community of users are described by such a model. The users and the MHS in total are described in this abstract model. They are described as abstract objects. The origination and the delivery of mail from and to the users is the interaction of abstract models on abstract ports. The users are consumers of services of the MHS, abstract objects consume and provide abstract

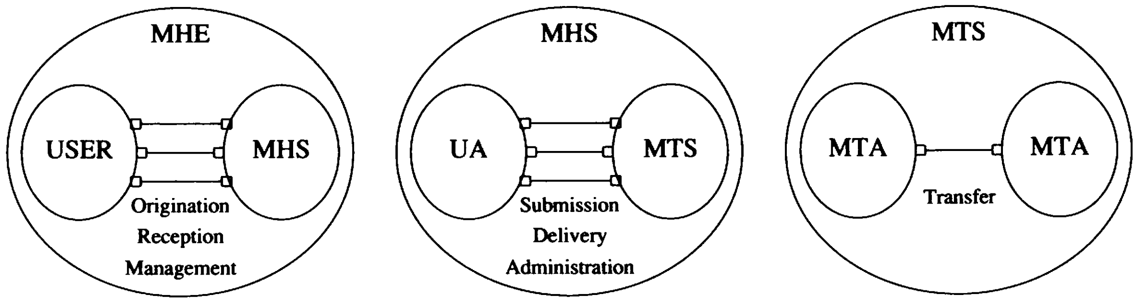


Figure 3: *Refinement of MHE, MHS and MTS*

services to each other. X.407 introduces the concept of refinement of the abstract objects. The refinement is a step in the top-down definition process. An abstract object just regarded as one entity will now be considered as composed of different other abstract objects. The MHS as an abstract object can be refined in its components which are User Agents, the Message Store, Access Units to telematic services and to physical delivery and the Message Transfer Service (MTS). These again interact via ports with each other. The refinement of the MTS shows it consisting of MTAs (Figure 3).

The abstract service is provided on an abstract port on an abstract object by abstract operations. In the X.407 recommendation abstract objects, abstract ports, abstract services, abstract refinements and abstract operations are defined in terms of ASN.1 macros. The ASN.1 macro mechanism is defined in the X.208 recommendation. In many programming languages macros are text substitution facilities. In ASN.1 macros allow extension of the ASN.1 syntax. The X.400 series of recommendations uses the macros from X.407 to define protocols and services.

The following example gives an impression of how the ASN.1 macro mechanism is used in the definition of the X.400 protocols and services. The abstract objects and the abstract ports are formally defined in X.407 by ASN.1 macros. These macros use the macro definitions of the ROSE definition in X.219.

The X.411 recommendation uses these macros to define the ports of the Message Transfer Service and the P3-protocol (in excerpts in Figure 4).

The X.407 recommendation realises that this ASN.1 macro notation can be no substitute for a formal description language for the provided services. An abstract operation is an abstract data type and contain the arguments, results and errors of the operation. Its ASN.1 notation describes the data types of parameters and results but not whether supplier or consumer provide them [Gor87a].

6. The X.400 Server Toolkit in the X.407 Abstract Model

The use of the description of abstract services as defined in X.407 is shown here only in two examples, the design of the access units and the postmaster agent.

```
mTS OBJECT
        PORTS { submission [S], delivery [S], administration [S] }
        ::= id-ot-mts
submission PORT
        CONSUMER INVOKED { MessageSubmission,
                           ProbeSubmission, CancelDeferredDelivery }
        SUPPLIER INVOKED { SubmissionControl }
        ::= id-pt-submission
MessageSubmission ::= ABSTRACT-OPERATION
        ARGUMENT SEQUENCE {
                envelope MessageSubmissionEnvelope,
                content Content }
        RESULT SET {
                message-submission-identifier MessageSubmissionIdentifier,
                ...
        }
        ERRORS {
                SubmissionControlViolated,
                ...
        }
```

Figure 4: *Example of a Port Defined by the Port Macro*

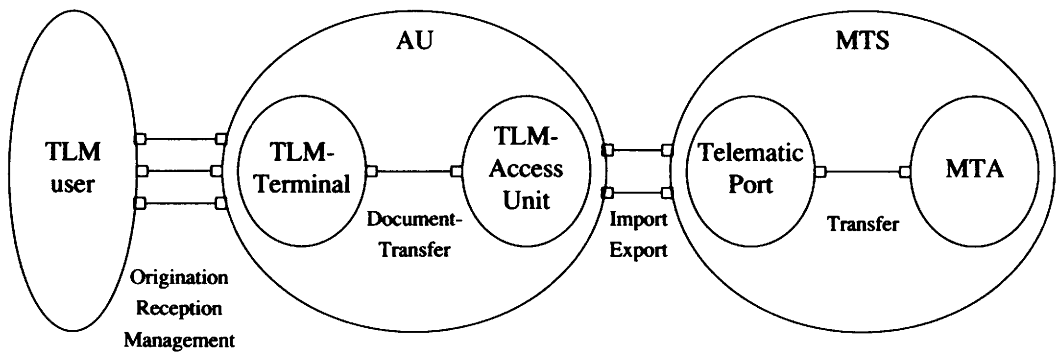


Figure 5: *Refinement of the AU and the MTA*

The X.400 Server toolkit contains access units for telex, FAX and physical delivery and a telematic agent for teletex. An access unit, introduced by the X.420 recommendation, helps a telematic user to to access the Message Transfer Service.

In Figure 5 the telematic agent is shown as an abstract object. Neither the X.420 nor the T.330 recommendation defines the ports "Import" and "Export" of the MTS in their ASN.1 notation. This description is the necessary starting point for the design of the telematic agent. The definition of the ports has implications for the generation of reports about the delivery of messages to the telematic user.

The T.330 recommendation assumes "import" and "export" to be defined like the "submit" and "deliver" ports of a user agent. This has the consequence that an X.400 user sending a message to a telematic user will obtain a report about the delivery of his message once it has been "delivered" to the telematic agent. A failure of the transmission via the teletex must be signaled by different means. The F.415 recommendation requires equivalent behaviour of a Physical Delivery access unit.

The U.204 recommendation has a different requirement for a telex access unit in this respect. Also a FAX-access unit – not defined yet by a standard – is expected to give the delivery notification when the FAX is transmitted.

To provide an "import" and an "export" port for the MTA, the MTA is refined to an MTA without these ports and an entity that is called "telematic port" (Figure 5). The telematic port is connected to the transfer port of the MTA and behaves as a neighbouring MTA. On its "import" and "export" ports it provides the service needed by the access unit or telematic agent. In contrast to the "submit" port of a MTA, the "export" port allows its consumer to specify a result which is used by the MTA to generate a delivery or non-delivery report. The transfer port between the MTA and the "telematic port" is realised using the "Gateway API" of the APIA as a standardised interface.

The physical delivery access unit and the telematic agent will give a positive result when they receive the message to transmit. The T.330 recommendation refines (see Figure 5) the telematic agent and introduces its refinement into a telematic terminal and a telematic access unit. The document transfer port defines the interface between them. This port is specified in the T.330 recommendation and is not duplicated here.

The postmaster agent is the entity in the Message Transfer System that interacts with the postmaster via the postmaster interface. A postmaster agent is not mentioned in the standard. It is purely a "local matter", but a useful function as stated above.

Again an additional function is incorporated in the MTA by a refinement. The MTA with a postmaster interface is refined to an MTA without one and a postmaster agent (Figure 6). The postmaster agent is connected to a transfer port of the MTA as a neighbour MTA. It provides the interface for the postmaster to access and to handle messages. On these ports the postmaster can inspect messages and correct them if necessary, and approve their addresses. In this way he can enter addresses of FAXes without automatically readable addresses. In this way incoming FAXes can be delivered via X.400. The postmaster can change addresses to a postal O/R name to let them be delivered via physical delivery.

7. The Realisation on UNIX

The X.400 server toolkit is portable UNIX software. This has implications for its interfaces. Communications software has to interface the host processor to a variety of front end cards. Standard interfaces give the chance to burden the system manufacturer with this problem. Therefore the X.400 server toolkit uses the TLI. There are UNIX systems that come with a X.400 implementation and there will

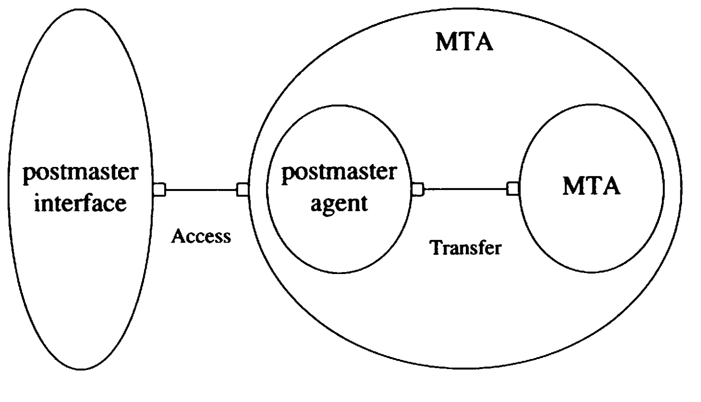


Figure 6: *Construction of the Postmaster Interface*

be more in the future – e.g. as a consequence of the cooperation of RETIX and AT&T. These implementations can be used together with the X.400 server toolkit if it provides an API interface, since the X.400 server toolkit is API based.

The components of the X.400 Server Toolkit are UNIX implementations of entities as they are described by abstract objects. The interfaces between these entities are described by abstract ports and abstract services.

These entities can be UNIX processes or a set of them, on one UNIX machine or distributed on different machines. They can be subroutines inside one process. Their implementation has consequences for the implementation of the interfaces. The CCITT X.407 mentions OSI and proprietary realisations of abstract ports. In one or more UNIX systems a proprietary realisation of abstract ports can be:

- C subroutine calls
- Interprocess communication (this includes files)
- Remote procedure calls

The OSI realisations can be ROS based or not. An OSI realisations is only necessary if a heterogeneous distribution of the entities is required.

The implementor of an X.400 server in UNIX has to choose for every port which kind of interface he will implement. This decision has implications for the performance and the reliability of the server. The implementor can e.g. distribute different components to different machines connected via RFS and RCP. In this way he can increase the throughput of the system and its reliability.

The X.400 Server Toolkit leaves this decision open. For all relevant interfaces it offers the choice between these interface types. This raises the problem of controlling the overhead to implement consistently all interfaces in more then one incarnation. This becomes even more relevant if the definition of the interface changes. This will happen if the X.400 Server Toolkit implements part of its functionality before the standardisation is completed as soon as the standard or functional standard comes out.

8. Generation of the Interface Functions

It is state of the art that the implementor of an OSI service on UNIX uses an ASN.1 compiler to generate the access functions to the protocol information. The components of the X.400 server toolkit are described above as abstract objects. There interfaces are defined by abstract operations. These are defined as ASN.1 macros. The ASN.1 compiler will generate the access functions to the arguments, results and errors of the operations.

As show above the ASN.1 compiler will not generate functions to perform the operations that realise the ports. This can be done by an additional tool, the "interface generator". It generates the interface functions in their different incarnations as subroutine calls, IPC or RPC. The cooperation with the ASN.1 compiler is shown in Figure 7. The interface generator uses the macros defined in the recommendations X.407 and X.219, a port definition and the type of interface requested being the input. An example for the port definition is given in Figure 4. This is used to identify the names and data types of the arguments, results and errors. The type of the interface specifies whether the interface is realised by a subroutine call, IPC or RPC and whether in is a consumer- or supplier-interface. The interface generator derives the ASN.1 definitions of the arguments, of the results and of the errors of the operations and C-code to provide these

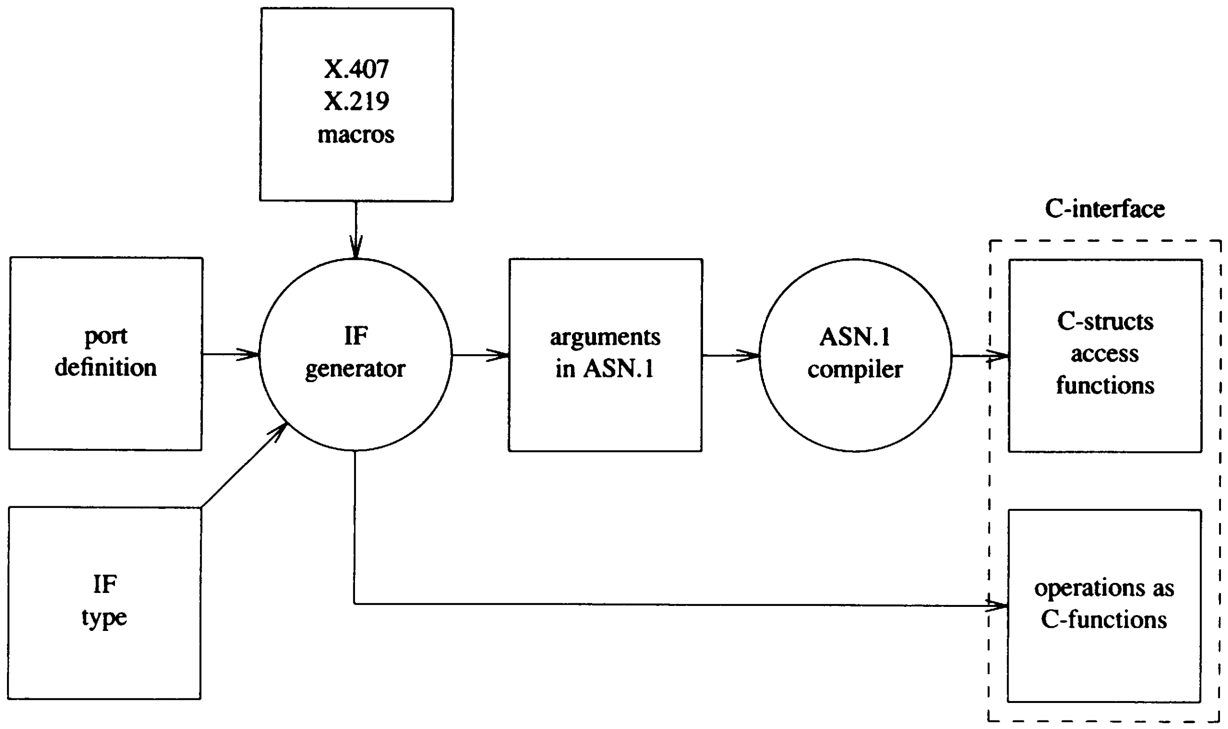


Figure 7: *Generation of the Interface Functions*

operations of the port as subroutine calls, IPC or RPC. The ASN.1 compiler generates the C-structs and the access functions for the arguments, results and errors of the operations.

9. Conclusion

The concept of abstract services can be used as an architectural principle for the design of a flexible X.400 Server Toolkit. This toolkit provides X.400 (1988) with additional functionality. It is configurable with respect to the functionality, the performance and the reliability of the X.400 server.

References

[CCI89a] CCITT, *Recommendations of the CCITT of the X.200, X.400 and T.300 series*, 1989.

[Gor87a] Walter Gora and Reinhard Speyerer, *ASN.1 (in German language)*, 1987.

[Man89a] Carl-Uno Manros, *The X.400 Blue Book Companion*, 1989.

Setting up an X.500 Directory Service

Dr Andrew J Findlay

Brunel University
Uxbridge, UK
Andrew.Findlay@brunel.ac.uk

ABSTRACT

One of the more difficult aspects of using electronic mail is finding the address of the person you want to talk to. At present there is no complete solution to this problem: a global directory is needed.

The X.500 standard describes a distributed directory service that has the potential to solve this problem. There are now several implementations of X.500 and the Directory is developing into a useful international service.

This paper introduces the X.500 system, and describes Brunel's experience with setting up and running a directory service based on ISODE/Quipu. Software configuration is briefly discussed and the data-gathering process is described. Suggestions are given for merging data from several sources to form coherent directory entries.

1. The Need for a directory

How often have you come back from a conference half-remembering somebody's name and then wanted to contact them? How many times do you refer to your organisations internal phone book each week? How many people come to you each week wanting to know the electronic mail address of somebody at a site you have never heard of?

Directories are a part of life. The problem (as with standards) is that there are so many to choose from. Just finding out what directories exist can be difficult; a directory of directories would be a useful thing!

Better still would be a single directory service, covering the whole world and listing everything that every existing directory contains. This single Directory would of course be totally user-friendly and would do exactly what was expected rather than what was asked for! This of course is impossible, but it is a reasonable description of what X.500 is trying to do.

2. The Scale of the Problem

To be really useful, the Directory must contain information on every person *in the World.*

Ultimately there will probably be three entries for each person:

- Residential person
- Organisational person
- Job title

Even if the X.500 Directory is restricted to people who can be contacted by some electronic means (telephone, telex, fax, electronic mail) the numbers are staggering.

Consider a few statistics:

Usenet News readership:	250 000
Telephones in the UK:	28 Million
People in the UK:	56 Million
People in the USA:	240 Million
People in China:	1000 Million?

It is obvious that a centralised directory is impossible. Apart from the amount of data, the number of queries must be considered. If each person in the directory is responsible for just one query each day, a

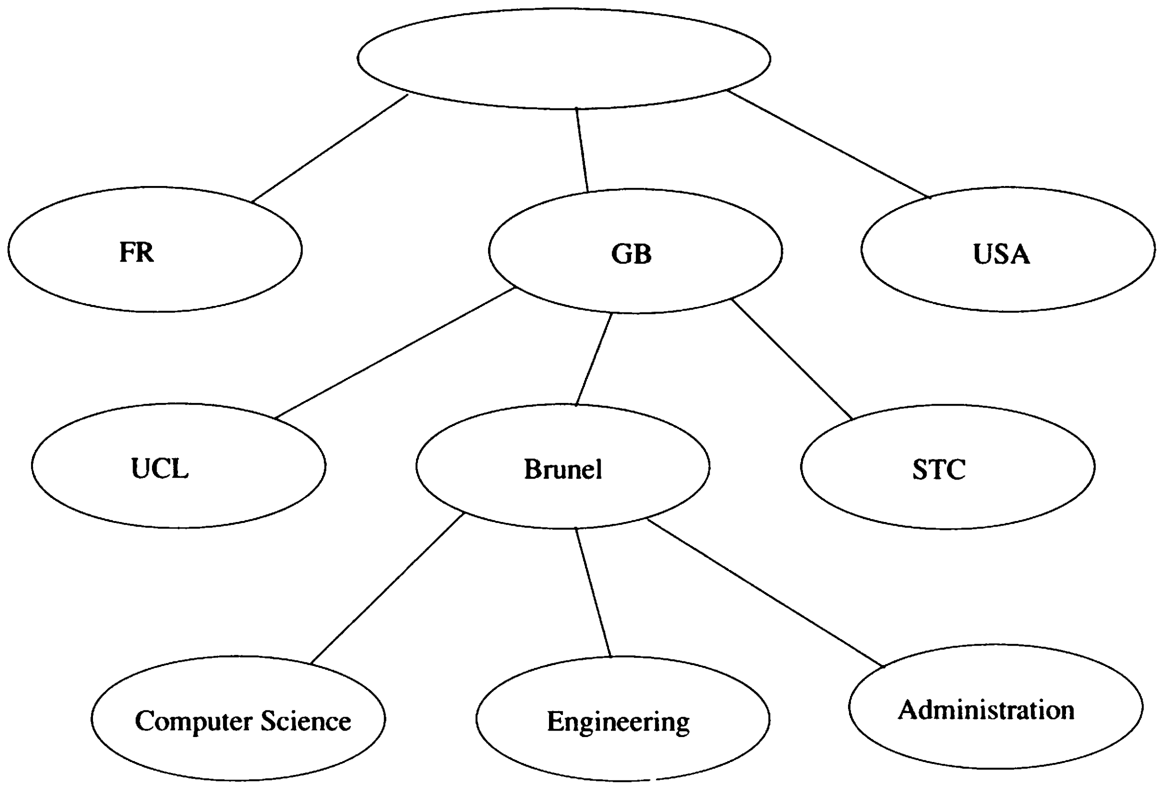


Figure 1: *Part of the Directory Information Tree*

central system would be swamped. Only a massively distributed directory system will stand any chance of working.

3. X.500

The X.500 standard was developed jointly by ISO and CCITT [ISO88a]. It describes a distributed directory system based on a network of "co-operating computer systems". The standard lays down the communications protocols to be used, and defines the overall structure of the information to be held.

3.1. The X.500 Information Model

The X.500 Directory is structured as a tree, very much like the UNIX file system. This structure is referred to as the Directory Information Tree or DIT. Each node in the tree may contain information, in the form of attribute-value pairs. One or more of these attribute-value pairs are taken to be the "name" of the node and are referred to as the Relative Distinguished Name, or RDN. For example, at the top of the tree there are nodes relating to countries; they have names like *country=GB* and *country=FR*. Below each country there are nodes representing organisations and geographical areas (Figure 1).

Organisations can be further divided into *organisational units* nested to any level. Entries for peopie and services are generally leaves of the tree.

Each node in the tree contains information relating to one object. For example, part of the entry describing Brunel University is shown in Figure 2.

Note that there are several values given for the attribute *organizationName*. The first one is the official name of the University, and is used to name the node in the DIT.

Further down the tree, entries for people may duplicate some of the information given at *organisation* level. This is because the current version of the X.500 standard does not define any attribute inheritance mechanism so each node has to be treated independently.

So far, the Directory has been described as a pure tree. In fact, it is not quite that restrictive as there are two mechanisms that allow cross-linking. A node may be set up as an *alias* for another node. In UNIX terms this is just like a symbolic link: two or more names can refer to a single object. As with symbolic links, it is possible to have an alias that does not point to a valid node. The other mechanism is the *see also*

Attribute name	Value
organizationName	Brunel University
organizationName	Brunel The University of West London
organizationName	Brunel
telephoneNumber	+44 895 74000
postalAddress	Brunel University Kingston Lane Uxbridge UB8 3PH UK
businessCategory	Education
description	Technological University offering sandwich courses
localityName	Uxbridge, West London

Figure 2: *Entry describing Brunel University*

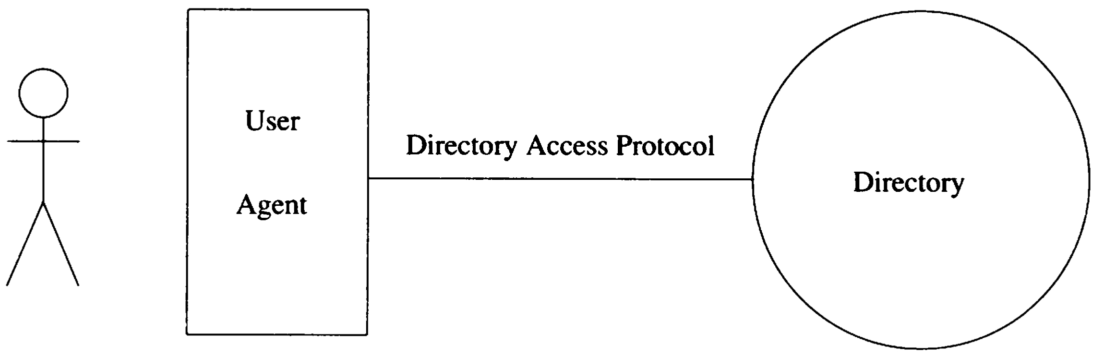


Figure 3: *The X.500 service model*

attribute, which allows a looser association to be established between two nodes. Together with other "connecting" attributes, these mechanisms allow most organisations to be modelled quite well.

3.2. The X.500 Service Model

In line with other communication standards, X.500 divides the work to be done into "user-related" functions and "system" functions. Each user has a "Directory User Agent" (DUA) which communicates with the directory system on their behalf (Figure 3).

The directory system itself is comprised of a number of "Directory System Agents" (DSAs). The DSAs communicate among themselves using a protocol called DSP (Directory System Protocol). A DUA normally connects to a single DSA which provides all services required by the user, referring to other DSAs where necessary (Figure 4).

The Directory Information Tree is distributed among the DSAs so that each subtree is held in a DSA "near" the objects or people described in that subtree. A large organisation might have several DSAs; each could hold information relating to one or more departments. It is likely that some information would be replicated, copies being held in two or more DSAs to improve performance and reliability.

4. Quipu

Quipu is a freely-available implementation of X.500 which was designed at University College London and is distributed as part of the ISODE package [Kil88a, Ros90a]. Quipu will run on most BSD-like UNIX systems, and can communicate using TCP/IP as well as X.25. The design places great emphasis on flexibility so that experiments can be carried out easily.

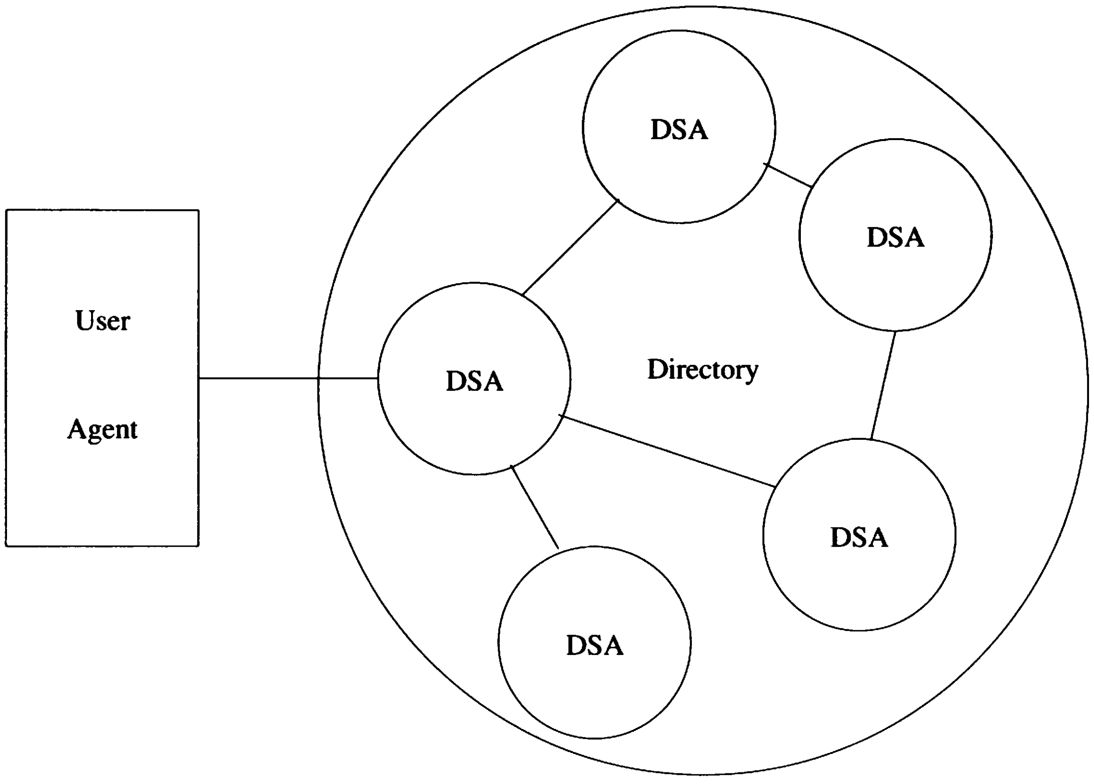


Figure 4: *The distributed service model*

Rather than having an interface to a conventional database system, Quipu holds all data in memory. This means that searches can be carried out on any attribute with equal efficiency, as there are no index files. The disadvantage of this approach is that DSA processes use a lot of virtual memory and can cause heavy paging on small machines. Good performance can be obtained though, and response times below 1s have been demonstrated.

5. Configuration

For such a large package, ISODE is remarkably easy to build. A set of configuration templates are provided covering a wide range of machines. At Brunel we use Sun-3, Sun-4 and Pyramid machines, and the only changes needed to the standard files concern our preferences for filesystem layout.

ISODE is large: the source code takes up about 16MB, and at least 60MB more is needed to build the system. The installation process uses about another 20MB so this is not a game for sites with a disk shortage!

Building and installing Quipu is relatively easy; the hard part is tailoring it to your local conditions. The ISODE distribution contains several distinct modules. It is advisable to build and test the simpler ones before starting on Quipu. The command `./make all` will build the core libraries and the *imisc* "little services" (OSI versions of *finger*, *write*, *rdate*, etc.) Once these simple services have been installed, they can be tested by hand or by using the automatic test program. At this stage, any glaring errors in the *isotailor* file can be sorted out.

With the simple services working, it is reasonable to start compiling Quipu with the command `./make all-quipu`. This is usually straightforward, and Quipu can then be installed. If you have access to the Internet or an established X.25 network the Directory User Agents can be tested by connecting to somebody else's DSA e.g.:

```
dish -c giant
```

The next stage is to set up a skeleton database for your own DSA. It is best to start from one of the example configurations and modify the entries one at a time with frequent tests. When you have a DSA with an entry for your own organisation, you can ask the manager of your nearest country-level DSA to add your

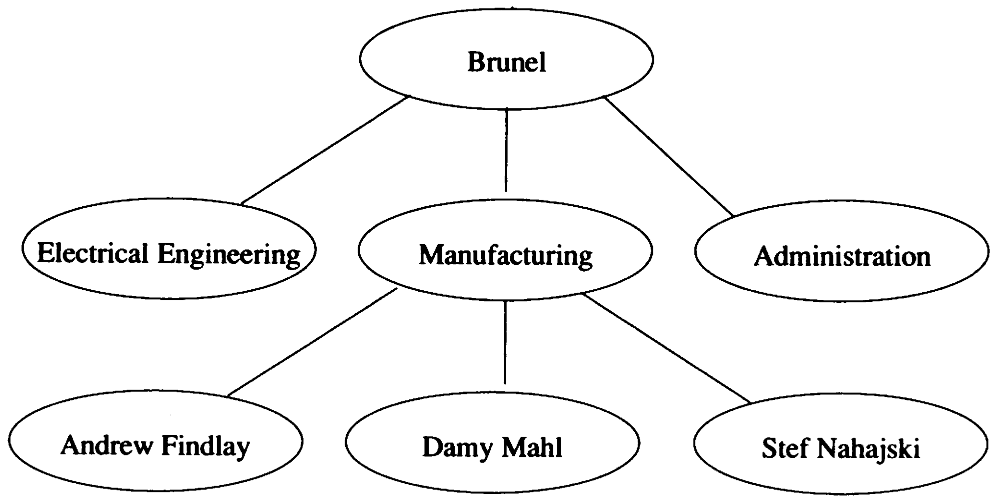


Figure 5: *The DIT structure for Brunel University*

details to the master database. Quipu DSAs usually update important data every six hours so a new DSA is soon widely recognised.

6. Data

A large organisation is a very complex thing. It may have many levels of hierachy and complex relationships between the individual parts. For example, a university might have several faculties, each containing several departments. The departments may have internal structure, being divided into research groups, teaching groups, administrative functions, and so on. To make matters worse, many people will be members of more than one group or even members of two departments.

When designing the structure of the DIT for an organisation, it is necessary to decide how closely the organisational structure will be modelled. Where an organisation has a well-defined structure, it is tempting to make the DIT match that structure exactly. This is probably a mistake. The main purpose of a directory is to allow people to locate other people and anything that might make that more difficult should be avoided. Although the structure of an organisation is usually well-understood by its employees, outsiders are likely to be confused by it. In any case, many organisations regard their internal structure as confidential information and are not willing to publish it. From a pragmatic point of view, the DIT structure should be easy to derive automatically from the available sources of data.

The DIT structure chosen for Brunel University is very simple, and follows the “wide flat tree” approach used at higher levels. There is a single level of “organisational units”, which represent the departments. People appear at the next level down (Figure 5)

So far, no attempt has been made to create entries for “organisational roles” as there is no central database that can supply that information. People have been entered as “organisational person” objects.

A few other object types have been used. For example, the “room” type defined by the THORN [Kil89a] project is used where the telephone number for a room is known. Interestingly, the current definition of a room does not permit a “room number” attribute! Presumably the room number was expected to be part of the name. Some OSI services are also registered in the Directory, including the Directory itself.

6.1. Sources of Data

There are four major sources of data for Brunel’s part of the DIT:

Staff Database	This is derived from the University’s personnel database. Each person’s entry contains their surname, forenames, initials, title, and department.
Student Database	This is derived from the Registry database. It contains the same information as the staff database, plus the course of study and yeargroup.

```
Surname=FINDLAY
Initials=A
PersonalTitle=Mr
Department=Manf and Eng Systems Group
Status=Staff
Room=EC 21
Phone=2512

Surname=FINN
Initials=A
PersonalTitle=Mr
Department=Electrical Engineering
Status=Staff
Room=E 409
Phone=2946
```

Figure 6: *Example showing the intermediate data format*

Telephone Database	This is supplied by the University telephone exchange. It contains entries for both people and rooms. For people, surname, initials, title, department, and phone number are listed. A room number is shown for most people. Entries for rooms show the name of the room, the department that it belongs to, the room number, and the phone number.
Electronic Mail Database	The tables that drive the electronic mail system are processed to supply surname, initials, one forename, department, class (staff or student), and RFC822 mailname.

The data from all sources is converted to a standard "attribute-value" format by a collection of shell scripts. An example of this format derived from the staff database is shown in Figure 6.

6.2. Building the Directory

Brunel's part of the DIT was constructed in two phases. In the first phase, a list of Departments was generated from the input data files and the appropriate *organizationalUnit* nodes were created. The second phase created entries for people and rooms under the departmental nodes.

Almost all the processing involved in loading the Directory is done with shell scripts. Where access to the Directory is needed, *dish* commands are used. Only two programs had to be written in "C" and both are small. The rest of the work is done by *sh*, *awk*, and *sed*.

Interesting problems were posed by the variable quality of the input data. For example, the three databases supplied by the University administration all used different conventions for naming departments. Some databases were upper-case only, and there were several variations in the way initials were represented. The two "C" programs mentioned above were used to sort out the initials and attempt to restore correct upper-case and lower-case letters in names. As the administration data came from a system with fixed-width fields, many department names were truncated so they could not be used directly. After an abortive experiment with X.500 aliases, it was decided that the *info* attribute would be hijacked to identify the departments that these "shortnames" referred to. Thus, when each department's entry was created, the *info* attribute was set up with a list of shortnames as well as a list of *organizationalUnit* names (see Figure 7).

Note the *objectClass* and *treeStructure* attributes. They define the type of the node, and the types of nodes that may appear beneath this one in the DIT, respectively.

The scripts that handle creation of entries for people and rooms are designed to be a general-purpose data merging tool. The same scripts are used for all data sources, working from the standard intermediate file format. Figure 8 shows the algorithm used on each entry in the input file.

Although all sources of data are handled in the same way, it is important to process the "best" sources first. For this purpose, a good data source is defined as one which is likely to have complete and correct information for the fields it defines. Thus, a database derived from the payroll is "better" than one derived from a UNIX password file because people are more likely to complain if their entry on the payroll is wrong! The payroll file is also more likely to have the complete name and the correct department.

Attribute	Value
organizationalUnitName	Computing and Media Services
organizationalUnitName	Computer Centre
organizationalUnitName	Media Services
info	Shortname: Computing and Media Services
info	Source: DEPARTMENTS-from-Admin
info	Shortname: Computer Centre
info	Shortname: Media Services
treeStructure	room & thornPerson & thornObject & quipuNonLeafObject & quipuObject & applicationProcess & organizationalPerson & organizationalUnit & alias
objectClass	thornObject & quipuNonLeafObject & quipuObject & organizationalUnit & top

Figure 7: *Example of a Department entry*

```
Build standard fields as they will appear in the Directory

Find the correct departmental entry using the "Shortname" fields
and make that the "current position".

Search this department for a node matching the surname exactly and
the full name approximately.
If more than one entry is found, report an error and stop processing this entry.
If no entry is found, create one using the available information.

Retrieve the information for the entry.

Add any fields that are supplied by the input file but are missing in the Directory entry.

If any changes were made in the previous step, write the entry back to the Directory.
```

Figure 8: *The data-merge algorithm*

The algorithm described is not very good when dealing with poor-quality data. Where the data sources are not consistent, it is possible that several directory entries get created for a single person. For example, the electronic-mail database is often wrong about the department that a user belongs to. This would cause two entries to be created for the person concerned: one containing only an e-mail address, while the full entry with phone number and postal address appear in a different department. It is difficult to be certain about how many errors are caused by this algorithm, but a rough figure can be calculated: There are 1200 people listed in the staff database, and 4300 in the student database. The Brunel DSA reports 6800 entries, so it is possible that 1300 of these are duplicates! In fact there are less duplicates, because the student database does not include most of the part-time and external students who have computer accounts and therefore e-mail addresses. Other interesting facts emerge from scanning the log of the build process: some people that are apparently members of staff are not on the payroll, and quite a few people who are neither staff nor student have e-mail addresses. Taking these factors into account it seems likely that less than 10% of the entries in the Brunel directory are duplicates. There is scope here for a tool to identify likely duplicate entries.

Several important areas of data-handling have not yet been addressed. The most significant one is the need to delete obsolete entries. A university has a very large annual turnover of people: about 25% of the students graduate each year and are replaced by freshers. Staff come and go, or even move from one department to another. Telephone numbers change, and so do personal roles. If a directory is to be respected by its users, it must be kept up-to-date. The "proper" way to do this is to provide an X.500 interface onto the central administrative database, but it will be some time yet before this is possible. (It helps to **have** a central administrative database to start with!) In the interim, much effort and ingenuity will be needed to derive update information from multiple sources of data.

Parallel work at UCL [Bar90a] addresses some of the problems described. The UCL update system is based on comparison of the latest information from each source with the previous version. The list of differences generated is then converted into a list of updates to be performed on the Directory. The advantages of this approach are that fewer Directory operations are needed, and deletions are easier to cope with. The disadvantage is that it can be very difficult to recover from the failure of an update.

7. Use of the Directory

Within Brunel there are now a few people who use the X.500 Directory instead of the printed phone book. The main factor that discourages online information services at present is the widespread use of DOS for administrative jobs. Staff are not willing to drop out of their wordprocessor or spreadsheet package to load a directory-browsing application. It is hoped that the spread of X-terminals (and even PC windowing systems) will enable people to keep a directory application on their "desktop" in the same way that they might keep a clock icon.

Brunel is involved in the design of Directory User Interfaces for both the X Window System and MS-Windows [Fin89a]. In the early stages of this work, a curses-based interface called *sd* was produced. This has been made available as a public service on the UK academic network (JANET) as well as on our local UNIX machines. The public-access service has handled about 100 calls per month, mostly from people looking for data on their own institution. *sd* is the most heavily used of all directory interfaces within the University because it works on ordinary character terminals, though the use of our X-based interfaces *xd* and *pod* is increasing.

The Directory is already proving useful as a wide-area service. In the UK there is a pilot project funded by the Joint Network Team, which is aiming to get X.500 services into all universities. So far, about 10 sites have significant amounts of real data in their DSAs, and a further 10 are expected soon. The UK pilot is linked to similar exercises in other European countries, and to the US and Australian networks. In June 1990 it was estimated that there were over 200 000 entries in the DIT.

8. The future

The Directory is expanding rapidly, though at present the number of queries is small. When mail systems start to use X.500 for routing and mailing list expansion, the level of use will increase dramatically.

At Brunel, we hope to use the X.500 directory as the master source of telephone data and the printed phone book will be generated automatically from it. With the new user interfaces that we are developing, it should be possible to make departmental secretaries responsible for maintaining directory entries. This will ensure that entries are provided by those people who have the most up-to-date information.

There is still a lot of work to be done on merging data from different sources into the Directory, and then keeping track of changes. A lot of this will be specific to individual sites, but it is hoped that general methods can be developed for common operations.

References

[Bar90a] Paul Barker, *Managing Data Derived from Multiple Sources in an X.500 Directory,* UCL, London (June 1990). UCL Computer Science Research Note RN/90/6

[Fin89a] Andrew Findlay and Damanjit Mahl, "Designing an X.500 User Interface: The Early Stages," *Proceedings of the UKUUG and UKnet Winter Technical Meeting,* Cardiff, UKUUG (December 1989).

[ISO88a] ISO/CCITT, *Recommendation X.500: The Directory - Overview of Concepts, Models and Services,* ISO 9594 is technically aligned with X.500, March 1988.

[Kil88a] Steve Kille, *The Design of QUIPU,* UCL, London (July 1988). UCL Research Note RN/88/32

[Kil89a] Steve Kille, *The THORN and RARE X.500 Naming Architecture,* UCL, London (April 1989). UCL-64

[Ros90a] Marshall T Rose, *The ISO Development Environment: Users Manual,* Performance Systems International, Inc., Mountain View, CA. (January 1990). Version 6.0

Integration of Message Handling and Directory:

A System Manager's View

Steve Kille

Department of Computer Science
University College London
S.Kille@cs.ucl.ac.uk

ABSTRACT

The OSI Directory is a tool which has large potential to improve the manageability of Message Handling Systems. This sort of statement has been made frequently, but surprisingly little work has been done on the exact manner in which these two services will interact. This paper considers how the OSI Directory could be used to provide this management support. It describes this usage in terms of system management, rather than by using detailed OSI Terminology.

1. Introduction

There has been much work on implementation of X.400 based Message Handling Systems, and some level of deployment in operation, although this is a long way short of the levels of other messaging services [CCI88a]. Demand for the new services offered by X.400, such as multimedia and security, are likely to cause this to change. However, a range of problems have delayed the wide-scale deployment of X.400.

One of the most off-putting aspects of X.400 is the addressing. Consider the Author's X.400 O/R (Originator/Recipient) Address:

```
    I=S; S=Kille; OU=CS; O=UCL; PRMD=UK.AC; ADMD=GOLD 400; C=GB;
```

It is hardly possible to call this "user friendly". The OSI Directory has been cited as the answer to almost any problem of this nature [CCI88b]. However, whilst there has been some very interesting initial deployment of the OSI Directory in pilot exercises, little benefit has arrived at the Messaging Systems. Indeed, the understanding as to how the Directory will help is currently very limited.

Managing a mailsystem requires significant effort to update a number of tables relating to local users and message routing. For many message systems, this is an overly complex task, and yet does not really provide the right functionality. There is often significant duplication of information between various tables, with consistency maintained in an ad hoc manner. It would be beneficial to store such information at a single point. The OSI Directory can be used as such a repository.

This paper looks at how the Directory will help Message Handling. It is oriented towards the system manager, who will understand the problems of providing a message service using current technology. Details of X.400, X.500, and "OSI Speak" are avoided as far as possible.

2. The User View

Before the manager's view can be understood, it is important to understand the user's view. The user is who the manager is supporting. The task being considered is communication of documents between users. Document should be considered in a broad sense, and might be: a conventional document; a memorandum; code; a binary file. Such a user needs three things:

- A mechanism to file and handle received messages/documents
- An appropriate document editor (e.g., a multi-media document editor)
- A mechanism to address messages

The first two issues are local, so only the last issue need concern us here. There are a number of ways in which a user might address a message:

1. The most common will be by reference to a private directory (an electronic filofax). This will have its entries filled initially by one of the other methods of addressing a message.
2. From an existing message (typically replying).
3. By use of the directory. This might be by use of a special browsing interface, or by supplying a free form name something like a paper mail address, which might be taken from a business card. Addresses such as:

   Steve Kille, UCL, England
   S Kille, Univ College London, GB

   should be directly usable, perhaps with a little prompting to resolve ambiguities. If there is a need to specify rigid directory names, then the directory has failed. This example syntax is not chosen arbitrarily, as it is being proposed for use by directory interfaces for resolution of purported names [Kil90a].

An approach which is relatively common now is likely to become rarer. That is, specifying the address directly either as an RFC 822 address (e.g., S.Kille@cs.ucl.ac.uk) or the uglier X.400 form shown earlier. This type of information can increasingly be kept invisible, in the same way that DTE numbers and IP addresses usually are now.

3. Managing Local Information

The managers view can now be considered. In many current systems, local information needed by the mailer is handled by editing entries into a number of local files (e.g., */etc/passwd,* and alias file). This is often done in a very ad hoc manner. It would be useful to systematically manage such information in an appropriate database. The OSI Directory is a useful database for this purpose, as it gives a suitably rich data model, and defines open access protocols, so that information can be shared easily. A model for the objects in the OSI Directory is now described, which shows in broad terms how the external information will be represented.

The following basic principles are adopted:

- All information managed should be meaningful in human/administrative terms.
- There should be mechanisms to enable (but not require) user hostile information such as user ids to be assigned automatically.
- Information should not have to be entered manually in more than one place.

To manage a local mail system, the following types of object are needed:

User
An entry representing a single human user. This will typically be named in an organisational context. For example:

Steve Kille, Computer Science, University College London, GB

This entry would have associated information, such as telephone number, postal address, and mailbox.

Machine
An entry representing a machine. This would be the name of a machine within an organisation. For example:

glenlivet, Computer Science, University College London, GB

The objects would be typed, and so there would be no possibility of confusing machines with users. This would contain information about the machine, such as its make, and operating system version.

Machine Group
A collection of machines grouped together for administrative purposes. The entry would be named appropriately:

OSI Project Machines, Computer Science, University College London, GB

This entry would contain the (Directory) names of all of the machines in the group.

Account

An account defined on a collection of machines. Where an organisation has unique accounts, this might be named in the context of the organisation:

steve, Computer Science, University College London, GB

Alternatively, it could be named relative to a machine or machine group:

root, glenlivet, Computer Science, University College London, GB

The account entry would contain information such as user id, and home directory. Directory procedures could be used for account authentication. The entry would contain the (directory) names of associated users. This might be zero (e.g., root), one (e.g., a normal user), or many (e.g., a shared account).

MTA

A Message Transfer Agent. In general, the binding between machines and MTAs will be complex. Often a small number of MTAs will be used to support many machines, by use of local approaches such as NFS. The MTA will identify its OSI location.

Mailbox

This defines the User Agent (UA) to which mail may be delivered. This will define the account with which the UA is associated, and may also point to the user(s) associated with the UA. It will identify which MTAs are able to access the UA.

Role

Some organisational function. For example:

System Manager, Computer Science, University College London, GB

The associated entry would indicate the occupant of the role.

Distribution Lists

There would be an entry representing the distribution list, with information about the list, the manger, and members of the list.

Special Services

Message systems are used to support a number of services, such as retrieval of information (Info Servers). These services should also be registered in the Directory.

It is now useful to describe how a message might be handled:

1. A directory name should be identified. This might be known already or identified by searching/approximate matching in the directory. This procedure can be used for local or remote addresses.
2. In some cases, the directory name will be dereferenced onto one or more other directory names. Possible cases are:
   - Mapping a role onto a role occupant
   - Expansion of a distribution list
   - A user specified redirect to a different user

   This may happen recursively, for example where a role is a member of a distribution list.
3. Each directory name will be mapped onto the mailbox (User Agent).
4. The case of remote User Agents is considered in the next section (routing). For local User Agents, an account is identified, which will show where to deliver the mail.

The generality of these bindings is important to reflect real organisational complexities. There are cases where all of these separations are needed to support a natural naming structure. However, there is a very common case where the User, Mailbox, and account have a 1:1:1 mapping. To support this, it is important that all of these objects can be represented by a single entry. This allows the manager to create a single entry for a user, and assign mailbox and account information to that entry. It means that multiple entries in the database need only be created where there is real information contained in the distinction (e.g., a user with a mailbox in a different organisation).

The issue of typing is interesting. The OSI Directory uses strongly typed information. This is important for the manager as a consistency check. It is useful to be able to isolate different types of entry, and not simply rely on implicit semantics (the case in most mail addresses at present). Whilst a (typeless) alias file is suitable for a small organisation, it gets rapidly out of hand as the system grows. However, it is important that this typing is made invisible to the end user. Users do not like redundant typing, and should be able to access the directory as if it supported typeless naming. The User Friendly Naming mechanism proposed by the author is one means of achieving this.

The major benefit that the user gets from this is that information on users of the Message Handling Service is now available, and the database (Directory) can be searched by tools appropriate to the user's needs. This approach means that the user does not need to know magic incantations to send mail. The system manager gains, as information can be managed in a more systematic manner. Because of the richer information framework, additional information can be stored in the same place. This should lead to the situation where information is stored in one place only, rather than being replicated in different places for different purposes.

4. Managing Routing

Non-trivial configuration of routing for current message handling systems is a black art, often involving gathering and processing many tables, and editing complex configuration files. Many problems are solved in a very ad hoc manner. Managing routing for MHS is the most serious headache for most mail system managers. Services which need to be considered include:

- Routing between multiple local MTAs (Message Transfer Agent), which may have a complex mapping onto hosts
- Access to external networks, offering different services, with different protocols and addressing structures
- Access to services such as Fax
- Managing authorisation and accounting

Two extreme approaches to routing are:

- High connectivity between MTAs. An example of this is the way the Domain Name Server system is used on the DARPA/NSF Internet. Essentially, all MTAs are fully interconnected.
- Low connectivity between MTAs. An example of this is the UUCP network.

In general an intermediate approach is desirable. Too sparse a connectivity is inefficient, and leads to undue delays. However, full connectivity is not desirable, because of the need for Organisations and Management Domains to remain autonomous. The reasons for this intermediate approach as the only realistic approach are discussed in COSINE STUDY [Kil88a].

Management considerations will lead to autonomous systems, with planned interconnection. Typically, an Organisation will provide a single entry point, and hide internal structure. Groups of Organisations will, as a Management Domain, have an internal agreement to use open connection. Such a group might be the UK Academic Community.

For each MTA, there will be two types of agreement:

Bilateral Agreements
: An agreement between a pair of MTAs to route certain types of traffic.

Open Agreements
: An agreement between a collection of MTAs to behave in a cooperative fashion to route traffic. This may be viewed as a general bilateral agreement.

It is important to ensure that there are sufficient agreements in place for all messages to be routed. This will usually be done by having agreements which correspond to the addressing hierarchy. This leads to MTAs at the top of the tree being fully connected, and MTAs lower down have a default MTA to route unknown traffic to. Other agreements may be added in to improve the efficiency of routing.

We can now consider how these agreements are represented in the Directory. A bilateral agreement is represented by an entry associated with each MTA. Each party to the agreement will set up the entry which indicates policy. The fact that these correspond is controlled by the external agreement.

An open agreement is represented as a part of the Directory, containing all of the MTAs participating in the bilateral agreement. This might be within a Management Domain named in the Directory. This subtree indicates the participating MTAs, and the role that they play. Each MTA has pointers to the set of open agreements in which it participates. Usually, an open agreement will require authentication that each MTA is a genuine participant in the agreement.

This layout is one of several possible mappings onto the directory. It has a large advantage that it can be configured so that all of the information needed by an MTA to make a routing choice can be replicated systematically into a local DSA. This will allow routing to be done in an efficient and robust manner.

5. Conclusions

This paper suggests an *external* model for the integration of Message Handling and Directory. It has been shown how the system can be structured in order to hide much of the underlying complexity, and to make the control of data reasonable in terms of the task being tackled, and how this will provide much useful functionality. Parallel work is being undertaken on the specification of detailed mechanisms, and to implement this in the context of the PP Message Handling System and the QUIPU Directory System [Kil88b, Kil88c]. This paper is intended to solicit comments on requirements, and so feedback is encouraged.

References

[CCI88a] CCITT/ISO, "CCITT Recommendations X.400 / ISO IS 10021-1," Message Handling: System and Service Overview (December 1988).

[CCI88b] CCITT/ISO, "The Directory - Overview of Concepts, Models and Services," CCITT Recommendation X.500 / ISO DIS 9594-1 (May 1988).

[Kil88a] S.E. Kille, "Topology and Routing for MHS," Cosine Study 7.7 (June 1988).

[Kil88b] S.E. Kille, "PP - A Message Transfer Agent," *IFIP WG 6.5 Conference on Message Handling Systems and Distributed Applications*, North Holland (October 1988).

[Kil88c] S.E. Kille, "The QUIPU Directory Service," *IFIP WG 6.5 Conference on Message Handling Systems and Distributed Applications*, North Holland (October 1988).

[Kil90a] S.E. Kille, "Using the OSI Directory to achieve User Friendly Naming," UCL Research Note RN/90/29. (June 1990).

Extending 4.3BSD UNIX to Support Greek Characters

Achilles Voliotis
Alexios Zavras

National Technical University of Athens
Greece

ABSTRACT

The UNIX system, traditionally suited for English-speaking environments only, has been modified to support Greek characters. This modification actually involved much work that should not have been necessary if many programs had made less assumptions. The current state of the project and the changes performed are presented, without too much technical detail. Finally, we present a discussion about the implementation of a truly language-independent UNIX system.

Introduction

The UNIX system, from the first years of its design, was oriented towards English-speaking users and provided an English-based environment. (Even the original excuse that Thompson and Ritchie gave at Bell Labs for being allowed to work on UNIX, was that of building an [English] word-processing system.) With the success and spread of UNIX throughout the world, it is nowadays being used by many people whose native language is not English. No particular attention is given to these non English-speaking users: all the commands are, beside cryptic, English-based, the manual pages are in English, etc.

The situation is even worse when one considers the total lack of support of non Latin-based languages, such as Greek. Not only is the UNIX system based on the English language, it also provides no way of dealing with characters of another language.

As undergraduate computer science students in the National Technical University of Athens, we decided to eliminate this drawback and set off to modify the UNIX system to support Greek characters.

The Greek Characters

The International Standard for the representation of Greek characters specifies that each character should be stored in an 8-bit byte. As it can be seen in table 1, the standard is actually an extension of the usual ASCII representation, with the Greek characters occupying the positions 182 to 254 (decimal), i.e., they have their high (eighth) bit set. Due to this characteristic, these characters will be often referred to as 8-bit characters.

One would expect that since the character representation is the same as the ASCII characters, few programs, if any, would have any problems dealing with Greek characters. Unfortunately, this is not so.

Our experiments showed that the majority of programs assume that the traditional ASCII characters (with codes less than 128 decimal) are the only ones that can ever be encountered. In order to have a complete Greek UNIX environment, these programs had to be modified and new programs written.

Input

The first problem was to be able to input such characters from a ordinary keyboard. We developed a program similar to `script`, appropriately called `gscript`. The user always runs an application (usually a shell) inside this program. The programs monitors the user input and passes it to the application that is running underneath, exactly like `script`. Its special feature is that upon a user-entered special character (e.g. control-A) or sequence of characters (e.g. the one sent by the "`Do`" key of DEC VT220 terminals), it toggles between Latin and Greek mode. When in Greek mode, the user input, before it is passed to the application, gets converted to its Greek counterpart, e.g. the '`a`' character gets converted to 'α', '`b`' to 'β', etc. The correspondence between Latin characters pressed and Greek characters produced is consistent with the layout of the Greek typewriter, so that a person knowing how to type would have no problem using the system.

				b8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				b7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b4	b3	b2	b1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0	0	0	0	00	□	□	SPC	0	@	P	‘	p	□	□	□	°	ΐ	Π	ΰ	π
0	0	0	1	01	□	□	!	1	A	Q	a	q	□	□	“	±	Α	Ρ	α	ρ
0	0	1	0	02	□	□	"	2	B	R	b	r	□	□	”	²	Β	□	β	ς
0	0	1	1	03	□	□	#	3	C	S	c	s	□	□	Δρχ	³	Γ	Σ	γ	σ
0	1	0	0	04	□	□	$	4	D	T	d	t	□	□	□	΄	Δ	Τ	δ	τ
0	1	0	1	05	□	□	%	5	E	U	e	u	□	□	□	΅	Ε	Υ	ε	υ
0	1	1	0	06	□	□	&	6	F	V	f	v	□	□	|	Ά	Ζ	Φ	ζ	φ
0	1	1	1	07	□	□	’	7	G	W	g	w	□	□	§	·	Η	Χ	η	χ
1	0	0	0	08	□	□	(	8	H	X	h	x	□	□	¨	Έ	Θ	Ψ	θ	ψ
1	0	0	1	09	□	□	)	9	I	Y	i	y	□	□	©	Ή	Ι	Ω	ι	ω
1	0	1	0	10	□	□	*	:	J	Z	j	z	□	□	□	Ί	Κ	Ϊ	κ	ϊ
1	0	1	1	11	□	□	+	;	K	[	k	{	□	□	<<	>>	Λ	Ϋ	λ	ϋ
1	1	0	0	12	□	□	,	<	L	\	l	|	□	□	¬	Ό	Μ	ά	μ	ό
1	1	0	1	13	□	□	-	=	M	]	m	}	□	□	□	½	Ν	έ	ν	ύ
1	1	1	0	14	□	□	.	>	N	^	n	~	□	□	□	Ύ	Ξ	ή	ξ	ώ
1	1	1	1	15	□	□	/	?	O	_	o	□	□	□	—	Ώ	Ο	ί	ο	□

Table 1: *The ISO–4873* ΕΛΟΤ–*928 character set*

Another special feature of the gscript program is that it also performs multiple-character conversions. This is necessary in order to handle Greek letters with accents or diaeresis signs. For example, the user types '' a' to get 'ά'.

Unfortunately the terminal driver on the kernel makes some assumptions on the nature of the characters, so it had also to be modified to allow 8-bit clean communication. We deliberately did not put all of the gscript functionality inside the terminal driver, accepting the extra running process for the sake of simplicity.

The X Window System interface for the support of Greek characters was developed when one of the authors (Zavras) first encountered the system while being a (post-)graduate student in the University of Wisconsin – Madison. Under the X Window System, there is no need for a process similar to gscript, since xterm (the terminal emulator) already performs the function of reading the keyboard and passing the user input to the application running underneath. Once again, assumptions about the nature of the characters made modifications to the code for the terminal emulator program (xterm) in the release 3 of the X Window System necessary. Fortunately, in the subsequent release 4 the emulator was 8-bit clean, therefore only the mappings between the keys pressed and the Greek characters produced needed to be specified.

Output

Of course, we still had to find a way to see the Greek characters on the screen and on printouts. On usual display terminals, which allow only 127 characters per font but multiple fonts (such as the DEC VT220 that we used in the National Technical University of Athens), a font consisting of Greek characters is downloaded at the beginning of the gscript run. From then, each time there is a change of font, gscript sends the appropriate escape sequence to the terminal to select the primary or the downloaded font. In the X Window System, the fonts contain 256 characters, so there is no need for escape sequences.

We are currently using a number of different fonts with Greek characters. Historically, the first one we designed was a downloadable font for a dot-matrix printer. This was later transformed to a downloadable font for the DEC VT220 terminals. The font was later added as a supplement to the existing 9x15 font for the X Window System. The resulting font was named g9x15 (for lack of a better name), and is the one used the most.

There are currently three more fonts available to be used on the X Window System. All three are extensions to various Sun fonts, namely `gallant.19`, `screen.roman.14`, and `screen.bold.14`. Since all these fonts were designed by Computer Scientists on small grids, their quality leaves a lot to be desired. All the fonts are displayed in the Figures 1 to 4 at the end of the paper.

To get the Greek characters printed out, we use a downloadable font on dot-matrix printers, and the Symbol font on Postscript laser printers (which is more or less standard, i.e., present on all Postscript laser printers). Through transformations in Postscript, one is also able to obtain slanted ("italic") versions.

Storage

The intuitively correct way to store the information in a file is to represent a Greek character as its standard representation, in the same way that the regular ASCII characters are handled. This method was actually chosen over the one modeled after the way `gscript` presents the output, i.e., keeping only seven-bit characters, and some special sequence to toggle between Greek and Latin modes.

Use

Even after ways of entering, viewing and storing Greek characters were implemented, the need to actually perform something useful with them was still present. As was mentioned earlier, many of the UNIX programs can not handle 8-bit characters, so we had to make modifications on a surprisingly large number of programs.

We were seeking a uniform solution, rather than individual fixes on the source code for each program. Among various proposed solutions, we finally chose to modify (extend) the `ctype` part of the C library. We provided the appropriate data in the C library and many new macros in `/usr/include/ctype.h`, like `isgreek(c)`, `isvowel(c)`, `togreek(c)`, etc. Having created this library, the modifications on most of the programs were straightforward. Of course, all of the changes in the source code were enclosed in `#ifdef GREEK`, so that the programs could be used (after re-compilation) in a different environment.

We present here the basic ideas behind modifications on a selected number of programs:

`cpp`
: In order to be able to write C programs that made use of the Greek characters, we decided to modify the C compiler. The easiest way to achieve the result was to limit the modifications to the C preprocessor. It was modified to accept strings with Greek characters and change them to their octal values (e.g. "αβγ" to "`\341\342\343`"), so that programs can display messages in Greek. Furthermore, comments in programs can be written in Greek.

`csh`
: Although it seems unbelievable, `csh` actually *uses* the eighth bit itself (to distinguish if any special meta-character is quoted, and thus deprived of its special functions). Therefore, the modifications were rather complicated, and involved the mapping of the quoted meta-characters to the unused positions of the upper half of the ASCII table.

`vi`
: The editor was modified to deal with Greek characters, in the same way it deals with Latin ones. All the commands are executed from either Latin or Greek mode (e.g. typing "δδ" in command mode deletes the current line). A couple of new commands were added, to convert Latin to Greek and vice-versa, and to add or remove an accent, when the cursor is on a Greek vowel (they both work as toggles, in the way the '`~`' command does). One of the biggest problems was to actually find keys that were not already bound to a function.

`nroff, troff`
: Both formatters were modified to accept Greek characters for input. Since all characters in `nroff` have constant width, there were no special modifications needed. In `troff` the characters are transformed to the special font (e.g. 'α' to "`\(*a`") and dealt in that form. Slanted Greek characters are produced by using the "`\S`" (slant) feature of `ditroff`. The extremely hard problem of hyphenation in the Greek language was limited by handling only the simplest cases when Greek text is processed.

From the modifications presented above, it can be easily seen that one can edit a file and write Greek text (seeing the characters in the screen), then format it and print it.

Excess

Once we knew the changes needed to support Greek characters we couldn't restrain ourselves: we sought and modified every piece of code in the UNIX system that places restrictions on the nature of the characters.

For example, the file system code in the kernel does not allow filenames to include 8-bit characters. Therefore, we had it "fixed" (and some programs like `fsck` and `dump`). Using links (and/or shell aliases) to map commands and files in general to Greek counterparts a user can now work in an environment without ever using Latin characters!

We also have a version of the C language with all the keywords changed into their Greek translations. This language also allows Greek characters outside strings, i.e., for identifiers. Needless to say, this language was never really used.

Filters

Even though all the previous mentioned modifications provided us with a rather complete Greek environment on one site (host), there was still problem of communicating with other sites that had not the same modified programs. For a typical example, the `sendmail` program clears the eighth bit of any character that sends. Therefore, we had to create a way to communicate Greek characters using only the regular ASCII set. (Fixing `sendmail`, of course, is not the solution, unless all the sites in the world use this new version, and maybe not even then).

In order to accomplish this, a variety of filters was developed. They mainly insert some kind of toggle-mode sequence in the text (e.g. the characters "`\fG`" in a `troff`-like way to denote a Greek font) and encode the Greek characters in a special way. This encoding is very important, since we do not only want any mapping, but one that leaves the characters recognizable without the reverse decoding programs. Therefore, it is not enough to merely strip the eighth bit, since this would map the Greek letter "Λ" to "K". Instead, "Λ" has to be mapped to "L". This was essential, since users on other sites may not have the capability of displaying the received text in its original Greek form, and should be able to understand its contents. These scripts (and the ones that perform the reverse transformations) are automatically invoked (e.g. when sending and receiving mail) and if both communicating ends have the software, the users are under the impression that there was a complete 8-bit clean exchange of messages.

Only after completing these filters could we talk about a Greek environment, not isolated from the rest of the UNIX systems, but communicating with them.

Language Independence

We present here some thoughts on the present and the future of truly language-independent applications on a UNIX system.

Our work has started almost two years ago and has taken a great deal of our free time as undergraduate and (post-)graduate students. When we started this project, we had only encountered the 4.2BSD UNIX, which provided no way of handling Greek characters.

Since then, many things have changed. We have witnessed a true effort for internationalisation, and the current state of most vendor-supplied UNIX systems allows input and output of 8-bit characters. Even the terminal driver of the 4.4BSD is "8-bit clean." We can, therefore, have an optimistic view for the future.

However, on the part of handling the information (as opposed to just acquiring it), the future may not seem so bright. In at least two vendor-supplied UNIX systems, the only use for 8-bit characters is for message printing, and the only use of the language specification (performed in the `csh` by something like `setenv LANG GREEK`) is to determine the date and monetary formats. Even worse, we have seen at least one version of `vi` that allows editing of 8-bit characters, but only when the first two characters of the file are usual ASCII. (Needless to say, this kind of half-fixed systems are the worse, since they usually look like truly 8-bit systems until you encounter the pitfalls.)

What should be done? The information about characters comes from the `ctype` library, which must be complete for any language. For example, only after trying to built an application using the `ctype` library for Greek characters we found out the need for macros like `is_accented(c)`, `add_accent(c)` and `rm_accent(c)`.

This information is made available at the program at compilation time, which creates the need of re-compiling all the programs when an addition or change to the `ctype` library is performed (or, of course, a different language is needed).

The situation is even worse in the case where one wants information not about a single character, but about a group of them (a word). The best example of such a need is hyphenation handling in `troff`. The `ctype` library is completely inadequate for such a task, and a completely new source of information should be designed and used.

Maybe, with the help of language specialists, a completely different approach, using something like `/etc/langcap` (à la `/etc/termcap`, `/etc/printcap`, etc.), should be developed. The programs then would be able to get the information needed at run-time, avoiding re-compilations.

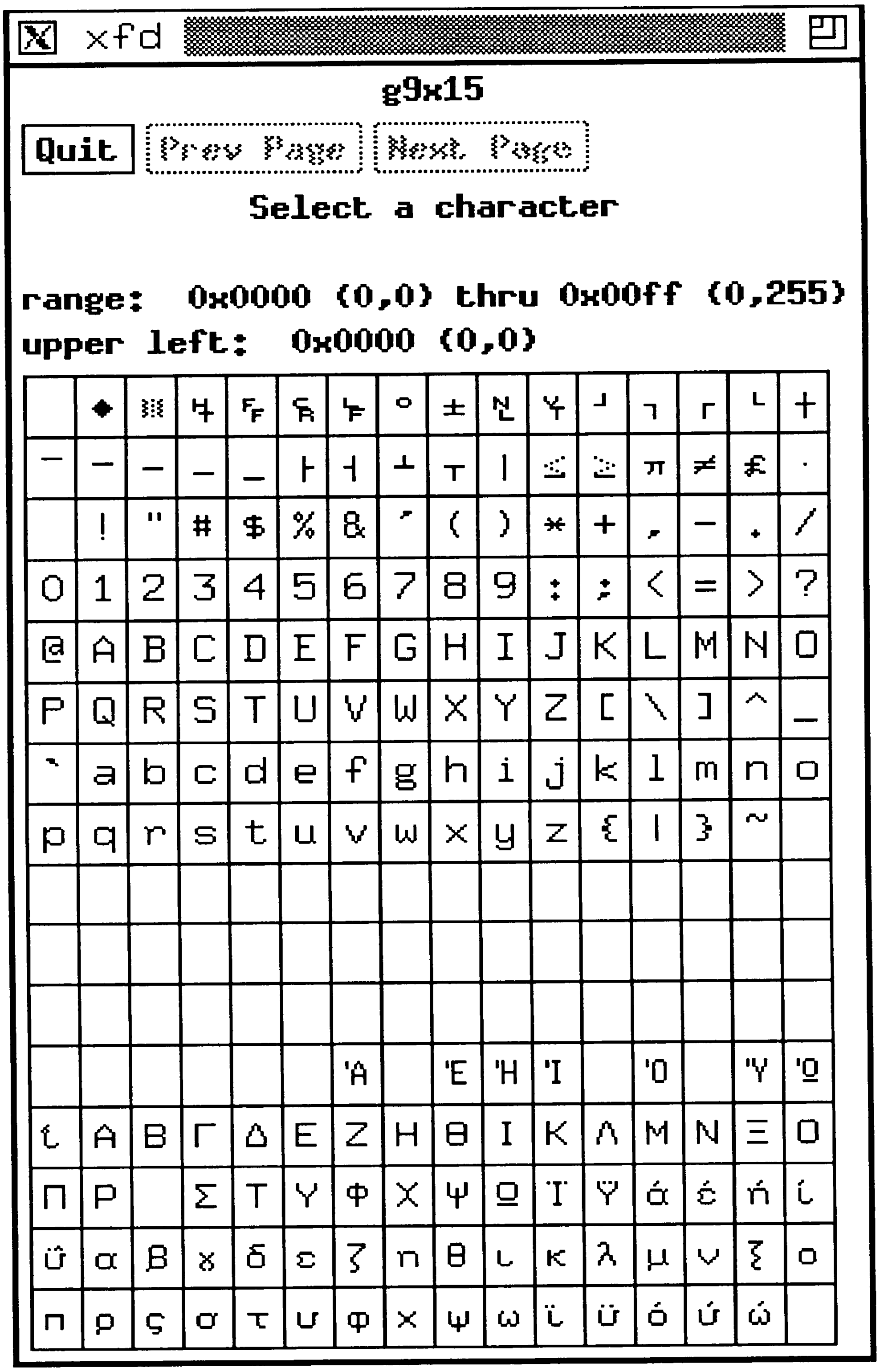
Acknowledgements

Over the years, many Greek students have worked on providing a complete Greek UNIX environment. Their efforts provided us with helpful ideas, almost complete solutions, and dead-end avoidance.

We are especially grateful to Sarantos Kapidakis and Manolis Tsangaris, since they were the first to systematically explore the possibility of a Greek UNIX environment, and without them the presented work would have been impossible.

Relevant Standards

ISO 646	Information Processing – ISO 7-bit Input/Output Coded Character Set.
ISO 2022	Information Processing – ISO 7-bit coded character sets – Code extension techniques.
ISO 4873	Information Processing – 8-bit coded character set for information interchange.
ISO 6429	Information Processing – ISO 7-bit and 8-bit coded character sets – Additional control functions for character-imaging devices.
ISO 8859	Information Processing – 8-bit single byte coded graphic character sets.
ISO 8859-7	Information Processing – 8-bit single byte coded graphic character sets.



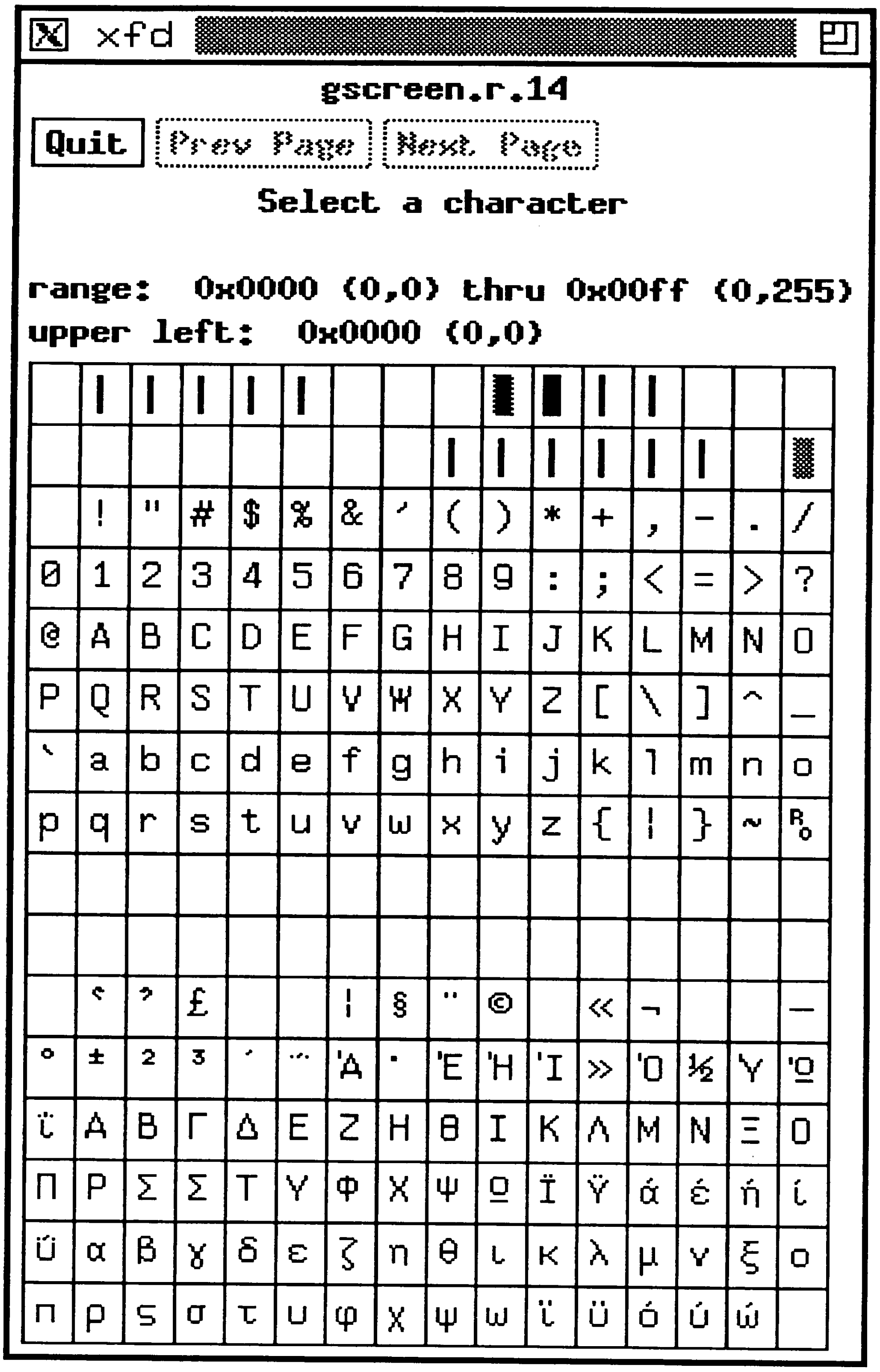
g9x15

Select a character

range: 0x0000 (0,0) thru 0x00ff (0,255)
upper left: 0x0000 (0,0)

	◆	▒	␉	␌	␍	␊	°	±	␤	␋	┘	┐	┌	└	┼
⎺	⎻	─	⎼	⎽	├	┤	┴	┬	│	≤	≥	π	≠	£	·
	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
						Ά		Έ	Ή	Ί		Ό		Ύ	Ώ
ΐ	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο
Π	Ρ		Σ	Τ	Υ	Φ	Χ	Ψ	Ω	Ϊ	Ϋ	ά	έ	ή	ί
ΰ	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
π	ρ	ς	σ	τ	υ	φ	χ	ψ	ω	ϊ	ϋ	ό	ύ	ώ	

Figure 1: *The* `greek-9x15` *font*



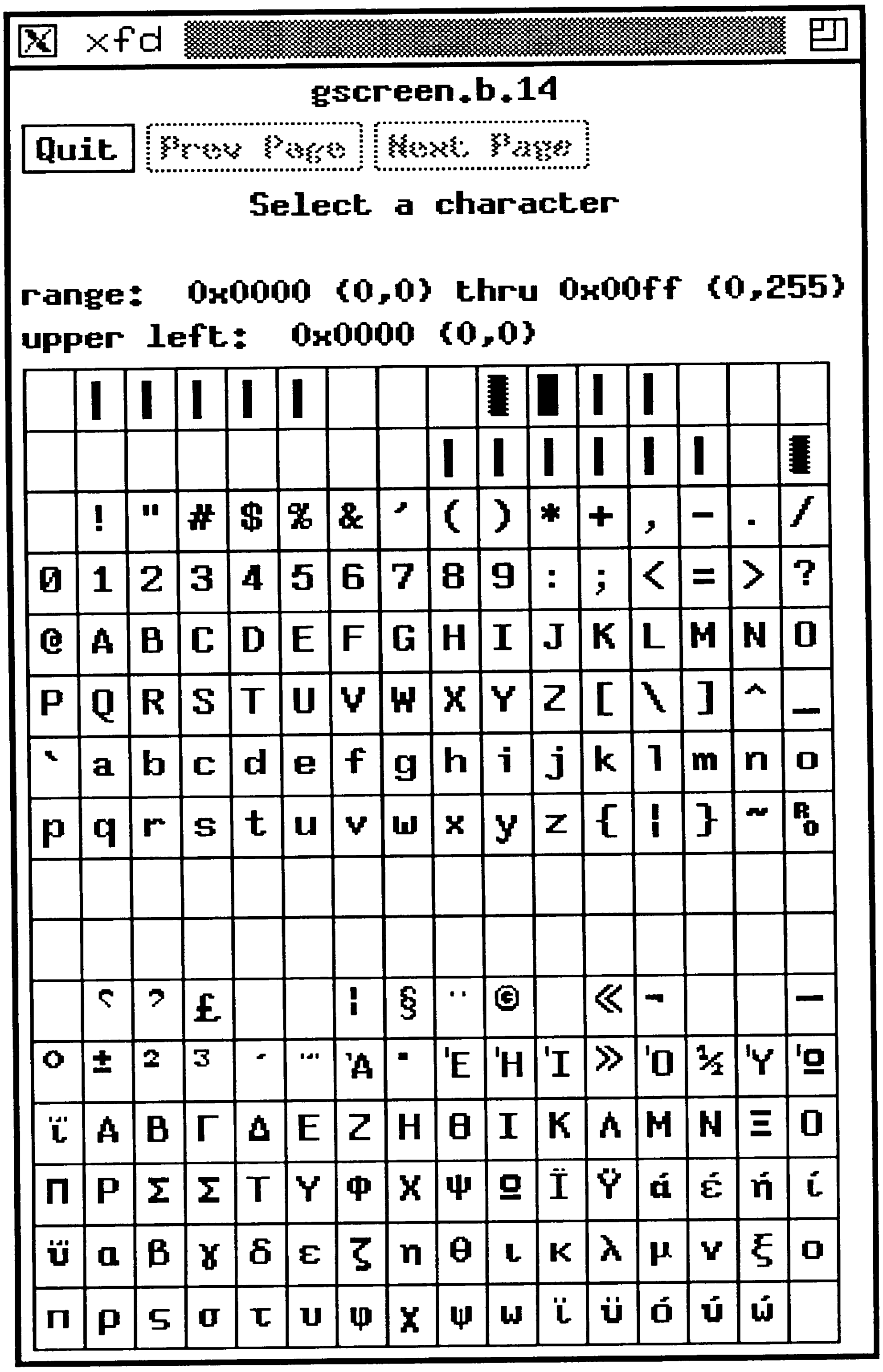
gscreen.r.14

Select a character

range: 0x0000 (0,0) thru 0x00ff (0,255)
upper left: 0x0000 (0,0)

	|	|	|	|	|				▓	█	|	|			
								|	|	|	|	|	|		▓
	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	℞
	ʽ	ʼ	£			¦	§	¨	©		«	¬			―
°	±	²	³	΄	΅	Ά	·	Έ	Ή	Ί	»	Ό	½	Ύ	Ώ
ΐ	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο
Π	Ρ	Σ	Σ	Τ	Υ	Φ	Χ	Ψ	Ω	Ϊ	Ϋ	ά	έ	ή	ί
ΰ	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
π	ρ	ς	σ	τ	υ	φ	χ	ψ	ω	ϊ	ϋ	ό	ύ	ώ	

Figure 2: *The* `greek-screen.roman.14` *font*



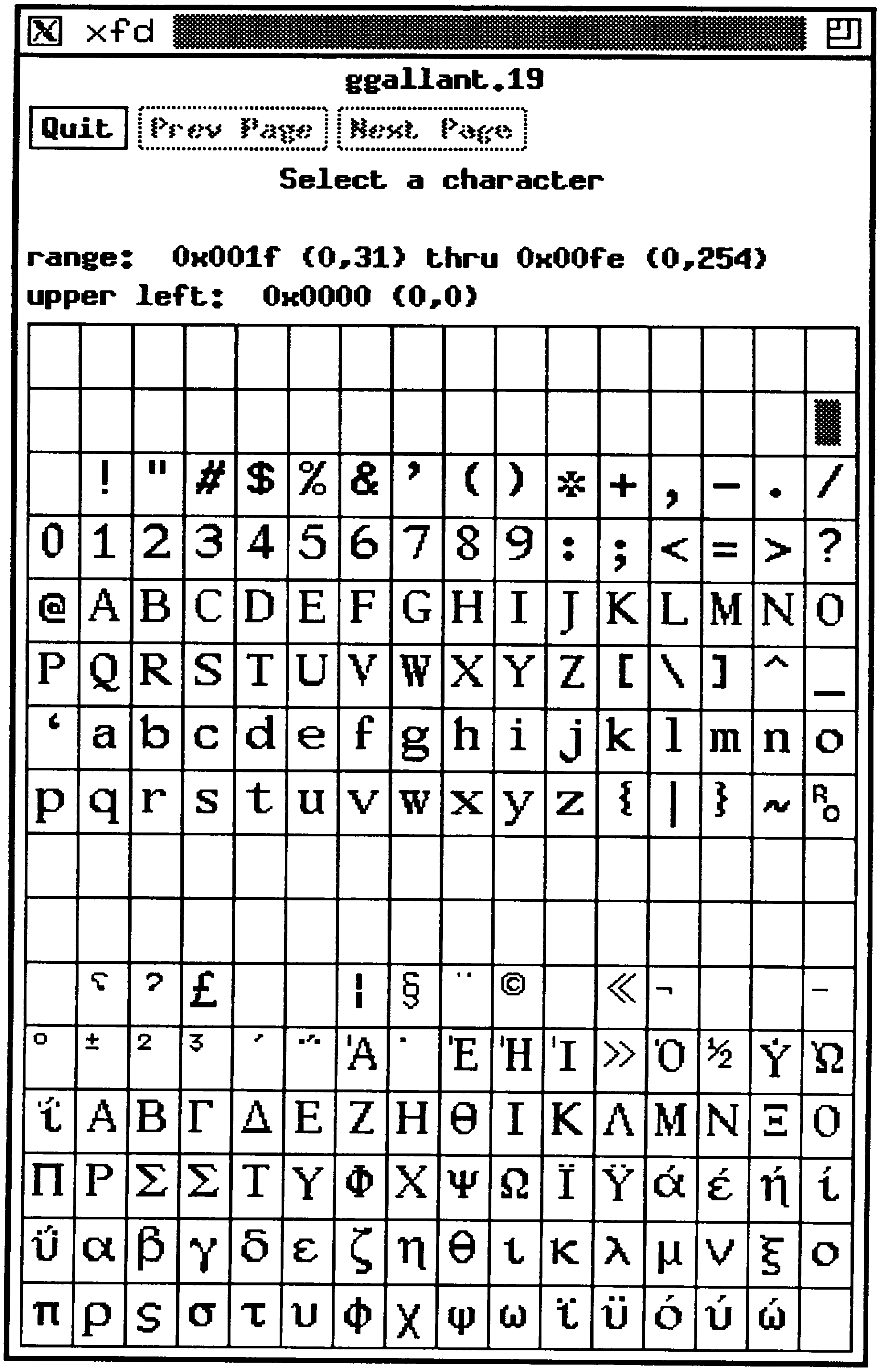
gscreen.b.14

Select a character

range: 0x0000 (0,0) thru 0x00ff (0,255)
upper left: 0x0000 (0,0)

	▌	▌	▌	▌	▌				▓	█	▌	▌			
								▌	▌	▌	▌	▌	▌		▓
	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	RO
	ʽ	ʼ	£			¦	§	¨	©		«	¬			―
°	±	²	³	΄	΅	Ά	·	Έ	Ή	Ί	»	Ό	½	Ύ	Ώ
ΐ	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο
Π	Ρ	Σ	Σ	Τ	Υ	Φ	Χ	Ψ	Ω	Ϊ	Ϋ	ά	έ	ή	ί
ΰ	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
π	ρ	ς	σ	τ	υ	φ	χ	ψ	ω	ϊ	ϋ	ό	ύ	ώ	

Figure 3: *The* `greek-screen.bold.14` *font*



xfd

ggallant.19

Quit | Prev Page | Next Page

Select a character

range: 0x001f (0,31) thru 0x00fe (0,254)
upper left: 0x0000 (0,0)

															▓
	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	RO
	ʽ	ʼ	£			¦	§	¨	©		«	¬			―
°	±	^{2}	^{3}	΄	΅	Ά	·	Έ	Ή	Ί	»	Ό	½	Ύ	Ώ
ΐ	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο
Π	Ρ	Σ	Σ	Τ	Υ	Φ	Χ	Ψ	Ω	Ϊ	Ϋ	ά	έ	ή	ί
ΰ	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
π	ρ	ς	σ	τ	υ	φ	χ	ψ	ω	ϊ	ϋ	ό	ύ	ώ	

Figure 4: *The* `greek-gallant.19` *font*

The Serpent UIMS[†]

Erik J. Hardy
Daniel V. Klein

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213 USA
erik@sei.cmu.edu
dvk@sei.cmu.edu

ABSTRACT

Serpent represents a new generation of User Interface Management Systems which manage the total dynamic behavior of an interface and which allow applications to remain uninvolved with the details of the user interface. Serpent is designed to manage the specification and dynamic behavior of (relatively) arbitrary toolkits. It provides for a fixed application programmer interface across changes in toolkits. This allows an application to evolve from one toolkit to another, or even to use multiple toolkits simultaneously.

Serpent is intended to be used either with an application (in a production environment) or without an application (in a prototyping environment). Prototypes can be built interactively, tested, and rebuilt very quickly. The dialogue description language is sufficiently rich to support the prototyping of reasonably complex user interfaces. This does not preclude, however, the construction of significantly more complex interfaces when used with an application. Applications and toolkits written in either C or Ada can be used with Serpent, although the interface description mechanism is designed to be extensible to other languages.

Two portions of Serpent of particular note are presented: Glue, which provides a mechanism for importing any X widget based on the Intrinsics, and the Dialogue Editor, which allows for the interactive construction of Serpent dialogues.

1. Introduction

It has been shown that problems exist in current methods for the specification and development of user interfaces. The evolutionary development approach has proven effective in mitigating the risks associated with the development of large, complex systems, particularly for the user interface portion of such systems [Bas90a, Boe88a, Fis87a].

The explosion in workstation capabilities in the last few years has sparked many new ideas about how to use these capabilities for user interface development [Mye88a, Mye88b, Kol87a, Col88a, Fol89a, Kas89a], leading to a multitude of tools and environments, such as Prototyper, XVT, UIL, Granite, Autocode, and MIKE. However, each is marked by the use of a specific language and/or interactive tools tailored to the capabilities of a particular platform and/or to the specific user interface technology supported. Application support in these packages usually takes the form of a fixed set of functions that can be invoked as necessary by the application, or a set of functions that are dynamically generated by the prototyping tool to implement the user interface, particularly the dynamic behaviors of the user interface objects. In using these tools, if the user interface changes, the application must be changed to invoke the new functions, or the behaviors must be recoded.

User interface technology is evolving rapidly. Today's leading edge data presentation theory becomes tomorrow's commonplace toolkit, giving way to some previously unimagined technology. None of the above approaches adequately provides for the effective integration and use of new technologies.

† This work was sponsored by the U.S. Department of Defense.

2. Goals of a Modern User Interface Environment

In 1987 the Software Engineering Institute started the User Interface Project to address perceived problems in user interface development and to assist the transition of user interface design and development technology into practice. Out of this effort arose a set of goals for the next generation of user interface environments:

1. In any computer system, there should be a true separation of concerns between the application and the user interface. This is simply the concept of modularity: the application should not try to perform the functions of a user interface, and vice versa. One should be able to develop the application independently of the user interface, in a language appropriate to the semantics of the application; similarly, user interface development should be independent of the application, again in a language suitable to the semantics of user interfaces.
2. There should be a single, consistent application program interface (API), and the API should contain no preconceptions about the nature of the user interface. This allows the application to remain unchanged, even though the user interface may change.
3. The user interface specification, design, and implementation should be simple and straightforward; prototyping should be fairly easy using the mechanisms provided by the environment. Non-programmers should be able to perform these activities with a minimum of training. The mechanisms used to perform these activities should not have to change, even though the user interface style or underlying user interface technology may change.
4. It should be possible to prototype the major portions of the functionality of a system without an application. The user interface support mechanisms should be sufficiently rich to support reasonably sophisticated prototypes. As the prototype matures, facilities should be provided to add an application, in part or in its entirety, thus allowing for evolutionary development. This allows a completed prototype to be included as part of the deliverable system.
5. Existing systems should be able to take advantage of new technologies as they become available, without affecting the application portion of the system. The mechanisms for incorporating new technologies should be relatively simple.
6. Performance, when the environment is used strictly as a prototyping vehicle, should be reasonable, although special performance considerations would have to be made when used in a production environment.

3. Serpent

Starting with the above goals, the User Interface Project developed a user interface environment known as Serpent. Serpent is a user interface management system (UIMS), constructed upon the standard Seeheim model [Pfa85a], that supports the development and execution of the user interface of a software system. Serpent supports incremental development of the user interface from the prototyping phase through production to maintenance. Serpent encourages the separation of concerns between the user interface and the functional portions of an application. Serpent is easily extended to support multiple input/output technologies.

3.1. Architecture

The figure shows the overall architecture for Serpent. The architecture is intended to encourage the proper separation of functionality between the application and the user interface portions of a software system. The three different layers of the architecture provide differing levels of control over user input and system output. The presentation layer is responsible for layout and device issues. The dialogue layer specifies the presentation of application information and user interactions. The application layer provides the actual system functionality.

The *presentation layer* controls the end user interactions and generates low-level feedback. This layer consists of various input/output technologies that have been incorporated into Serpent. A standard interface has been defined, which simplifies adding new technologies. Each technology defines a collection of *objects* which are available for presentation to, and interaction with, the end user.

The *dialogue layer* specifies the user interface and provides the mapping between the presentation and application layers. The dialogue layer determines which information is currently available to the end user and specifies the form that the presentation will take as previously defined by the dialogue specifier

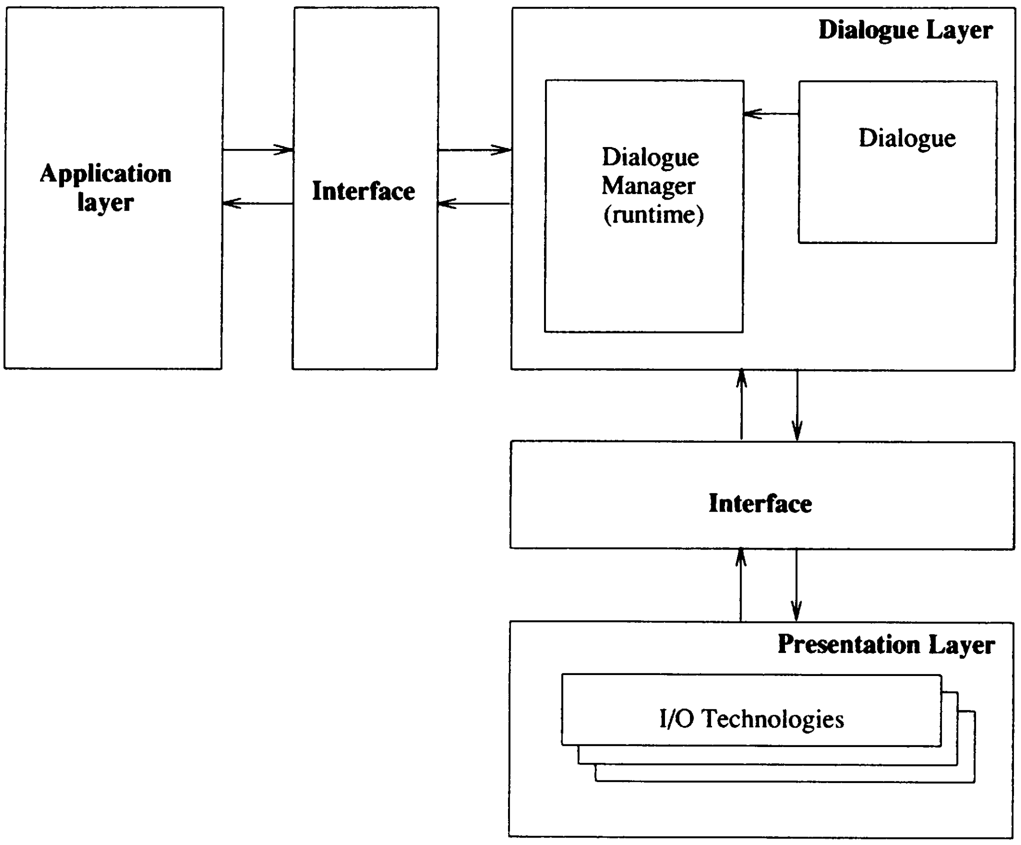


Figure 1:

(individual responsible for creating the user interface specification, or *dialogue*). The dialogue layer acts like a traffic manager for communication between application and technologies. The *presentation layer* manages the presentation; the dialogue layer tells the presentation layer what to do. For example, the presentation layer manages a button that the end user can select; the dialogue layer informs the presentation layer of the position and contents of the button and will act when the button is selected.

The *application layer* performs those functions that are specific to the system's core functionality. Since the other two layers are designed to take care of all the user interface details, the application can be written to be presentation-independent; there need not be any dependencies in the application on a specific technology.

The data that is passed between different layers is known as *shared data*. A *shared data definition* provides the format of the data interface.

3.2. Slang

In Serpent, user interface dialogues are specified in a special-purpose language called *Slang*. Slang provides a mechanism for defining the presentation of information to, as well as interactions with, the end user, without enforcing specific user interface policies. A Slang program enumerates a collection of interaction objects and allowable actions to be available to the end user.

The interaction objects available to the dialogue writer are defined by the input/output technology. Each technology defines a set of primitive objects that may be used in a dialogue. Each object has a collection of *attributes* that define its presentation and a collection of *methods* that determine how the end user may interact with that object.

Slang also provides variables for intermediate storage and manipulation, along with a full complement of primitive arithmetic operations. The specification of an attribute or variable can take one of three forms: a constant value; an expression involving one or more attributes or variables; or a body of procedural code called a *code snippet*.

By contrast, a method usually takes the form of a code snippet that is executed once each time the method is selected.

When a variable or attribute is specified in terms of another variable or attribute, this sets up a *dependency* between the two. In Slang, dependencies between items set up constraints, i.e., when a data item is changed, any data items that are dependent on it are automatically re-evaluated. This important and powerful feature allows the dialogue writer to build complex, interdependent interaction objects simply by specifying a series of dependencies.

Slang also allows a dialogue writer to group arbitrary objects into a logical collection called a *view controller* that may be created or destroyed as a unit. Specifying a view controller in Slang defines a view controller *template*; each template has a *creation condition* that defines when an *instance* of the template should come into existence. The creation condition thus defines the existence criteria for the view controller's child objects. The existence of a view controller instance and its child objects can be controlled by the values of Slang variables or by the creation or destruction of application data. When a view controller's creation condition is no longer valid, it and its associated objects are destroyed. Multiple instances of a template may exist at any time.

A view controller may also have sub-view controllers. For example, a common condition under which a submenu may appear would be that its parent menu must also exist. Of course, the submenu must have also been selected from the parent menu; when the parent menu is removed, the submenu is automatically removed. This illustrates the power that may be obtained from sub-view controllers; it also illustrates that creation conditions may be arbitrarily complex.

3.3. Application Program Interface (API)

From the application developer's perspective, Serpent behaves like a database management system. Shared data may be manipulated in the database by the application, the presentation layer (usually in response to end user actions), or the dialogue (in response to actions within the dialogue).

The application can add, modify, or delete shared data. Information provided to Serpent by the application is available for presentation to the end user, although there is no requirement on the dialogue writer to actually present any particular application data item. The application has no interface to the presentation layer and therefore cannot affect how data is presented to the user. When end user actions cause the dialogue to change the application shared database, the application is automatically informed. In this sense, the application views Serpent as an *active database manager*. There are both synchronous and asynchronous mechanisms in Serpent for the application to be informed of database changes.

3.4. Saddle

The type and structure of data that is maintained in the shared database is specified in a *shared data definition* file, defined in a language called Saddle. This data definition corresponds to the database concept of *schema*. A shared data definition file is created once for each application and once for each technology that is integrated into Serpent. A shared data definition is required before an application can use Serpent.

The shared data definition file is processed to produce a language-specific description of shared data. Processors currently exist for Ada and C. If the application is written in C, the processor will generate structure definitions that can be included into the application program. If the application is written in Ada, the processor will generate package specifications. Processors for other languages which support complex data structures are easily generated.

4. Technology Integration

The process of integrating a technology into Serpent is conceptually simple. It can be logically divided into three parts:

1. the objects with which the end user will interact must be determined, along with their behaviors;
2. these objects must be defined to Serpent through the use of Saddle; and
3. a driver must be written to allow the technology to communicate with the dialogue manager, through Serpent's shared database facility.

Technology integration presents some practical difficulties. The Serpent model of a technology is that it consists of a set of objects; if the technology does not support that model, a set of objects and their attributes and methods must be defined and constructed, such that the objects together span the desired functionality of the underlying technology. The integrator also has to decide how much of the underlying technology to expose to a dialogue writer, whether to change any of the default behavior of the system, and whether to make the system more robust by, for instance, performing error checking that the technology does not handle.

Our experience in the past is that technology integration efforts, with respect to X-based widget sets, have been clumsy and time-consuming, due to the arcane nature of their associated toolkits and the differences between the toolkits. Each prospective widget set, as well as its associated toolkit, has to be learned from scratch; there are just enough similarities between them to make the debugging process difficult. Due to our limited resources, we had to make tradeoffs to determine which widget sets to integrate into Serpent, having necessarily to discard some.

5. Glue

With the imminent convergence of the major competing toolkits, specifically those of MIT, OSF, and XView, we have created a widget integration tool, called *Glue*. Glue allows any widget set based on the Xt Intrinsics to be quickly and easily incorporated into the Serpent presentation layer, thus making it available for use in a Slang program.

Since the primary technologies that we use are based on X, and since most X toolkits use the concepts of widgets, Glue was designed primarily to support the widget paradigm. The mechanism, however, does not place such a great onus on the technology integrator that other, disparate technologies and technology interfaces could not be used.

5.1. Equivalence Classes

Serpent supports six basic data types, namely: `integer`, `real`, `boolean`, `string`, `buffer`, and `ID`. Toolkits typically support a wide range of datatypes, expressible as either C primitive data types (such as `int`, `float`, `char`, etc.) or as constructions of these primitive types (i.e., arrays and structures). In order for Serpent to be able to operate on the toolkit data types, a mechanism is needed to equate each of the toolkit types with a Serpent type.

By providing an equivalence type, we allow the technology integrator to specify what a data type looks like on both sides of the Serpent/technology interface. Because all equivalence classes must be declared before they are used, and because each X type is mapped only once, it is possible to make global changes to the interface specification by simply changing an equivalence type. For example, if it was previously decided that the `Pixel` type could be represented as `string[80]`, and later it was found that in certain cases, 100 characters were required, changing the single equivalence type to `string[100]` would increase the allocation for all `Pixel` types and remedy the problem throughout the interface.

5.2. Widget Definitions

Each widget definition consists of a number of subcomponents. The term "widget" used here does not constrain the technology integrator to working with just widgets. Any technology can conceivably be integrated using this mechanism of describing visualized components of the technology. The widget definition contains a list of attributes of the widget. An attribute definition allows the technology integrator to specify the name and type of each attribute. Since most technologies have default values for each attribute, it is not neccessary to specify a default value; however, Glue allows the technology integrator to specify a default value which may be different from that provided by the technology.

The technology integrator may also customize the technology appearance by exposing as much or as little of the technology as is desired. New attributes that may not have been considered by the original toolkit builder may also be defined. An attribute may be "hidden" as well, simply by not specifying that attributes for a particular widget.

5.3. C Programs

Nothing in this world is perfect, and no technology integration can be completely seamless. When a technology integrator first begins to integrate a new toolkit or widget set, the routines to bind the Serpent method actions with the technology must be written. The Glue program cannot anticipate what technology-dependent actions must be performed for a method, so all it can do is generate the routine stubs for each method. It is up to the technology integrator to fill in the details.

6. The Serpent Development Process

Slang was designed explicitly for user interface specification. A Slang dialogue writer is not burdened with the technical and procedural details necessary to manipulate specific interaction objects; those details are hidden in the presentation layer. The dialogue writer merely specifies the objects that make up the user interface and indicates how they relate to one another and to the end user; the Serpent runtime system manages the interaction objects. The dialogue specifier needs to be familiar with the characteristics of various objects, such as knowing that an Athena label widget appears as a rectangle on the screen; however, the specifier does not need to know how to tell the X Toolkit or the X server how to display such a widget.

Slang dialogues can be executed without the benefit of an application, allowing one to build, test, and refine a prototype before designing and implementing the rest of the system. Often, however, a prototype requires the existence of some application functionality, if only to initialize display values. Slang's rich set of primitive operations allow the user interface designer to "mock up" application operations in the prototype dialogue. Once the prototype has been refined, the simulated application behavior may be removed from the dialogue and replaced with the real actions in the application program.

7. The Serpent Editor

Although Slang was designed to be easy to use and to allow the specification of user interfaces without the normal syntactic arcana associated with a traditional procedural language, it is nevertheless a new language, with its attendant syntax and peculiarities. An interactive editor that hides most of the details of the user interface specification has been designed and is nearing completion at the time of this writing. Early experience with the Serpent Editor indicates a significant reduction in the time required to specify a complete working user interface. More importantly, the specific syntax of Slang is hidden, thus freeing dialogue specifiers from the drudgery of coding and allowing them to devote their time instead to the overall conceptualization of the user interface.

8. Status

The initial implementation of Serpent was done under ULTRIX 2.2 on DEC microVAX II and III workstations. Serpent has now been ported to many popular UNIX platforms, including Sun, DEC, and HP machines; Serpent also runs on several UNIX-based RISC machines, including DECstations and SPARCstations. We expect porting to similar UNIX platforms to be relatively straightforward.

Applications may be written in either C or Ada, and simple mechanisms exist to extend Serpent to support other high level languages. Serpent was implemented predominantly in C, with additional support software written as shell scripts. The Serpent Editor is expected to be released 3Q 1990.

Currently, four different interfaces to the X Window System have been written for Serpent: one implements a subset of the Athena widget set, another uses the Athena widgets supplemented by specially developed widgets for drawing, and a third implements a subset of the Motif widget set. In addition, Lockheed's Softcopy Map Display System has been integrated. Other ongoing or contemplated integration efforts include an integrated video/graphics workstation, a video mapping system, an experimental gesturing device, and XView – Sun's implementation of the Open Look User Interface. Finally, a full Motif integration is in progress at the time of this writing and is expected early 3Q 1990.

Serpent source files and complete documentation (in Postscript format) are available from the Software Engineering Institue and from MIT through anonymous ftp. Serpent resides on `fg.sei.cmu.edu` (128.237.2.163), in the directory `/pub/serpent`, in the compressed tar file `serpent.tar.Z`; a README file is also resident there. Serpent is also available from `expo.lcs.mit.edu` (18.30.0.212), in the directory `/contrib`; the help file is named `serpent.README`.

9. Conclusions

As a result of our experiences in developing user interfaces with Serpent, we have concluded that Serpent offers the following advantages over other user interface development approaches:

1. The active database model for applications allows the true separation of application issues from user interface issues, thus ensuring modularity. Application writers are also free from the syntactic drudgery inherent in programming large, complex input/output technologies.

2. The constraint mechanisms implemented via automatic dependency updates ensure that all participants (application, dialogue manager, and presentation) are synchronized in terms of the state of the system.
3. Serpent's language-independent interface definition and inter-process communication mechanisms help in achieving modularity. Application developers are not constrained to work in a single language.
4. Serpent's technology integration support reduces the integration process to a series of concise, well understood steps. Once a particular technology is integrated, its objects are available for use in any dialogue.
5. Due to Serpent's inherent separation of concerns, system developers can experiment with different user interface styles, and even different technologies, without changing either the application code or the API. This also provides for the injection of new technologies and user interface paradigms into an existing system, while minimizing the system portions which are impacted.

Serpent has achieved the goals of a modern user interface environment set forth earlier. The user interface specification mechanisms are simple and direct; changes in the user interface are made easily, without changing the application. The application program interface is simple and easy to use and enforces a true separation between the application and the user interface portions of the system. Prototyping is accomplished rapidly, with reasonable provision for application functionality simulation. Serpent's technology integration mechanisms allow a new technology to be incorporated into Serpent easily without affecting the application. Finally, Serpent is itself a prototype, implementing the goals listed above. Although we would not yet recommend it for time-critical production environments, performance is quite reasonable, and improvements are ongoing.

References

[Bas90a] L. Bass, B. Clapper, E. Hardy, R. Kazman, and R. Seacord, "Serpent: A User Interface Environment," *Proceedings, 1990 USENIX Winter Conference* (1990).

[Boe87a] Barry W. Boehm, "Improving Software Productivity," *Computer* **20**(9) (September 1987).

[Boe88a] Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer* **21**(5) (May 1988).

[Col88a] Kate Colborn, "OSF Determines User Interface; Choices Could Affect the Development of Applications Software," *EDN* (December 1988).

[Fis87a] Gary E. Fisher, *Application Software Prototyping and Fourth Generation Languages,* National Bureau of Standards (May 1987).

[Fol89a] James Foley and et al., "Defining Interfaces at a High Level of Abstraction," *IEEE Software* (January 1989).

[Kas89a] David J. Kasik and et al., "Reflections on Using a UIMS for Complex Applications," *IEEE Software* (January 1989).

[Kol87a] Stan Kolodziej, "User Interface Management Systems," *Computerworld* (July 8, 1987).

[Mye88a] Brad A. Myers, *Tools for Creating User Interfaces: an Introduction and Survey,* Carnegie Mellon University (1988).

[Mye88b] Brad A. Myers, *The Garnet User Interface Development Environment: a Proposal,* Carnegie Mellon University (1988).

[Pfa85a] G. (Ed.) Pfaff, *User Interface Management Systems,* Springer-Verlag, Berlin (1985).

User Interface Builders: A Compromise Platform in the User Interface War

Zoltan Hidvegi

Videoton Development Institute
Budapest
Voroshadsereg u. 54
H-1021 Hungary
sztaki!vidi!hidvegi

ABSTRACT

Although X has become the dominant window system in the UNIX community the user interface competition is not over at all. In fact, there is an interface war among different look-and-feel specifications. This paper considers the possibility to support multiple toolkits and multiple graphical user interfaces from user interface builders (UIBs). To accomplish that it is proposed to elaborate a user interface description language and to use it as an intermediate level between the internal structures of a UIB and the C source code generated for the application.

The User Interface War

Standardisation in the graphical user interface components started on the level of window systems. The specification and development of the X Window System was started early enough and had such significant sponsors as IBM and Digital that nothing could really stop X becoming the de facto standard for window systems.

This occured despite Sun's announcement of NeWS (Network extensible Window System) which was a success in the technological viewpoint but had no real chance to overcome X. At most Sun could have its hybrid X11/NeWS server accepted as a part of UNIX SVR4. See [She90a] for a historical overview and a comparision of X and NeWS.

Having chosen X as the basic window system several toolkits have been developed. The early products had their proprietary graphical user interfaces (GUIs) but later on different GUI (or "look-and-feel") specifications emerged; OPEN LOOK from AT&T and Sun (prominent members of UNIX International) and OSF's Motif are the major competitors. Currently the best known toolkits and desktop managers built around them have chosen one of these GUIs, see [Sou89a] and [Sou89b] for comprehensive reviews.

Among them OPEN LOOK and Motif are eagerly intended to be (de facto) industry standards, so both hardware vendors marketing their toolkits and independent software vendors are facing the risky decision to choose one or the other. The current situation seems to be a deadlock because X/Open, the potential judge, having both UNIX International and OSF (Open Software Foundation) as its members, delays the decision to include one of the specifications into its guidelines saying that "the market hasn't spoken yet" [Oul89a].

In any case, it can not really be expected in the near future that either of the standard candidates can win over the other. In fact, even if you have chosen a certain GUI you can still be confused when choosing a proper application programming interface (API) provided by a toolkit compliant to the GUI chosen, e.g. Xt+ and XView compliant to OPEN LOOK or the toolkit components of X.desktop and Looking Glass compliant to Motif [Sou89a].

Therefore an applications programmer either has to write different versions of an application for different APIs of different GUIs, or needs a tool supporting the migration among them.

User Interface Builders

Some years ago developers of graphics-intensive applications were quite happy to be able to program a (de facto) standard window system instead of writing tens of thousands of lines of source code to build a nice user interface and repeating it for every new workstation. However, it was obvious that standardisation had to go on to provide standard user interface (UI) elements and methods to manipulate them. To meet this requirement several toolkits have been developed offering different sets of UI elements ("widgets") with different appearance ("look") and behaviour ("feel"). See above and also [Sou89b].

On the other hand window systems and toolkits do not merely offer convenient APIs for applications programmers but also make it possible to create comfortable and efficient environments for program development (and for many other activities as well). Sophisticated user interface management systems (UIMSs) have been developed to support each phase of user interface development and to provide a complex control on resources and devices in use, see [Pri90a] for a comprehensive review on currents UIMSs.

When working in such an attractive environment it turns out that programming a toolkit quickly gets boring. Even to create a simple user interface with some buttons and scrollbars can require hundreds of lines of source code with very funny and very long function and variable names and with very difficult syntax in function calls. It is particularly annoying if you only want to create a rough UI sketch just to run and test a program and you do not (yet) want to make a complete user interface for an application.

That is why most UIMSs include tools for fast prototyping and code stub generation [Pri90a], and dedicated tools for interactive user interface building (user interface builders – UIBs) have been developed for this purpose, as well. For the latter ones the most recent examples are AutoCODE, ExoCODE, NeXTStep UIM [She90b], and GUIDE (Graphical User Interface Design Editor), [Sim89a].

Currently existing UIBs provide facilities to design user interface for an application in the most convenient way: dragging and placing the user interface elements with the mouse and adding some more pieces of textual information e.g. names for labels and call-back functions. When you are satisfied with the UI designed the UIB can generate the corresponding C code for the UI part of the application (or at least some useful code stubs) and probably after some manual modifications you can have your application prototype compiled and run.

Although the features above are very useful for rapid prototype creation and for preparing attractive demos, existing UIBs have quite strict limitations. The two most significant are as follows:

If you have modified the generated C code manually you can hardly retrieve the corresponding appearance in the UIB for further interactive modifications.

A certain UIB is stuck to one API (one toolkit) of a GUI, i.e. the source code can only be generated according to that API.

The key consideration of this paper is that if UIBs generated an intermediate user interface description language (UIDL) code first and had different translators for different APIs then the above deficiencies of current UIBs could be eliminated and we could also have the chance to support different GUIs from UIBs and therefore to provide a compromise platform in the UI debate.

Design Principles of a UIDL

The above problems imply that the intermediate UIDL should meet the following requirements:

1. It should comprise all the widgets accessed via the UIB in use. The nature of widgets also implies that it should be an object oriented language.
2. It should have a standard extension mechanism to append new objects.
3. It should have translators for different APIs of a certain GUI.
4. The translation rules should be separated from the UIB code, preferably placed in different files for different APIs, in order to add code generation for newly coming toolkits easily.
5. The UIDL should be able to allow conversions among UIDL files of different GUIs. (Yes, this is the hardest one, some more discussions are coming below.)
6. Each UIB should include a UIDL code generator, generating UIDL files from the internal structures of the UIB. (These UIDL descriptions could be translated into any API later, or could even be used in different applications.)
7. Each UIB should include a UIDL interpreter to retrieve previously generated and stored UI descriptions.

In this paper it is not intended to propose any syntactic or semantic aspects of a would-be UIDL, neither to deal with the ways to meet the above requirements. Nevertheless, I think all but the fifth one can be met within a reasonably short time and it would need reasonable efforts.

As for the fifth requirement, theoretically there are two ways to accomplish a useful conversion mechanism e.g. between OPEN LOOK and Motif descriptions.

> A very rigorous comparision could tell whether a UIDL as a superset of both OPEN LOOK and Motif (probably with some slight modifications in both) could be defined. It would mean that e.g. for an OPEN LOOK widget, it can either have a one-to-one mapping into a Motif widget or can be built up with several Motif widgets through the UIDL and vice versa.
>
> If the above comparision has a negative result some guidelines could be worked out so that a rough functional conversion still could be performed automatically, using the UIDL, and some refinement within the target GUI environment could complete the migration work between the different GUIs.

Note, that even the first conversion way implies weaker constraints than a complete mapping from one GUI to another, because it still allows some freedom in the "feel" part of the GUI specifications.

Further Advantages of a UIDL

So far UIBs have been considered as tools for fast prototyping. Now if we take a UIB equipped with a UIDL can also be very useful when you have a ready-made application and you want to develop a new UI or different UIs for that. The key for this usage is that a UIDL makes it possible that the UI part of the application is separated from the actual application not only in the design phase but even in the test phase.[†]

In the usual way of using a UIB you can immediately test the UI part, just designed, visually i.e. you can play with the created UI objects inside the UIB and you have dummy functions instead of the real application call-back procedures. The next step is to generate the corresponding source code for the application, have it compiled and linked and run the application. Although it is much more efficient than the traditional way of writing every piece of code manually, it still can be boring in more complex user interfaces or if you try different UIs, or just one with many slight modifications.

If you have a UIB capable for interpreting UIDL descriptions of UIs why not to inherit this capability to a temporary UI adaptor of the application? The main difference in the interpretation would be that a UIB will create its internal structures with dummy functions for the application call-backs and the UI adaptor of the application should generate the actual function addresses. It can be solved if we use string representation of the call-back functions provided by the application in a header file (Figure 1).

Then if a string representation of a particular function is encountered by the interpreter the corresponding function can be invoked. A bit more tricky way should be provided to pass the arguments for the application functions but it is not considered here.

```
/* declarations of call-backs */

void funct1(),
     ...

     functn();
```

And here is how it is used in the UI adaptor:

```
        ...

struct {
        char *funct_name;
        void (*funct_ptr)();
        } UIad_callbacks [] =

        {
         {"funct1",funct1},

            ...

         {"functn",functn}
        };
```

Figure 1: *String representation of call-back functions*

† Naturally, these considerations can not apply to all kinds of applications. It is assumed implicitly throughout this paper that applications are ideal in the sense that only a separated UI adaptor part communicates directly with toolkits or window systems.

The application compiled and linked with the UI adaptor now could interpret different UIDL descriptions generated previously by a UIB and also could dynamically choose between the APIs of the available toolkits providing that it is linked with all of them, or the dynamic linking facility can be used.

Conclusion

It would be better to put effort on the elaboration of a UIDL and the technology how to use it from UIBs, rather than to be waiting for the victory of one of the GUI fighters or for the conciliation between them.

References

[Oul89a] A. Ould, "X/Open faces tougher going," *UNIX World* **6**(10), pp. 51-59 (October 1989).

[Pri90a] M. Prime, "User Interface Management Systems – A Current Product Review," *Computer Graphics Forum* **9**(1), pp. 53-76 (March 1990).

[She90a] B. Shein, "Primal screens," *SunExpert Magazine* **1**(3), pp. 56-69 (January 1990).

[She90b] B. Shein, "Expert Object," *SunExpert Magazine* **1**(4), pp. 54-61 (February 1990).

[Sim89a] N. Simpson, "Porting Applications to the XVIEW Toolkit and the OPEN LOOK Graphical User Interface," *EUUG Conference Proceedings*, pp. 219-228 (September 1989).

[Sou89a] A. Southerton, "Friendly desktops," *UNIX World* **6**(11), pp. 62-67 (November 1989).

[Sou89b] A. Southerton, "Making UNIX easier to use ," *UNIX World* **6**(7), pp. 68-73 (July 1989).

Putting UNIX on Very Fast Computers

or

What the Speed of Light Means to Your Favorite System Call

Michael D. O'Dell

Consultant

ABSTRACT

A computer with a 250 MHz clock and built from leading-edge technology works in fundamentally different ways compared with a one-chip CMOS VLSI processor clocking at less than 50 MHz. The interactions between a UNIX implementation and its supporting hardware have always been quite subtle and remain a considerable headache for those charged with porting the system. But in addition to the imprecisions of fuzzy functional definitions for some key system facilities, the laws of physics conspire to make the marriage of very fast computers and modern UNIX systems an even more interesting challenge than it would normally be. This paper discusses some of the matchmaking necessary to achieve a matrimonious accommodation.

1. Introduction

Computers which operate on the hairy edge of physics are radically different beasts than the bus-based minicomputers where UNIX was born. Many machines of this latter class are now called "workstations" or "servers". Traditional supercomputers are not ideal platforms for UNIX, given their penchant for word addressing and limited memory management hardware. For many years, though, they were the fastest game in town by a large margin, so the architectural inconveniences in the name of raw speed were simply endured. Recently, though, this has started to change. High-performance implementations of what one might call the new "commodity instruction sets" hold the promise of providing supercomputer-class scalar performance with all the amenities that modern UNIX systems want to see in the underlying hardware (support for demand-paged virtual memory, for example).

The modern reality is that, rightly or wrongly, the software base available for a machine goes a long way toward determining its success in the marketplace, so building a new computer with Yet Another Instruction Set, and therefore no existing software base, is an undertaking fraught with enormous risks. To the reasonably risk-averse person (e.g., the people who fund such projects), it is clearly advantageous to *leverage*[†] the large installed software base associated with an established instruction set and operating system implementation. (The IBM PC clone-builders are some of the most successful proponents of this approach.) Going even further, if the new machine is sufficiently like its archetype, one can even leverage a very large part of the base operating system software implementation. This approach has the potential of maximizing return on investment while minimizing risks. The overall implication is a very high degree of compatibility even at the level seen by the kernel, with perfect compatibility when seen by user-mode programs. As was discovered, however, this "perfect user mode compatibility" can be a rather elusive goal when the differences between the underlying implementations are dramatic enough.

There are two ways to pursue the goal of building a mainframe-class, very fast instruction-set clone. The first is to use the highest performance off-the-shelf processor chipset available as the core of the machine and then concentrate on the rest of the computer to add value. This is a potentially workable approach because the instruction pipeline is only a small part of a large computer – the memory and I/O systems of a mainframe-class machine can easily consume the lion's share of the logic packages and the engineering effort needed to build the whole machine, particularly if the processor is but a few VLSI chips. The big advantage here is that many of the subtle instruction-set compatibility issues are resolved by simply buying a working processor chip. One problem with this approach, however, is that such a machine probably can't

† Boardroom talk for "use", "run", or "execute unmodified".

go dramatically faster than any other machine built from the same, commonly available chips. If one intends for performance to be the fundamental, market-distinguishing characteristic of a new machine, this may not be the best approach.

The other alternative is to license a machine architecture from one of the several purveyors, and then set out to build a new implementation which squeezes every iota of performance from whatever base technology one feels comfortable choosing. In this approach, the down side is that you must now build *all* of the computer; the up side which may make the approach worth the risk is that one now gets a *chance* to achieve a real performance margin over machines built with commodity processor parts. At Prisma Computers, Inc., we embarked on this second path – construction of a supercomputer which would be binary compatible with a popular workstation family, and built from "whole cloth". For the purpose of this effort, the definition of *binary compatible* was beguilingly simple: if a binary program runs on one of the to-be-cloned workstations, it must run identically on the new machine, only much, much faster.

In the course of designing this computer and preparing an operating system for it, we discovered many strange and often confounding things. Most of these issues relate directly to what promises a machine and its operating system must make to programs. Particularly troubling are the dark corners or boundary conditions in an architecture which ultimately are revealed to a user program in great detail. Programs often have deeply-ingrained notions about what they expect to see in these revelations, and what they have come to expect of certain hitherto inadequately-specified operating system features. These features can be viewed as an example of "inadequate law", so as with traditional jurisprudence, one must fall back on the available historical precedent which is seldom more compelling than the statement, "This feature has always worked on other machines, so my program is free to use it!" Programs which take this view of the world make "absolute binary compatibility" an interesting concept and a tricky goal to achieve.

At this point, it is useful to examine a new player in the compatibility game: the Application Binary Interface, or ABI, which was developed for UNIX System V Release 4. To understand the importance of the ABI, we need to look at history for a moment. One of the great success factors in the IBM PC world has been the large software base available for the machines. Historically, this has been achieved by everyone simply building the same computer (as seen by programs). In the PC world, the definition of "compatible" has been: if a program or operating system works on a genuine IBM Personal Computer of some flavor, it should work on a Brand-Y PC clone; if not, the Brand-Y machine is broken. The problem is that in the UNIX world, the design space is much, much larger than the PC world, so expecting everyone to build similar machines isn't reasonable. However, if UNIX is to take off, it is vitally important that all UNIX systems using the same instruction set be able to execute the same binaries so vendors can market "shrink-wrapped" user application software. The ABI, then, is an attempt to provide the needed compatibility at the user level while not overly constraining the operating system implementation.

An ABI is created for a specific architecture and defines *all* system interfaces available to user-level programs on any implementation of System V Release 4 running on that architecture. The operating system's internal structure, on the other hand, is allowed to be changed as desired (e.g., for efficiency or performance), as long the changes don't violate the ABI specification seen by user-mode ABI-compliant programs. The goal, then, is that ABI-compliant binaries for a given architecture should run on all implementations of that architecture, with which provide the constant, basic functionality specified by the ABI. In general, the ABI specifications try to be as high-level as possible, striving to avoid documenting details which should ideally be left as implementation choices. However, a great many things must be specified if programs are to achieve their portability goals so some of the problems addressed later in this paper are still real issues. The reason that Prisma didn't simply provide an ABI-compliant interface, instead of the more rigorous PC-style absolute user compatibility, was simply one of timing. The operating system for the Prisma P1 was not going to be System V Release 4 until well after the first shipments; and a very critical business issue was leveraging the existing large third-party software catalog without any changes. Some day, ABI compliance will be sufficient for a new machine, but it wasn't in Prisma's development time frame.

2. The Rules of the Game

The Prisma P1 was originally designed to a target clock cycle of 4 nanoseconds, and since the P1 was a RISC machine, the fervent hope was that it execute very nearly one instruction per clock. To give you a feel for this target clock rate, a 4 nanosecond clock period translates to a 250 MHz clock frequency (well into the UHF television broadcasting band). At these frequencies, signals are *not* digital. It is quite true that the intent is that the signals represent ones and zeroes, but at these frequencies, the signals themselves are profoundly analog.

If the machine is to run with a 4 nanosecond clock, the logic used in the machine must have loaded gate delays of about a hundred picoseconds, with the rise times of a 250 MHz "square wave" logic signal measured in a few tens of picoseconds. The harmonics of such signals extend well above 2 GHz into the serious microwave and radar frequencies. This means that "wires" or "traces" on a circuit board are all transmission lines, and keeping the signals contained in the desired paths becomes an RF engineering task. Because the signal traces are transmission lines, the trace routing is further complicated by the restriction there can be no "stub" paths branching off a main run to a non-collinear connection on the way. Instead, the signals must be routed in a purely linearizable "connect-dots-without-lifting-your-pencil" path. Complicating matters even further, the transmission line traces were "differential pairs," which means that instead of running one wire from place to place, Prisma ran two, with one wire going from one to zero while the other goes from zero to one at the same time. The logic signal is taken as the difference between the two lines (hence the name "differential") instead of between a single line and ground as in the case of standard TTL logic. This technique has important electrical propagation characteristics as well as some handy logical advantages – for instance, inverters are almost unheard of – just reverse the connections! These traces must be routed on the circuit board maintaining a specific distance between them in order to preserve the characteristic impedance, while being routed as a pair introducing only an *even* number of half-twists en route to maintain signal polarity. PC board routing software which can adequately do this complex job proved quite hard to come by.

On real circuit board material manufacturable at finite cost, electronic signals propagate noticeably slower than the speed of light in a vacuum. The rule of thumb for the board materials used in the P1 was that 1 inch of circuit board trace eats 180 picoseconds of flight time, so 10 inches of trace will eat about half the available clock cycle. Given the differential-pair stub-free routing requirements, it was frighteningly easy to get routes on a 4 inch by 4 inch board which came dangerously close to 10 inches, thereby leaving the logic in such a path about 2 nanoseconds in which to do all its work.

All of these signalling and routing requirements also apply to connectors and cables. The most vexing issues are that signals generally propagate even slower through cables, making impedance control for minimize reflections a constant concern. These issues particularly complicate the physical construction of tiny, fragile connectors which require many, many pins to interconnect modules with ever-growing complexity.

One of the most serious problems Prisma faced was the level of integration available in the Gallium Arsenide (GaAs) technology being used for the machine. The GaAs logic chips were medium- to large-scale integration, meaning that a chip contained between about 200 and 500 gates, as opposed to the tens-of-thousands of gates available on a CMOS VLSI chip. If a function needed more gates than would fit on a chip, the signal path had to go off-chip, paying dearly for driving the signals across the circuit board instead of across the chip surface. This made the partitioning and physical floor plan of the logic and the resulting chips quite critical. There were never enough gates, so only functions which were absolutely essential for correct behavior could be done in the GaAs logic. The processor had no choice but to be a RISC – there wasn't enough real estate to allow alternate ideas.

One other characteristic of the interconnection constraints had a profound architectural impact. At the speeds Prisma was targeting, there is no such thing as an arbitrated "bus." The data interconnects in the P1 were nominally 64-bit wide, 4ns clock, simplex point-to-point data paths. An "address path" was just a data path which carried address information as the data and usually required fewer bits than the 64-bit data paths. A full-duplex data path (two simplex paths which connect the same two parts of the machine but in opposite directions) required about 300 wires total (the signals are all differential, so 64 bits of data require 128 wires each direction; differential parity and control signals consumed the rest). These paths were hideously expensive in both space and time, so adding even one path to the implementation required an overwhelming argument in its favor – on the order of "The machine cannot possibly get the right answer if it's not there." Because there were no busses carrying a combined data and address path, there was no convenient place to put useful trinkets like EPROMs or UARTs which have come to be expected for mundane tasks like booting and printing last-gasp death messages when something goes horribly wrong. This constraint, more than any other, forced the machine to have a service processor. (The kernel development group successfully avoided having anything to do with the service processor beyond defining some requirements and a few necessary interfaces needed between the ersatz "ROM" code placed into P1 memory by the service processor and the kernel.)

The final, most important rule of the game was that the machine had to be realizable using parts which are actually available – not vaporspecs for non-existent chips, but **real** DRAMs and **real** SRAMs and **real** logic chips (or at least a real fabrication process), running at speed, which one could buy (or make) in production quantities, at rational prices. This constraint is a genuine nuisance.

All the rules from physics and engineering discussed above induced two fundamental constraints which profoundly affected everything in the design of the Prisma P1:

1. Doing anything takes at least a clock, and
2. Getting there to do it often takes at least a clock as well.

This means that the back end of the instruction pipe *cannot* communicate with the front of the pipe in one clock cycle, and therefore, there can be no "atomic" decisions whereby the instruction fetch unit decides what to do next based on the entire state of the CPU (e.g., take a pagefault). This presents a non-trivial problem.

3. Suggestions on How to Play the Game

The general problem to be addressed is that processing an instruction always takes too long, either because the logic takes too long to get the answer, or that the logic to be consulted is too far away as well as being too slow. One of the most productive ways to hide latency is to increase parallelism, thereby letting more operations progress at once. This means that functions which are one clock on other machines became pipelines on the P1 and therefore had to cope with multiple outstanding operations.

The biggest latency issue was the dramatic disparity between the speeds of the memory system and the processor. Since the machine was to be relatively low cost and support large memories, hidden somewhere behind the curtain had to be commodity 80 nanosecond DRAMS. Note well that the 80 nanosecond number is *access* time, not the transaction time, which is on the order of 180-200 ns. This limit drove the decision to make the memory system a 3-level hierarchy. The first level instruction and data caches were integral to the GaAs portion of CPU. It was indeed a dark day when it became apparent that the GaAs RAMs we had planned to use in the primary caches were not real, and that high-performance ECL "474" RAM parts would have to be used instead. This meant that the primary caches would have to become pipelined: the first 4 nanosecond cycle did the cache tag access, and the second cycle accessed the cache line and returned data. The primary caches could start an operation every clock and the data cache would have to support multiple outstanding cache misses. To do this for both loads and stores, the primary data cache implemented the equivalent of load-store locks for all of main memory. The second level cache was a very large, interleaved, integrated physical cache front-ending the third-level DRAM arrays. I/O traffic went through the secondary cache with some clever tricks to prevent cache-wiping during I/O bursts. For a load, a hit in the secondary cache returned data to the CPU register in about 15 cycles round-trip (updating the primary data cache en route), while awakening the DRAMs from their deep slumber required 60+ cycles round-trip.

As was intimated in the description of the primary data cache, load and store instructions were implemented as non-blocking operations. For example, a primary data cache miss caused by a load did not stall the pipe until a later instruction attempted to use the destination register of the load as a source operand. Because the cache miss could be answered by either the secondary cache or the DRAM array, load instructions could complete out of order. When coupled with the non-atomic behavior of the pipeline, this could cause multiple outstanding pagefaults to occur, and resulted in considerable grief for the operating system. While the non-blocking loads and stores added significantly to the complexity of both the hardware and software, the system simulator clearly showed that this feature had a dramatic impact on machine performance and that the work in the kernel was well worth the effort.

Another area impacted by the speed disparity between the processor and memory system was the architecture of the memory management unit (MMU) – both in the page table structure and the translation look-aside-buffer (TLB). In some circles, with processors operating at more modest performance levels, it is considered reasonable to implement an MMU with only a software-reloaded TLB; but for a machine like the P1, such a design is untenable for performance reasons. The P1 was designed to be a supercomputer and to run processes requiring large address spaces, so the TLB-miss processing speed, directly related to the average page-table walk length, was an important issue. The "reference MMU" for the P1's underlying RISC processor uses a short-circuit 3-level page table with 4Kbyte leaf pages. The problem with this design (from the P1 perspective) is that the deep page table tree required a large and rather clever TLB design to prevent multiple memory references when walking the page tables for a TLB miss. While CMOS implementations of the reference MMU easily have enough transistors to support the architecture, the required cleverness was beyond the means of the GaAs implementation technology used in the P1. Simulation studies of alternative designs led back to a two-level version of the reference MMU using 8Kbyte pages. This approach allowed the TLB to be divided in half, dedicating half to level-one entries and half to level-two entries. The resulting design allowed processes with very large address spaces to exhibit very low TLB miss rates, and when a miss did occur, the mean page-table walk length was well under one entry. The entire MMU design effort was a case where the synergy of developing the kernel and the hardware in parallel proved extremely valuable. Several "strong candidate" designs were actually coded into the architectural simulator being used for kernel development and were evaluated by running programs (including the operating system) before the final design was frozen.

In summary, building a machine as fast as we intended the P1 to be requires all the tricks you can think of, and probably more. It also requires a great deal of attention to matters which are second-order effects (if not third!) on slower machines. In fact, you almost end up standing on your head; the first-order and other-order issues almost seem to be reversed in this world – things which matter a lot in slower or more cost-sensitive designs are often down in the noise, and things which most designs don't attempt to do anything about matter a great deal.

4. Great Expectations – What UNIX Wants from Hardware

UNIX systems at least as early as the Sixth Edition used their hardware in interesting and aggressive ways. The technique used by the V6 system to grow the stack segment on demand assumed instruction restart capability which was easy to do on the DEC PDP-11/45 and PDP-11/70 because of the splendid support in the MMU. Providing the equivalent capability on other members of the PDP-11 product line was considerably less trivial. The early microprocessor-based UNIX implementations (V7 from UNISOFT on the Codata 3200, as an example) had to deal with the unavailability of instruction restart on Motorola 68000 processors. Various tricks were devised, but the general approach was to add a "sacrificial instruction" to the procedure prolog code which did a probe of the address space designed to provoke a stack-growing protection trap if required. The instruction could not be salvaged, so the MMU trap recovery code looked at the offending instruction to see if it had the designated sacrificial form and seemed to be just probing the stack. If so, then the instruction was just ignored. This is the first example known to the author where a restricted but unprivileged instruction sequence was used to paper over an inadequacy of the underlying hardware behavior. Other UNIX implementations have used similar work-arounds to deal with various problems, often relating to MMU and trap recovery behavior, but the days of those tricks covering for inadequate hardware support are over.

Modern UNIX systems such as System V Release 4 demand much more from the underlying hardware. While one might argue academically that copy-on-write is not strictly required for a System V Release 4 implementation, it is most certainly required if the implementation is not to be at a serious disadvantage in the real marketplace. Furthermore, support for facilities such as copy-on-write must not exact excessive performance penalties.

Adding to the complexity of providing these facilities at high speed are new processor architectures which specify what we call "atomic trap observance." Atomic traps appear to happen as part of the instruction execution. For example, a load instruction which pagefaults is specified as causing the fault, rather than completing the load, as part of the instruction execution. The machine must begin the trap processing instead of executing the next instruction after a pagefault.

The inclusion of this atomic behavior in the architecture is understandable, particularly if you have ever worked on fault recovery code on a complex machine. The beguilingly simple specification is a blessing for the low-level kernel programmer because it makes fault recovery quite straightforward and generally makes instruction restart quite easy to achieve. However, if the MMU which detects the pagefault is 3 or 4 pipeline stages removed from the instruction fetch unit, and when, because of out-of-order completion, at least two of those pipeline stages can irrevocably alter registers which were used to form the effective address of the load or store, the operating system is faced with several serious problems.

The first problem is simply assuring the fundamental semantic correctness of operations such as copy-on-write by "re-effecting" out-of-order instructions (since one cannot just backup the program counter!). The approach taken in the P1 was to replace the usual "MMU Registers" which record the necessary trap recovery information with a pagefault queue whose entries provide enough information to "re-effect" the faulting instruction in spite of its out-of-order completion.

For the private purposes of the operating system kernel, the MMU Fault Queue provided all the support needed to recover from pagefaults and continue, and it worked quite well on the simulator. The simulator used by the kernel group actually allowed more asynchrony than the real P1, thereby catching a few subtle bugs which would have been much harder to find running "at speed" on a real P1. The real nightmare, however, related to facilities afforded to User Programs.

Modern UNIX systems (or possibly more aptly, systems gene-spliced with TENEX) now have virtual memory systems which let user programs actively participate in memory management functions by allowing them to explicitly manipulate their memory mappings. This capability is not a problem in and of itself, but rather, serves as the courier of an engraved invitation to Hell. In this Brave New World, programs may register signal handlers for pagefaults and in turn, they expect to take the fault, diddle some mapping information, and return from the fault to carry on as if nothing happened. All the clever design work to make the P1 go fast and still let the operating system transparently make things come out right seemed to be torpedoed by this one requirement. Several loud voices suggested that such programs should just get what they so richly deserved, but when it was pointed out that the Bourne shell was one of the most commonly-executed offenders,† that alternative was clearly unacceptable. The stated business requirements for absolute binary compatibility were also persuasive. Even more unacceptable, however, was slowing down the entire machine by a large factor just to support infrequent exception handling.

Interestingly enough, most architectures already have "non-atomic" or deferred traps generated by long-running coprocessor instructions (usually floating-point), and programs seem to have resigned themselves to dealing with them as a matter of course. An interesting case to consider is what must be done in a language with frame-based exception handling (Ada or Modula-3) to deal with returning from a subroutine before all the floating-point instructions issued in the subroutine have finished. The unfinished instructions are still subject to the exception-handling wrapper around the invoking subroutine call. Therefore, an explicit instruction to synchronize the state of the floating-point coprocessor must be used in the procedure return sequence to prevent the integer pipeline sequence from completing too soon (thereby abolishing the exception-frame environment while incomplete coprocessor instructions could still cause exceptions). This need to be explicitly aware of the potential asynchrony has resulted in lowered expectations as to exactly what information can be revealed about the exact state of the machine when the trap is induced, and lowered expectations about what a program can do as a result of a trap (delivered to the user program as a UNIX signal), at least for coprocessor traps. The problem with atomic trap behavior for loads and stores is that it essentially requires synchronizing the entire memory system pipe on *each* load or store instruction, and this completely defeats all the effort to pipeline those functions in order minimize latency by maximizing overlap. It is unfortunate that the direction processor architectures in general seem to be headed is toward more synchrony and lock-step determinism in such behaviors when it can be readily shown that reducing synchrony and determinism is a powerful way to improve performance.

Another expectation that UNIX systems have about machines has to do with the way kernel and user address spaces co-exist. The UNIX system at issue clearly wanted the kernel to be mapped into the upper one-half gigabyte of the user's 4 gigabyte virtual address space, leaving 3.5 gigabytes for user programs. Indeed, the System V Release 4 ABI for the P1's underlying RISC architecture requires that the valid user virtual address space be 3.5 gigabytes starting at location zero, so changing the virtual memory layout in a substantial way was never a realistic option. Since the primary caches use virtual address tags for reasons of speed, redundantly mapping the kernel into each user address space caused the *entire* kernel to alias in every address space. Since the primary cache also had process-id context bits in the tags to prevent unnecessary flushing and cache-wiping, a naive implementation would require flushes on every context switch to insure kernel coherency. The first alternative (discarded rather quickly) was simply marking the entire kernel "no-cache" in the page tables, but the performance impact was too severe. The alternative chosen was to modify the MMU behavior equations so that the uppermost 0.5 gigabytes of virtual space always got processed as if in context zero, with the proper considerations given to the supervisor or user mode of the processor. This prevented aliasing by forcing the kernel to only appear in one context.

The advantage of this design is that it allowed aggressive cache-flush prevention algorithms which were added to the virtual memory system to perform quite effectively. The disadvantage is that to some degree, the decision permanently forces a considerable chunk of user virtual space to be dedicated to kernel use, whether or not it is needed down the road. The implications of not making these decisions, however, were considered to be much worse, based both on calculations and tests validating the effectiveness of the cache-flush prevention algorithms.

† The Bourne shell uses *sbrk()* to manipulate the mappings instead of *mmap()*, but it does catch the appropriate signal and expects to return and restart the failing instruction nonetheless.

5. Great Assumptions – What Programs Think UNIX Provides

The complexity of the UNIX virtual machine assumed by programs (and only partially documented by the system call interface manual pages) only seems to be increasing. In the halcyon days of the Sixth Edition, the manual page for the *signal()* system call warns that signals were primarily a way for a suddenly-dying program to discover its assailant, and not the event-based interprocess communications mechanism they have become of late. Admittedly this author would be the last person to complain about signals becoming reliable, but rather, the complaint is that the flexibility of the interface has not tracked the richness of use. Again, the issue of user programs intercepting pagefault traps with a signal handler was at the center of the problem.

In order to resolve the problem of the significant performance impact caused by the architecture specifying atomic trap behavior for pagefaults, a cooperative effort between Prisma and the purveyor of the architecture produced a revision to the architectural specification which provided that *deferred* traps, already legal for floating point operations, could also be legally generated by certain additional kinds of exceptions (e.g., pagefaults), but *only* when a User Program has not requested visibility of the trap. On the face of it, this sounds like a reasonable compromise. The system is allowed to do whatever it wants, ripping along as fast as possible, as long as no one asks to see the details. On the other hand, if a process asks to watch the entrails in action, the machine must behave in such a way as to allow the operating system to reveal to the interested User Program all the relevant gore about what was happening, and in such a way that the user program can diddle with the state and have the diddling take immediate effect. And how does a program request to view the entrails? By registering a signal handler, of course.

This still sounds just fine – programs that ask get hurt; those that don't, don't. There is still a serious flaw with in model, however. A great many programs (all programs linked with the FORTRAN run-time system, to pick just one group at random) routinely catch *all* signals in an attempt to print the moral equivalent of:

```
yourprogram bailing out near line 1 with Segmentation Violation
```

all in the name of "user-friendliness." The program isn't going to try and recover from the disaster; it only wants to report the fatal mistake and then die peaceably. By registering the signal handler for the appropriate signal, however, it incurs all the penalties of a program expecting to examine every trap while trying to do its own demand paging. As was said above, in the case of the Prisma P1, these penalties have a quite serious performance impact. The crux of the problem is that a user of the *signal()* system call doesn't have any way to request a specific "quality of service" – so the system must provide the most expensive, guaranteed-to-be-sufficient service, when the barest minimum would be perfectly adequate in many cases.

6. Great Consternations – Life at the Boundary Conditions

No UNIX Programmer's Manual known to the author contains detailed information about what questions a signal handler can expect to ask of the machine; about what state information must be revealed in response to those questions; nor about how a signal handler can expect to effect the state of the running program, particularly by modifying the machine state revealed to the signal handler and then "returning" from the signal handler. Indeed, in some programs, the signal handler doesn't ever return – it does a *longjmp()* to forcibly redirect the control flow of the program! The ANSI C standard attempts some limited statement of allowable behavior applicable in the broadest, most portable cases which must work under essentially any operating system, but it is insufficient. The current state of confusion exists because under specific operating systems, certain behaviors have been discovered to work and they have largely been perpetuated, even if undocumented. One solution is to categorize "quality of service" for signal handlers, and thereby allowing a handler to register its intent and its requirements as to the completeness of information needed by the handler. In this way, a program need not pay in reduced performance for more information than it needs.

Prisma implemented a very limited (lame?) version of this quality-of-service notion by providing a new system call named *sigdefer()* for manipulating a mask which indicates that certain traps should remain deferred even if the program registers a signal handler for them. This mask is potentially "sticky" across *exec()* system calls, thereby providing support for a **sigdefer** command which takes another command as its argument. The **sigdefer** command sets the "force deferred signal" mask for the indicated signals and then runs the command, thereby allowing it to run unimpeded by *signal()* calls intended to only print messages. While knowledgeable programs can use the new system call directly, the **sigdefer** command allows existing binaries to run at full speed when the user decides it is safe. Within the kernel group, the

sigdefer command was called, at various times, **gofast** and **benchmark**. This is approach is admittedly a bit disreputable, but it solved the problem until someone defines a quality of service interface for signal handlers.

In return for specifying some behavior more flexibly, this author would like to encourage the use of the word *undefined* more frequently in standards and architectural behavior specifications. While *defined* means that a certain behavior is guaranteed, the author intends *undefined* to mean:

> Even if you can somehow discover how an *undefined* facility behaves on an implementation, you are expressly prohibited from using that knowledge in any program because any implementation is completely free to give you a different answer EVERY time you use it!

Maybe the threat is enough to make people stick to the *defined* behavior, but it is delicious to contemplate making an *undefined* behavior actually work differently each time!

7. Comments and Conclusions

Building fast computers is an interesting and exciting challenge, particularly when you first discover that *Everything You Know Is Wrong!* In doing the kernel for the P1, the biggest single cause of grief was not the complexity arising from nominal behavior of the machine (even including multiple outstanding pagefaults), but rather it was because programs, especially in signal handlers, are allowed to ask arbitrary questions and request arbitrary behavior of the machine at very inopportune moments. Given the current operating system interfaces (ABI specifications), answering the questions and implementing the requested behavior can only be done at the expense of serious performance penalties. These penalties can unfortunately be inflicted upon unsuspecting programs just trying to be friendly with the user, rather than only on those programs truly interested in exactly what the machine is doing on a cycle-by-cycle basis.

Historically, UNIX programs have taken a quite naive view of the world: system calls always work (who checks error returns?), *malloc()* never fails, and a signal handler is free to crack stack-frames and diddle saved register values to its heart's content with perfectly predictable (or at least, experimentally discoverable on a specific machine) results. Well, maybe you should check the return value from *unlink()*; some machines rap your knuckles quite smartly for dereferencing through Location Zero; just maybe catching that signal is a lot more expensive than you thought; and most certainly, meddling with the saved state to achieve a particular effect is much harder than you thought. In the future, machines will continue to get faster and more internally concurrent, if not more parallel, and continuing this fantasy of the world being atomically lock-step-perfect simply flies in the face of the physical reality of fast machines. The sooner we can disabuse programs and ABI specifications from this fantasy, the sooner we can really exploit of the machines we could build.

The kernel for the Prisma P1 contained a lot of very creative work; this paper has described only a small part of the issues we faced in trying to make a workstation-tuned operating system safe for a serious supercomputer. The virtual memory system, the filesystem, the scheduler (see [Ess90a]), the basic I/O system and fundamental system resource management were all modified in various ways.

One final note: The kernel for the Prisma P1 was up and running in two forms. A complete P1 kernel except for the lowest-level machine-specific MMU hardware support and I/O system code ran on a production basis on all the large compute servers at Prisma, day-in and day-out. This system contained over 95% of the code in the ready-for-hardware P1 kernel. The entire P1 kernel, complete with P1-specific MMU and I/O system code, was running solidly on the P1 Architectural Simulator under extreme stress-test loads.

8. Acknowledgements

The author did not learn any of these things alone, rather they were learned in a very creative and rewarding group effort. Brian Berliner, Ray Essick, Bill Fuller, Tony Schene, and the author comprised the Prisma P1 Kernel Team. Bill Wallace made the Prisma software group a great place to work. Chris "Roz" Rozewski and Rob Ober created the MMU and the memory system for the Prisma P1, and Mike Baird's group designed the I/O system. The author is very grateful to all of them for their contributions and the chance to work with them.

References

[Ess90a] Raymond B. Essick, “An Event-based Fair Share Scheduler,” *Washington USENIX Proceedings* (January 1990).

tmpfs: A Virtual Memory File System

Peter Snyder

Sun Microsystems Inc.
2550 Garcia Avenue
Mountain View, CA 94043
peter@eng.sun.com

ABSTRACT

This paper describes *tmpfs,* a memory-based file system that uses resources and structures of the SunOS virtual memory subsystem. Rather than using dedicated physical memory such as a "RAM disk", tmpfs uses the operating system page cache for file data. It provides increased performance for file reads and writes, allows dynamic sizing of the file system while requiring no disk space, and has no adverse effects on overall system performance. The paper begins with a discussion of the motivations and goals behind the development of tmpfs, followed by a brief description of the virtual memory resources required in its implementation. It then discusses how some common file system operations are accomplished. Finally, system performance with and without tmpfs is compared and analyzed.

1. Introduction

This paper describes the design and implementation of *tmpfs,* a file system based on SunOS virtual memory resources. Tmpfs does not use traditional non-volatile media to store file data; instead, tmpfs files exist solely in virtual memory maintained by the UNIX kernel. Because tmpfs file systems do not use dedicated physical memory for file data but instead use VM system resources and facilities, they can take advantage of kernel resource management policies.

Tmpfs is designed primarily as a performance enhancement to allow short lived files to be written and accessed without generating disk or network I/O. Tmpfs maximises file manipulation speed while preserving UNIX file semantics. It does not require dedicated disk space for files and has no negative performance impact.

Tmpfs is intimately tied to many of the SunOS virtual memory system facilities. Tmpfs files are written and accessed directly from the memory maintained by the kernel; they are not differentiated from other uses of physical memory. This means tmpfs file data can be "swapped" or paged to disk, freeing VM resources for other needs. General VM system routines are used to perform many low level tmpfs file system operations. This reduces the amount of code needed to maintain the file system, and ensures that tmpfs resource demands may coexist with other VM resource users with no adverse effects.

This paper begins by describing why tmpfs was developed and compares its implementation against other projects with similar goals. Following that is a description of its use and its appearance outside the kernel. Section 4 briefly discusses some of the structures and interfaces used in the tmpfs design and implementation. Section 5 explains basic file system operations and how tmpfs performs them differently than other file system types. Section 6 discusses and analyzes performance measurements. The paper concludes with a summary of tmpfs goals and features.

2. Implementation Goals

Tmpfs was designed to provide the performance gains inherent to memory-based file systems, while making use of existing kernel interfaces and minimising impact on memory resources. Tmpfs should support UNIX file semantics while remaining fully compatible with other file system types. Tmpfs should also provide additional file system space without using additional disk space or affecting other VM resource users.

File systems are comprised of two types of information; the data for files residing on a file system, and control and attribute information used to maintain the state of a file system. Some file system operations require that control information be updated synchronously so that the integrity of the file system is preserved. This causes performance degradation because of delays in waiting for the update's I/O request to complete.

Memory-based file systems overcome this and provide greater performance because file access only causes a memory-to-memory copy of data, no I/O requests for file control updates are generated. Physical memory-based file systems, usually called RAM disks, have existed for some time. RAM disks reserve a fairly large chunk of physical memory for exclusive use as a file system. These file systems are maintained in various ways; for example, a kernel process may fulfill I/O requests from a driver-level strategy routine that reads or writes data from private memory [McK90a].

RAM disks use memory inefficiently; file data exists twice in both RAM disk memory and kernel memory, and RAM disk memory that is not being used by the file system is wasted. RAM disk memory is maintained separately from kernel memory, so that multiple memory-to-memory copies are needed to update file system data.

Tmpfs uses memory much more efficiently. It provides the speed of a RAM disk because file data is likely to be in main memory, causing a single memory-to-memory copy on access, and because all file system attributes are stored once in physical memory, no additional I/O requests are needed to maintain the file system. Instead of allocating a fixed amount of memory for exclusive use as a file system, tmpfs file system size is dynamic depending on use, allowing the system to decide the optimal use of memory.

3. tmpfs usage

Tmpfs file systems are created by invoking the mount command with "tmp" specified as the file system type. The resource argument to mount (e.g., raw device) is ignored because tmpfs always uses memory as the file system resource. There are currently no mount options for tmpfs. Most standard mount options are irrelevant to tmpfs; for example, a "read only" mount of tmpfs is useless because tmpfs file systems are always empty when first mounted. All file types are supported, including symbolic links and block and character special device files. UNIX file semantics are supported.† Multiple tmpfs file systems can be mounted on a single system, but they all share the same pool of resources.

Because the contents of a volatile memory-based file system are lost across a reboot or unmount, and because these files have relatively short lifetimes, they would be most appropriate under /tmp, (hence the name *tmpfs)*. This means /usr/tmp is an inappropriate directory in which to mount a tmpfs file system because by convention its contents persist across reboots.

The amount of free space available to tmpfs depends on the amount of unallocated swap space in the system. The size of a tmpfs file system grows to accommodate the files written to it, but there are some inherent tradeoffs for heavy users of tmpfs. Tmpfs shares resources with the data and stack segments of executing programs. The execution of very large programs can be affected if tmpfs file systems are close to their maximum allowable size. Tmpfs is free to allocate all but 4MB of the system's swap space. This is enough to ensure most programs can execute, but it is possible that some programs could be prevented from executing if tmpfs file systems are close to full. Users who expect to run large programs and make extensive use of tmpfs should consider enlarging the swap space for the system.

4. Design

SunOS virtual memory consists of all its available physical memory resources (e.g., RAM and file systems). Physical memory is treated as a cache of "pages" containing data accessed as memory "objects".

Named memory objects are referenced through UNIX file systems, the most common of which is a regular file. SunOS also maintains a pool of unnamed (or *anonymous,* defined in section 4.4) memory to facilitate cases where memory cannot be accessed through a file system. Anonymous memory is implemented using the processor's primary memory and *swap* space [Gin87a].

Tmpfs uses anonymous memory in the page cache to store and maintain file data, and competes for pages along with other memory users. Because the system does not differentiate tmpfs file data from other page cache uses, tmpfs files can be written to swap space. Control information is maintained in physical memory allocated from kernel heap. Tmpfs file data is accessed through a level of indirection provided by the VM system, whereby the data's tmpfs file and offset are transformed to an offset into a page of anonymous memory.

† Except file and record locking, which will be supported in a future SunOS release.

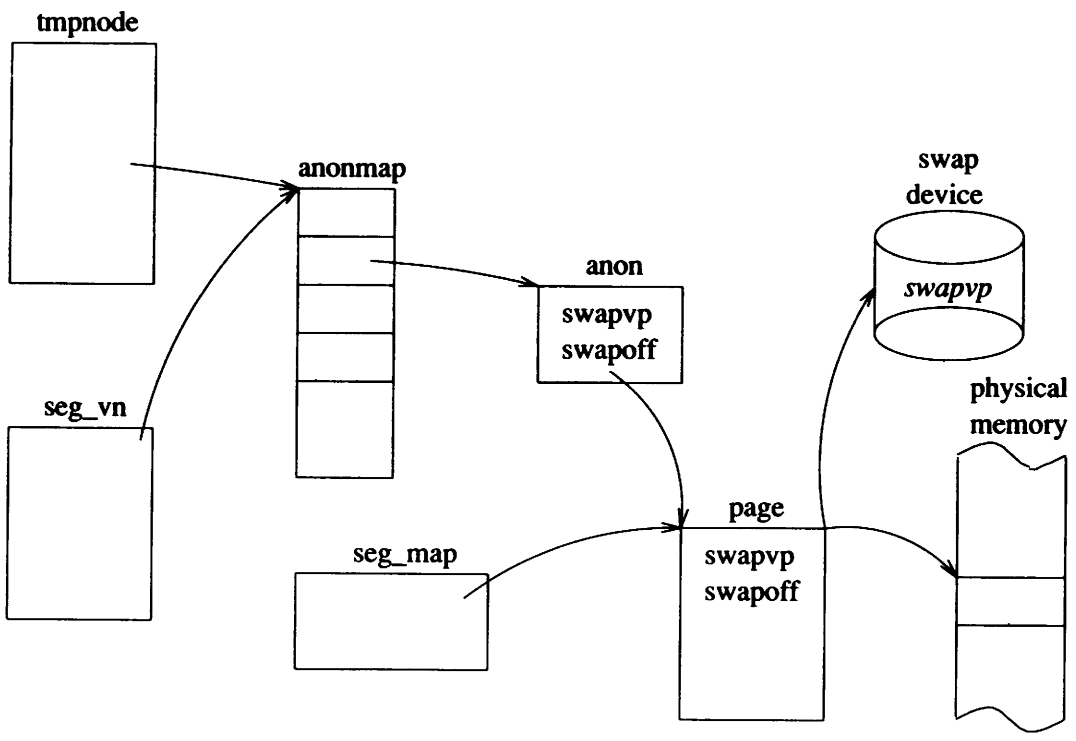


Figure 1: *tmpfs block diagram*

Tmpfs itself never causes file pages to be written to disk. If a system is low on memory, anonymous pages can be written to a swap device if they are selected by the pageout daemon or are mapped into a process that is being swapped out. It is this mechanism that allows pages of a tmpfs file to be written to disk.

The following sections provide an overview of the design and implementation of tmpfs. For detailed information about the SunOS virtual memory design and implementation, please consult the references [Gin87a] and [Mor88a] listed at the end of the paper.

To understand how tmpfs stores and retrieves file data, some of the structures used by the kernel to maintain virtual memory are described. The following is a brief description of some key VM and tmpfs structures and their use in this implementation.

4.1. tmpfs data structures

Figure 1 shows the basic structure and resources that tmpfs uses for files and their access. Their use in tmpfs is defined below.

4.2. vnodes

A *vnode* is the fundamental structure used within the kernel to name file system memory objects. Vnodes provide a file system independent structure that gives access to the data comprising a file object. Each open file has an associated vnode. This vnode contains a pointer to a file system dependent structure for the private use of the underlying file system object. It is by this mechanism that operations on a file pass through to the underlying file system object [Kle86a].

4.3. page cache

The VM system treats physical memory as a cache for file system objects. Physical memory is broken up into a number of *page frames,* each of which has a corresponding *page structure*. The kernel uses page structures to maintain the identity and status of each physical page frame in the system. Each page is identified by a vnode and offset pair that provides a handle for the physical page resident in memory and that also designates the part of the file system object the page frame caches.

4.4. anonymous memory

anonymous memory is term describing page structures whose name (i.e., vnode and offset pair) is not part of a file system object. Page structures associated with anonymous memory are identified by the vnode of a swap device and the offset into that device. It is not possible to allocate more anonymous memory than there is swap space in a system. The location of swap space for an anonymous page is determined when the anonymous memory is allocated. Anonymous memory is used for many purposes within the kernel, such as: the uninitialised data and stack segments of executing programs, System V shared memory, and pages created through copy on write faults [Mor88a].

An *anon structure* is used to name every anonymous page in the system. This structure introduces a level of indirection between anonymous pages and their position on a swap device.

An *anon_map* structure contains an array of pointers to anon structures and is used to treat a collection of anonymous pages as a unit. Access to an anonymous page structure is achieved by obtaining the correct anon structure from an anon_map. The anon structure contains the swap vnode and offset of the anonymous page.

4.5. tmpnode

A *tmpnode* contains information intrinsic to tmpfs file access. It is similar in content and function to an inode in other UNIX file systems (e.g., BSD Fast file system). All file-specific information is included in this structure, including file type, attributes, and size. Contained within a tmpnode is a vnode. Every tmpnode references an anon_map which allows the kernel to locate and access the file data. References to an offset within a tmpfs file are translated to the corresponding offset in an anon_map and passed to routines dealing with anonymous memory. In this way, an anon_map is used in similarly to the direct disk block array of an inode. tmpnodes are allocated out of kernel heap, as are all tmpfs control structures.

Directories in tmpfs are special instances of tmpnodes. A tmpfs directory structure consists of an array of filenames stored as character strings and their corresponding tmpnode pointers.

4.6. vnode segment (seg_vn)

SunOS treats individual mappings within a user address space in an object oriented manner, with public and private data and an operations vector to maintain the mapping. These objects are called "segments" and the routines in the "ops" vector are collectively termed the "segment driver".

The *seg_vn* structure and segment driver define a region of user address space that is mapped to a regular file. The seg_vn structure describes the range of a file being mapped, the type of mapping (e.g., shared or private), and its protections and access information. User mappings to regular files are established through the *mmap* system call.

The seg_vn data structure references an *anon_map* similarly in structure and use to tmpfs. When a mapping to a tmpfs file is first established, the seg_vn structure is initialised to point to the anon_map associated with the tmpfs file. This ensures that any change to the tmpfs file (e.g., truncation) is reflected in all seg_vn mappings to that file.

4.7. kernel mapping (seg_map)

The basic algorithm for SunOS read and write routines is for the system to first establish a mapping in kernel virtual address space to a vnode and offset, then copy the data from or to the kernel address as appropriate (i.e., for read or write, respectively). Kernel access to non-memory resident file data causes a page fault, which the seg_map driver handles by calling the appropriate file system routines to read in the data.

The kernel accesses file data much the same way a user process does when using file mappings. Users are presented with a consistent view of a file whether they are mapping the file directly or accessing it through read or write system calls.

The seg_map segment driver provides a structure for maintaining vnode and offset mappings of files and a way to resolve kernel page faults when this data is accessed. The seg_map driver maintains a cache of these mappings so that recently-accessed offsets within a file remain in memory, decreasing the amount of page fault activity.

5. Implementation

All file operations pass through the vnode layer, which in turn calls the appropriate tmpfs routine. In general, all operations that manipulate file control information (e.g., truncate, setattr, etc.) are handled directly by tmpfs. Tmpfs uses system virtual memory routines to read and write file data. Page faults are handled by anonymous memory code which reads in the data from a swap device if it is not memory resident.

Algorithms for some basic file system operations are outlined below.

5.1. mount

Tmpfs file systems are mounted in much the same manner as any other file system. As with other file systems, a vfs structure is allocated, initialised and added to the kernel's list of mounted file systems. A tmpfs-specific mount routine is then called, which allocates and initialises a tmpfs mount structure, and then allocates the root directory for the file system. A tmpfs mount point is the root of the entire tmpfs directory tree, which is always memory resident.

5.2. read/write

The offset and vnode for a particular tmpfs file are passed to tmpfs through the vnode layer from the read or write system call. The tmpfs read or write routine is called, which locates the anon structure for the specified offset. If a write operation is extending the file, or if a read tries to access an anon structure that does not yet exist (i.e., a hole exists in the file), an anon structure is allocated and initialised. If it is a write request, the anon_map is grown if it is not already large enough to cover the additional size. The true location of the tmpfs file page (i.e., the vnode and offset on the swap device) is found from the anon structure. A kernel mapping (using a seg_map structure) is made to the swap vnode and offset. The kernel then copies the data between the kernel and user address space. Because the kernel mapping has the vnode and offset of the swap device, if the file page needs to be faulted in from the swap device, the appropriate file system operations for that device will be executed.

5.3. Mapped files

Mappings to tmpfs files are handled in much the same way as in other file systems. A seg_vn structure is allocated to cover the specified vnode and offset range. However, tmpfs mappings are accessed differently than with other file systems.

With many file systems, a seg_vn structure contains the vnode and offset range corresponding to the file being mapped. With tmpfs, the segment is initialised with a null vnode. Some seg_vn segment driver routines assume that pages should be initialised to the vnode contained in the seg_vn structure, unless the vnode is null. If the vnode is set to be that of the tmpfs file, routines expecting to write the page out to a swap device (e.g., pageout daemon), would write the page to the tmpfs file with no effect. Instead, routines initialise pages with the swap vnode of the swap device.

Shared mappings to tmpfs files simply share the anon_map with the files tmpnode. Private mappings allocate a new anon_map and copy the pointers to the anon structures, so that copy-on-write operations can be performed.

6. Performance

All performance measurements were conducted a SPARCStation 1 configured with 16MB physical memory and 32MB of local (ufs) swap.

6.1. File system operations

Table 1 refers to results from a Sun internal test suite developed to verify basic operations of "NFS" file system implementations. Nine tests were used:

create	Create a directory tree 5 levels deep.
remove	Removes the directory tree from create.
lookup	stat a directory 250 times
setattr	chmod and stat 10 files 50 times each
read/write	write and close a 1MB file 10 times, then read the same file 10 times.

Test type	nfs *(sec)*	ufs *(sec)*	tmpfs *(sec)*
create	24.15	16.44	0.14
remove	20.23	6.94	0.80
lookups	0.45	0.22	0.22
setattr	19.23	22.31	0.48
write	135.22	25.26	2.71
read	1.88	1.76	1.78
readdirs	10.20	5.45	1.85
link/rename	14.98	13.48	0.23
symlink	19.84	19.93	0.24
statfs	3.96	0.27	0.26

Table 1: *nfs test suite*

readdir	read 200 files in a directory 200 times.
link/rename	rename, link and unlink 10 files 200 times.
symlink	create and read 400 symlinks on 10 files.
statfs	stat the tmpfs mount point 1500 times.

The create, remove, setattr, write, readdirs, link and symlink benchmarks all show an order of magnitude performance increase under tmpfs. This is because for tmpfs, these file system operations are performed completely within memory, but with ufs and nfs, some system I/O is required. The other operations (lookup, read, and statfs) do not show the same performance improvements largely because they take advantage of various caches maintained in the kernel.

6.2. File create and deletes

While the previous benchmark measured the component parts of file access, this benchmark measures overall access times. This benchmark was first used to compare file create and deletion times for various operating systems [Ous90a].

The benchmark opens a file, writes a specified amount of data to it, and closes the file. It then reopens the file, rereads the data, closes, and deletes the file. The numbers are the average of 100 runs.

File access under tmpfs show great performance gains over other file systems. As the file size grows, the difference in performance between file system types decreases. This is because as the file size increases, all of the file system read and write operations take greater advantage of the kernel page cache.

File size *(kilobytes)*	nfs *(ms)*	ufs *(ms)*	tmpfs *(ms)*
0	82.63	72.34	1.61
10	236.29	130.50	7.25
100	992.45	405.45	46.30
1024 (1MB)	15600.86	2622.76	446.10

Table 2: *File creates and deletes*

Compile type	nfs	ufs	tmpfs
large file	50.46	40.22	32.82
benchmark	50.72	47.98	38.52
kernel	39min49.9	32min27.45	27min.8.11

Table 3: *Typical compile times (in seconds)*

6.3. kernel compiles

Table 3 presents compilation measurements for various types of files with "/tmp" mounted from the listed file system types. The "large file" consisted of approximately 2400 lines of code. The "benchmark" compiled was the NFS test suite from the section above. The kernel was a complete kernel build from scratch.

Even though tmpfs is always faster than either ufs or nfs, the differences are not as great as with the previous benchmarks. This is because the compiler performance is affected more by the speed of the CPU and compiler rather than I/O rates to the file system. Also, there can be much system paging activity during compiles, causing tmpfs pages to be written to swap and decreasing the performance gains.

6.4. Performance discussion

Tmpfs performance varies depending on usage and machine configurations. A system with ample physical memory but slow disk or on a busy network, notices improvements from using tmpfs much more than a machine with minimal physical memory and a fast local disk. Applications that create and access files that fit within the available memory of a system have much faster performance than applications that create large files causing a demand for memory. When memory is in high demand, the system writes pages out and frees them for other uses. Pages associated with tmpfs files are just as likely to be written out to backing store as other file pages, minimising tmpfs performance gains.

Tmpfs essentially caches in memory those writes normally scheduled for a disk or network file system. Tmpfs provides the greatest performance gains for tasks that generate the greatest number of file control updates. Tmpfs never writes out control information for files, so directory and file manipulation is always a performance gain.

7. Summary

The tmpfs implementation meets its design goals. Tmpfs shows the performance gains associated with memory-based file systems but also provides significant advantages over RAM disk style file systems. Tmpfs uses memory efficiently because it is not a fixed size, and memory not used by tmpfs is available for other uses. It is tightly integrated with the virtual memory system and so takes advantage of system page cache and the kernel's resource management policies. Tmpfs uses many of the kernel's interfaces and facilities to perform file system operations. It provides additional file system space, and supports UNIX file semantics while remaining fully compatible with other file system types.

8. Acknowledgements

Thanks to Glenn Skinner, Howard Chartock, Bill Shannon, and Anil Shivalingiah, for their invaluable help and advice on this paper and the implementation of tmpfs. Also thanks to Larry McVoy for lively debate and suggestions, Marguerite Sprague for editing skills, and Herschel Shermis who wrote the initial implementation of tmpfs.

References

[Gin87a] Robert A. Gingell, Joseph P. Moran, and William A. Shannon, "Virtual Memory Architecture in SunOS," *Proceedings of the Summer 1987 Usenix Technical Conference*, Phoenix Arizona, USA, Usenix Association (June 1987).

[Kle86a] Steven R. Kleiman, "Vnodes: An Architecture for Multiple File Systems Types in Sun UNIX," *Proceedings of the Summer 1986 Usenix Technical Conference*, Phoenix Arizona, USA, Usenix Association (June 1986).

[McK90a] Marshall Kirk McKusick, Michael J. Karels, and Keith Bostic, "A Pageable Memory Based Filesystem," *Proceedings of the Summer 1990 Usenix Technical Conference*, Anaheim California, USA, Usenix Association (June 1990).

[Mor88a] Joseph P. Moran, "SunOS Virtual Memory Implementation," *Proceedings for Spring 1988 EUUG Conference*, London England, EUUG (Spring 1988).

[Ous90a] John K. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware," *Proceedings of the Summer 1990 Usenix Technical Conference*, Anaheim California, USA, Usenix Association (June 1990).

UNIX pipe extension on network for creating distributed functions

Istvan Szeker and Zoltan Vinceller

Computer Centre
Eotvos Lorand University
Budapest
Bogdanfy u. 10/B
H-1117 Hungary
h250sze@ella.uucp
h1093vin@ella.uucp

ABSTRACT

Present work highlights a method of organisation UNIX network wide distributed functions and network system tools. The Microcomputer System Software Research Group and the Information Systems Research Department of the Eotvos Lorand University Computer Centre have been taking part in a research project dealing with distributed information systems. The task of the group has been to investigate and develop methods and support tools, good for organisation of distributed information systems. One result was introduced in EUUG Spring 1990 conference (BUD Backtrackable UNIX Data Control).

1. Introduction

We faced the following problems:

- There are characteristics of certain problems, that it is reasonable using resources distributed in network – connected UNIX machines rather than local resources,
- Growth of the number of sources and the amount of data to be processed makes us reorganise the local data throughout the network,
- The sources of data are usually in different places,
- Certain problem solving methods are more intelligent and general realised by concurrently cooperating, network-wide-spread "function-elements" i.e. distributed functions.

The goal of the research work is threefold. The three parts of research has been running parallel.

First,

to develop network tools, good for research purposes: free from restrictions of the existing tools, ready to accept needs just arising in the network-tool user's field. Of curse it may seem to be unnecessary, but the reason of it can be seen on the parallel developing of the network tool and the distributed functions.

Second,

to research and develop technics for organising distributed functions. It means to get possibility to organise new set of functions, the elements of which – being part of different processes on one or several machines – can cooperate with each other. The development of this technics takes into account the needs and characteristics of the existing function sets, and the potential function sets.

Third,

to organise set/sets of functions: from existing ones, and from new ones, to be developed for distributed cooperation.

2. Networking

2.1. Network-pipe concept

Process – process communication method based on pipes. As one pipe is read or write only, there must be used two pipes for two-way communication between two processes. Our system prepares named pipe pairs in machines. The pipe-pairs can be opened – one for reading the other for writing – in the user process. Through the network the user processes in different machines ask to connect the bottom sides of the pipes. The connection is organised by the network system and supported as long as the cooperation of the processes is on. In this concept the user processes feel that they communicate with each other thorough named pipes. This is the "end to end" process communication. In the case of distributed-functions-cooperation more then two processes can take part in the cooperation. This is the "multi-process" communication. For the multi-process communication each process must open by one named pipe pair. One process is the "caller", the other processes are the "called". It is the case when we organise a logical star connection from the named pipes. The data block written to the pipe must be supplied with information about which connection to be used for transferring the block.

2.2. Scheme of network pipe organisation

For the sake of simplicity we described a scheme of four logical levels: The logical levels scheme doesn't correspond to the realisation scheme. The logical levels exchange information thorough shared memory: "SwitchShare"

- **User level:** This level in one machine contains the user program, which cooperates with the user programs running in other machines (in the user level of their native machine). The functions which are for networkwide usage (for distributed usage) are developed by common programming scheme, so the functions are formed into a set and added with certain user (strategy) and network support program elements. This level deals with the user data transformation if necessary.
- **"Pipe-manager and pipe-drive" level:** Deamon programs in this level which organise the bottom side of the named pipes, form user blocks to/from transport units – packages. The pipe drive makes the switch of the packages to be send to another drive. The switch takes place on the base of what is called "logical channel table". In this context the pipe drive plays the same logical role as the communication's drive. There is one deamon for one named pipe – two for communication with one process. The drive and management functions are in one program – of curse different for pipes of different directions.
- **"Network management" level:** Deamon program (one for one machine) which sets up the configuration table (which keeps the network topology information), update logical channel table, maintains the network integrity (network level).
- **"Data-link and communication-drive" level:** Deamon programs – one for output (send direction), one for input (receive direction) – which are in communication contact with the corresponding data link, and communication drive programs in the neighbour machines. This level also makes switch. The "data-link and communication-drive" level of course has responsibility for the quality of transferring data thorough the network.

2.3. Network pipe functioning

2.3.1. The network pipe forms

The network pipe concept makes programs able to cooperate with each other by exchanging information formally writing and reading to/from a pair of named pipes which are connected with a similar pair of named pipes being somewhere in the network (where the partner user program is running). The connection is organised with software tool – the object of this report. The network pipe allows the communication between pairs of program parts: the function representative at the caller and the function executer at the called machine. The data block has a field: user code – the identifier of correspondence between these parts. This field is up to the user. The network pipe has two forms:

1. **End to end pipe:** One user program needs creating information connection with another user program, (running or able to run) in some machine anywhere in the connected network. The connection must support two direction data flow. The settled network pipe means for the user programs that they have a pair of named pipes. The names of the pairs can be derived from the same name by making name extensions different for in and out directions. The user programs write/read information is in forms of data blocks which are pure user information added with header. There are no restrictions by the network pipe system on the size of information block.
2. **Multi ended pipe:** One user program needs creating information connections with another user programs (running or able to run) as above. The initiator program is in the center of the the logically

star shaped network pipe. It consists of several "end to end" pipe. The initiator program is in MASTER position. The other programs are in SLAVE position. The header of the data block has a field for logical channel number. The logical channel number is used to distinguish between the star directions. The MASTER program uses all settled logical channels (one for one star direction), each SLAVE program uses by one logical channel.

2.3.2. The data block types

The user communicates with the network system and the partner program on the user level thorough network pipe. We use three types of block:

1. **Data block:** It contains information – related to the user level – in the data field.
2. **Command block:** It contains information – necessary to organise the work on the network system – in the data field.
3. **Status block:** It contains information about
   - the result of fulfilling the issued command,
   - current network status,
   - the changes of the network situation, concerning the user,
   - return code of a called function or program.

2.3.3. Network commands. Basic network functions

Network commands issued by the user are the following:

- **Set connection to a user process being in the network.** This command initiates the network pipe. In case of multi-ended pipe, the user must issue this command once for every partner in the network. One program goes into connection with the needed partner program by asking the net system to settle connection to the machine where the would-be partner program is. This program is the caller, the other is the called one. The set connection command makes the partner program start (if it isn't active) which (for the sake of unity of work) issues a service offer command. The service is immediately accepted and connected to the network pipe end. The connection is managed by the net system through machines the connection goes along.
- **Service offer.** One program which is potentially called partner of a caller tells the net system to take it into account (administer) as one would-be called. The program in such a way becomes active. A program issues this command in two situations:
  - by its own initiative,
  - by a call from somewhere in the network.
- **Release connection.** This command can be issued either by the caller or the user program being called.
- End of service offer.
- Ask network status.

Network functions called by the user level are the following:

- **Write block to the network pipe.** The user writes command or data block to the named pipe. The command block is interpreted by the "network management". The data block is transferred thorough the network to the partner user program at the other end of the network pipe. The write has "wait" character. The network system reads the bottom end of the named pipe. After writing the data block is in the network system. The partner user program gets the data having delivered to it. The network system reads the block in two steps. First it reads the header, and determines the length of the remaining part of the block (the data field) and reads it.
- **Read block from the network pipe** The user program reads a block from the named pipe. The read must be done in two steps.

2.3.4. Layout of one logical channel

One logical channel laid down from the caller program to a machine to a called set of functions. If the caller want to work with sets of functions (the representatives of the same set in different machines in the network) there must be settled by one logical channel to each set of functions. It is the case of the multi ended-pipe. The named pipe in the caller machine can house several logical channels. The network system organises the logical channels (analogous to the virtual channels in the ISO-OSI). The user program however sees the network pipe which houses the logical channel(s).

2.3.5. Programing considerations, implementation in UNIX environment

The different levels of the system must be active in one machine. It was easy to organise. The sufficient problem was to organise information communication between the levels. It is done by using the features of the UNIX system. The network organisation levels scheme (given in 2.2) is different a bit from the program levels scheme.

2.3.5.1. The programming levels

- **User level** As in the pipe scheme. it is out of the network system. So it can be thought as the outer world relating to the network system in one machine. In this level there is a program running under UNIX.
- **Drivers for connection to the outer world:** It consist of two pipe scheme levels
  1. **The pipe manager and drive:** The pipe manager and drive appears to be as one program deamon for one pair of named pipes.
  2. **The data link and communication drive:** The data link has one receiver deamon and one sender deamon for one RS232C connection.

They are the same from the point of view of programming. They give the interface between the network system in one machine and the outher world of the network system. So the user level of the machine and all what can be find in an another machine turns out to be the outher world. The pipe and communication drivers themselves make switch of the data between each other by using the logical channel table placed in shared memory, managed by network management.

- **Network management.** The network management is a deamon program too. It is the same as described in the network pipe scheme.

2.3.5.2. Exchange information between program levels, program parts

- **User level – pipe manager and drive** The user program has connection to the network system via named pipe pair. It searches for a free pipe pair (using administration information available in shared memory: Config_Shared), opens for writing and reading.
- **Pipe manager and drive, data link and communication drive – network manager.** The information, concerning the network management accessible for it via the what is called Switch_Shared – the shared memory. The network manager gets information usually in form of command block.
- **Pipe manager and drive, data link and communication drive.** These parts must communicate with each other in certain situations:
  - **Pipe manager and drive** – data link and communication drive (vice versa) exchange user data blocks (disassembled to packages) moving in logical channels, via the Switch_Shared.
  - **Pipe manager and drive** – pipe manager and drive exchange information similar to the pipe manager and drive – communication link and drive connection. As we can see later the network call can remain in the caller machine in unified – network organised – way. In this case the logical channel, coming from the pipe manager doesn't goes to a communication drive but back to the pipe manager (to the called program).
  - **Data link and communication drive** – data link and communication drive connection is organised in the case of stretching the logical channel via transit machine. The communication is organised again via the Switch_Shared. So it can be seen, that the pipe manager and drive and the data link and communication drive takes the same position in the system.
  - **Communication drive receiver** – same communication drive sender. The receiver and sender daemons exchange information via "Short_Shared". The information is needed to support the data link algorithm.

2.3.5.3. Access permissions

The caller programs must have well defined access permissions on the machines they want to work with. There are two methods to control network user access. These methods complete each other:

- **Using NETPASSWD file on every machine.** The set connection control block contains user name, and password (in encrypted way), set by the user. At the called machine the what is called NETPASSWD file describes "userid" and "groupid" of the real user (in the /etc/passwd file), standing for the net-user. The called program at the user level sets effective "userid" and "groupid" to the defined real "userid" and "groupid".
- **Legal user list is given by "service offer" command.** The program at the called machine (or to be called) gives a list of users whose call can only be accepted. It restricts the group of users to be served in general or during a certain period. The list of users is up to the program, writing service offer to the net system. Of course the net system later uses the NETPASSWD file.

3. Distributed functions

3.1. Problem evolution

We discuss this subject in the order of the problems arisen. The problems were solved in different principal and practical completeness. The problems of the first phase and the last phase are in different level of intelligence and software technology.

3.1.1. First problem: Remote file access, remote I/O

The files are specified by their local pathname in a certain machine. In the connected network the file specification is the concatenation of the routing list (from the machine where the specification is made to the target machine where the file resides) and the local file specification. In case of using remote file access, we must have linked the program and the library, which contains the local implementation "requester" function set. This set would deal with the function calls filter the remote calls etc. As the file specification needed at the file open process, the file open with network file specification would initiate setting up logical channel through the network to the remote file access function set: the "server" function set on the target machine. The "requester" and "server" sets would agree about the function to be done, the parameters, the results etc. and cooperate. After opening a file via network the file is referred from the user call by usual way: by FID. The FID not the real FID, it is the logical channel identifier supplied with certain offset. This offset makes it possible to filter the network file references. Remote I/O organisation is the question of using special devices specified bye network address. It can be a simple problem if the input and output device is the same. The organisation of the "server" and the "requester" sets will be discussed later.

3.1.2. Second problem: Multiplicated remote file access

It is the question of complicity of administration to deal with several network files. It is not a distributed function management. It is only a method of local file access extension to network. For instance the local file functions are working in WAIT fashion, the network equivalents are working in WAIT fashion too. So the file access functions work in turn. Remote I/O organisation in case of different input and output devices belongs to this problem.

3.1.3. Third problem: Single remote process execution

Remote process execution means that we force certain program execution on some other machine in the network. We call the "far shell" (with restricted capabilities) program – it is a local program – which gets the user parameters defining the remote program and its parameters. The first parameter is the program network pathname. The further parameters are the parameters of the remote program. It should be supplied some parameters concerning to the organisation remote execution. The remote shell goes in contact with its remote agent on the target machine passes the parameters to execute the program needed by the user. The far shell program organises, keeps track of the programs in the network. Existing, old programs, started by the far shell are not able to interpret the network file specifications, so the files, defined by parameters, passed to them must be local to the called program. In this case only the standard I/O can be redirected to an another machine in the network. A new program written for network purposes can take into account the network file specification. There are two ways of organisation of the remote program execution:

- **Off line execution:**
  - the standard I/O of the remote program is on the target machine, directed to certain files or devices, defined by redirection information.
  - the remote program only started with no return information about the success of execution but only the success of the start.
- **On line execution:**
  - the standard I/O of the remote program machine will be connected to the standard I/O of the caller remote shell. Redirection can be made to the files and devices on machine of the caller far shell.
  - there are return information about the success of the start and the success of the execution.

3.1.4. Fourth problem: Multiplicated remote process execution

Piped command line with network-spread command elements.

Multiplicated remote process execution means that several programs, locating in different machines in the network, can be executed the standard I/O of which are chained, as specified in the piped command line. The programs, given in the chain, have local or network program pathname depending on their place in the net. The trivial solution is to call the far shell (for single execution) one after each in a piped command line with the UNIX shell whatever is. The parameters to the far shells are the programs to be executed in pipe connection. The effect will be what we wanted, but the far shell's several examples will be running in the machine. The better solution of network-piped managing and execution of several programs is to do it with one far shell (for multiple execution), more complicated. One logical pipe between two programs located in machines, different from the caller machine, lines thorough the caller machine (and the far shell in it) and consist of two "end to end" network pipes. The far shell interprets the redirection and organises it the networked basis. Opposite to the previous problem it can be organised remote piped command line execution only in "on line execution" way. It is because the caller far shell program organises and switches the network pipes between processes.

3.1.5. Fifth problem: Remote functions (set of remote functions)

The user program can call functions which is executed somewhere in the network. A number of functions to be called via network are gathered to a set of functions. All initialisation, administration, management are common in such a way. But it is reasonable to set up sets including functions relative in some way. The criteria of grouping functions to a set can be the common goal of work but it is not not necessary. The set is added with certain new manager functions as opening, closing the set. Opening means setting up logical information channel via network to the target machine to the just initiating remote function set. The remote set houses the real working function's code. At open the user program specifies the target machine and the function set the elements of which are to be executed. The following calls of the elements of the set mean calls to the elements to be executed on this target machine. If only remote functions available, there are no extra parameters needed to distinguish between the same local and remote functions. If both local and remote functions can be called, the user program must supply information necessary for separation the local and remote calls. This question would be discussed later.

3.1.6. Sixth problem: Multiplicated remote functions

The user can call the elements of the set to work on different machines of the network in mixed way. It means that several function set opens must be in effect at the same time. There must be an administration organisation which performs the following:

- sets up and maintains several logical channels, going to different network machines.
- returns a parameter to the caller program which makes possible to refer to the just opened remote function set.
- makes correspondence between the parameter and the logical channel, switches the information blocks between the calling place and the logical channel and back.

The parameter can be either explicit or implicit one, composited in other parameter. It is a very interesting problem in case of networkisation of existing functions. Using explicit parameter in this case means changing the syntax which is not a good solution. In the file access functions the solution was using implicit parameter passed by FID).

3.1.7. Seventh problem: Distributed functions (set of distributed functions)

The most interesting problem is the organisation of distributed functions. Calling a distributed function means calling a function witch can be performed by performing several (in usual case) functions being in different parts of the network. There must be organised a relatively complicated management of the partial and networkwide-spread functions on the machine of the user program, making the call. The functions can be divided into three types:

- **strategy functions:** The central management is supported by these functions on organisation and schedule of the executer functions. These make possible for instance to execute partial functions in conditional and delayed way i.e.:
  - one partial function is executed in dependency of the result of previously executed functions.
  - the partial function's results in the machines not fixed until the success of all partial functions turns out to be obvious to the central management. The delayed results can be reset, "undone" in case of no hope of total success.
- **executer functions:** These functions are the functions the user want to execute and which are composed of partial tasks (function service programs) on the machines.
- **auxiliary management functions:** The function set must contain management and network organisation elements with the complexity of management of multiplicated remote functions (or a bit higher). The "network function set open" call for instance must pass the "distributed function configuration" and the open is spread to several simple "function set opens". There is analogous case with the "close" (without configuration parameter).

The strategy and executer functions are problem oriented and subject of investigation on the problem field.

3.2. User level distributed functions scheme and organising methodology

3.2.1. Structure, offered skeleton

To work in the network with distributed functions there must be several user level program-parts located in several machines and cooperating with each other. The user program works with a set of executer functions. As mentioned above, the set contains user oriented functions (strategy, executer). In the first approach the user calls the executer functions only, but it is not the truth when the user develops the system of his own, knowing that it is planned to work on network. The strategy, the executer functions form the user elements of the distributed functions scheme. The user elements are surrounded by certain programming elements on the user level which makes it possible to the executer functions to live on the network. These elements called system elements.

3.2.1.1. User elements

The user elements are in the user's responsibility. The different parts must be located in the caller machine and in the called machines. Of curse one machine can be caller and called at the same time. One distributed function set's user elements are the following:

- One caller program manager on one machine. The caller program manager accepts the function calls from the user, decomposes it to partial functions, controls their execution according to the results of execution of them. At the caller program there are representatives of the user functions. This is the strategy part.
- **Function service representative on the caller machine.** This is the starting point of one simple end to end pipe. It gives a function interface for the local caller. It makes the necessary data conversions. Sends and receives data to and from the network pipe. These data are user concerned.
- **Function service agent (representative) on the called machine.** This is the counterpart of that of the caller machine. It receives and sends user related data from and to the network pipe. Makes necessary data conversions, calls the function service program. In reality this is the agent of the service functions.
- **Service functions on the machines.** These functions are called from the central caller program via the network pipe. These programs usually doesn't know anything about the method they get the parameters, and the way of returning the result.

3.2.1.2. System elements

The user elements work with the network by using the system elements support. The system elements consist of the necessary organiser and administer functions dealing with the network pipe. There are different system elements on the calling and the called machines. Usually system elements are the same in different distributed function's sets. These make the skeleton for housing the user elements. The difference can be in the size of tables describing the logical channels.

3.2.1.3. Local – remote considerations

It is clear that functions to be executed somewhere not in the calling machine must be treated through the network. Local functions can be estimated to be either as local or network functions.

Local function organisation in direct way:

The user program can call the local functions in simple way with the name as described in the programming manual of the function.

The network equivalents can be called:

- By some other name, different from that of the local functions. The user must distinguish which function is to be called: local or remote. The local functions work as usual, they don't know anything about their network equivalents existing. The remote functions work as described in the "multiplicated remote functions" paragraph. In this case extra parameters should be passed for remote functions either in implicit or explicit way. The user program must be "re-programmed" according to the needs of new (remote) function calls and "re-linked" with the library of the remote functions set.
- By the same name as the local functions. The functions set system elements must distinguish between the local and remote calls. The distinction made by some extra parameter. Certain parameter value means that the call is local function call. This value elaborated by the system element on initialisation of the functions and returns to the user. The parameter can be either explicit or implicit. The user program don't need to care about the type of the call. To use remote and local functions with explicit extra parameter, the program in source code must be "re-edited" concerning to the call with more parameters and "re-linked" with the library of the remote functions set. In case of implicit extra parameter, there must be certain not used parameter value range at some of the old parameters, large enough to code not only the original meaning, but the local – remote type too.

  Using implicit extra parameter is the best solution in reorganisation of the existing functions. This can give the highest level of compatibility between local functions and their network equivalents. It is necessary only to "re-link" the object code of the program with the necessary libraries.

If we develop a new set of functions to work in network there is no need to use implicit extra parameter. The system elements at the caller machine are responsible for to make local or network call by making distinction on the base of the extra parameter.

Local function organisation via network pipe system:

The local functions can be called by the same way as the functions to be executed in a machine other then the caller. There are no differences between the methods and network organisation of local and network calls. The organised network pipe goes down to the net system and returns back to the called function set. This is the simpliest (and the best) solution especially in case of developing new function sets. Of course it gives some extra burden to the network system.

4. Implementations

4.1. Multiplicated remote file access (basic I/O functions -> net I/O)

The existing file system functions were "networked" so there are available the network equivalents of the functions in library form. The user programs even in object code can remain unchanged. Network pathname can be used in file specification. The old programs must be relinked with the new networked function library. The user program makes a call for open a file specified somewhere in the network. Afterwards the following calls relating to this file (by FID) mean working the network system, calling the function set representative at the executer end. The information: function parameters, user data, and return code are transferred by the network system thorough network pipe. The user in this case doesn't sees the pipe because its usage is hidden in the function set. Several files, residing in different machines of the network, can be opened and accessed.

4.2. Multiplicated remote process execution

This is organised by the implemented far shell. In the calling machine a multi ended network pipe is organised; the standard outputs and inputs of the programs are connected by the "far shell caller". The implemented far shell doesn't interpret shell script. It is because of programming reasons.

4.2.1. Examples of a piped line with programs to be run in different machines of the network:

4.2.1.1. Example 1.

- The following command interpreted by UUX called from machine "a":

  ```
  uux 'c!diff b!/usr/dan/xyz c!/usr/dan/xyz >d!xyz.diff'
  ```

  UUX organises the execution of the command line in the following way:

  - copies the network-source (non-local) files to the native machine (into uucp default directory) of the program,
  - executes the program with the input files according to the previous step and puts the output file (in case of being non-local) into the uucp default directory,
  - copies the output file (if non-local) to the destination machine.
- The working equivalent of this command interpreted by using far shell (running on machine "a"): The input files remain in their original places, on-line information connection (logical channels thorough network pipe) is organised:
  - trivial method:

```
fsh 'b!cat /usr/dan/xyz'|fsh 'c!diff - /usr/dan/xyz'|fsh 'd!cat >xyz.diff'
```

(redirection is made by the UNIX shell on the "d" machine)

  - non trivial method:

```
nsh 'b!cat /usr/dan/xyz|c!diff - /usr/dan/xyz|d!cat >xyz.diff'
```

(redirection is made by the UNIX shell on the "d" machine again)

```
nsh 'c!diff - /usr/dan/xyz <b!/usr/dan/xyz >d!xyz.diff'
```

(network-redirection made by the far shell)

4.2.1.2. Example 2. Text processing.

- Processors tbl, eqn, troff are in the machines "a", "b", "c" called from machine "d" (input and output files are in "d"):
- Methods:
  - trivial method:

  ```
  fsh a!tbl|fsh b!eqn|fsh c!troff -ms  <infile >outfile
  ```

  (redirections made by the UNIX shell – for the infile and outfile are local)

  - non trivial method:

  ```
  nsh 'a!tbl|b!eqn|c!troff -ms <infile >outfile'
  ```

  (redirections interpreted and made by the far shell)

  ```
  nsh 'a!tbl|b!eqn|c!troff -ms' <infile >outfile
  ```

  (redirections made by the UNIX shell)

5. Current research work, developments

- Distributed functions set being under development: the distributed BUD (Backtrackable UNIX Data Control introduced in EUUG Spring 1990 conference, Munich by Zsolt Hernath and Szokolov Makar) from local BUD,
- Dependency between the level of the functions and the intelligence of the distributed equivalent,
- Formalisation of the method of function networking (distributing),
- Implementing far shell, which interprets shell script,
- Developing the pipe-networking on the base of standard and fabric tools,
- Connecting to the UNIX-network non UNIX-elements (as users of functions acting in UNIX environments),

6. The perspectives

- research of new distributed applications,
- Research of distributed functions strategies,
- Method of formalising network system management of distributed functions,
- Research of network-wide self organising programs.

FlexFAX – A Network-based Facsimile Service

Samuel J. Leffler

Silicon Graphics, Inc.
2011 N. Shoreline Boulevard
P.O. Box 7311
Mountain View, CA 94039-7311
sam@okeeffe.berkeley.edu

ABSTRACT

The FAX machine is now a standard piece of office equipment for businesses and is becoming more commonplace in the home. FAX machines with a computer interface are relatively expensive, but FAX modems are not. This paper describes FlexFAX, an effort to provide low-cost, easy to use, FAX communication services to computer-based users through a network-based FAX server. FlexFAX uses commercially available FAX modems and can run on any UNIX-based platform to which a FAX modem is interfaced.

Introduction

FlexFAX is a UNIX-based software system that provides a network-accessible facsimile service. FAX transmission is as simple as printing the contents of a file or sending electronic mail. When combined with a suitable imaging processor, such as a PostScript interpreter, a wide range of documents can be sent as facsimile. Facsimile reception is automatically handled by a server and administrators may request automatic notification of arriving facsimile. When suitable routing information is included in a received FAX, FlexFAX automatically dispatches the message to the user, or users, through electronic mail. Users may interactively examine facsimile on a graphics display and print facsimile on common PostScript printers.

The facilities offered by FlexFAX are similar to commercially available systems for personal computers; e.g. [Aba89a]. These systems however do not serve a community of users, but are, instead, designed for personal use; i.e. one modem, one user. Server-based systems have been built for communities of users [Com89a, Res89a, Sol90a]; but none is as general-purpose as FlexFAX – [Sol90a] does not support automatic routing of incoming messages, while [Res89a] requires special-purpose hardware and communication services to do automatic routing of incoming facsimile. FlexFAX supports automatic routing of incoming facsimile, using a routing scheme that works with most any FAX modem and requires only the cooperation of the sender.

Section 1 of this paper describes the origin of the system and identifies the important features of a good facsimile system. Section 2 briefly describes existing systems. Section 3 gives an overview of the architecture of the system, identifying the major software components and their interactions. Section 4 discusses issues involved in imaging facsimile, while section 5 covers the problem of printing facsimile on a PostScript printer. Section 6 describes the scheme used by FlexFAX to do automatic routing of incoming facsimile. Section 7 summarizes current and future work.

1. Background

Silicon Graphics, like most companies, has a number of facsimile machines scattered around the facilities. These machines tend to be located in each group or department, with access treated like a copy machine: first come first served, through a sign-up sheet, or through a facsimile machine owner. There is also a central "FAX Mail service" similar to a company copy service. This organization typically results either in under-utilization of resources (too many FAX machines) or extended delays and wasted time (not enough machines). Furthermore, incoming facsimile are always printed; all too often resulting in piles of unwanted or undelivered FAX messages. In response to this situation (and an abandoned FAX modem that was left in the corner of a colleague's office) FlexFAX was conceived as an attempt to eliminate all FAX machines at Silicon Graphics.

For FlexFAX to be a successful replacement for the installed facsimile machines however, the services it offers need to be equivalent or better to those available with the existing scheme.† In particular, the typical services offered by a facsimile machine can be itemized as:

1. Transmission of printed matter (text and graphics),
2. Reception of printed matter,
3. Printing of received materials, and
4. Scanning of printed matter.

Furthermore, any computerized alternative to a FAX machine must preserve the properties that make FAX communication popular:

- Immediacy of delivery (no postal system, electronic relaying, or other queueing),
- Universality and simplicity of addressing (phone numbers and surnames),
- Low cost of communication (the cost of a phone call),
- Unattended operation (no "telephone tag" or answering machines).

2. Existing Systems

FAX modems and software for personal computers have been widely available for some time [Aba89a, Res89a, Com89a]. These systems use one of two types of modem hardware: an add-in board or an external device that is connected through a serial port. Examples of the former are the Hayes JT FAX 9600B, the Brooktrout TR111/112 family of boards, and the Intel Connection Coprocessor; an example of the latter is the Abaton Interfax 24/96 modem (made by Everex Systems). PC- and Macintosh-based add-in boards are more prevalent than external modems for the obvious economic reasons, but also because there is no standard for communicating with the external modem. This situation will change with the availability of "Class 1" and "Class 2" modems (probably sometime this fall).

The software for personal computer-based facsimile systems varies, but typically provides the following features:

- Unattended transmission and reception of facsimile,
- Viewing of facsimile,
- Spooling with scheduling capabilities (for sending when phone rates are low),
- Printing to local printers,
- Dialing directories,

Other features often found are: support for locally-connected scanners, notification of transmission/reception, enhanced imaging capabilities (e.g. support for PostScript). A few products provide gateway facilities and automatic routing of incoming messages with a Direct Inward Dialing (DID) telephone trunk line – a facility supplied by the phone company whereby blocks of phone numbers are assigned to one incoming phone line.

The most common scenario is a personal computer with a *dedicated* FAX modem connected to an office line on a shared or dedicated basis – basically the standard FAX machine model, but with a computer. This usage model simplifies/eliminates certain problems (access control, routing, scheduling, accounting), but limits potential resource sharing.

3. System Architecture

A simplified view of the FlexFAX architecture is shown in Figure 1. One FAX server process exists for each FAX modem on a system. This process implements the CCITT Group 3 protocols [CCI80a] for transmitting and receiving facsimile, and is also responsible for handling the FAX modem. An ancillary queueing agent is responsible for handling system-related issues such as spooling, scheduling, notification, and administrative tasks. When multiple FAX modems are present, the queueing agent is responsible for balancing load and, potentially, for selecting the best FAX server/modem to use in sending a document.

† Actually, the promise of eliminating wasted unrecyclable paper alone might have been enough to make the plan worthwhile.

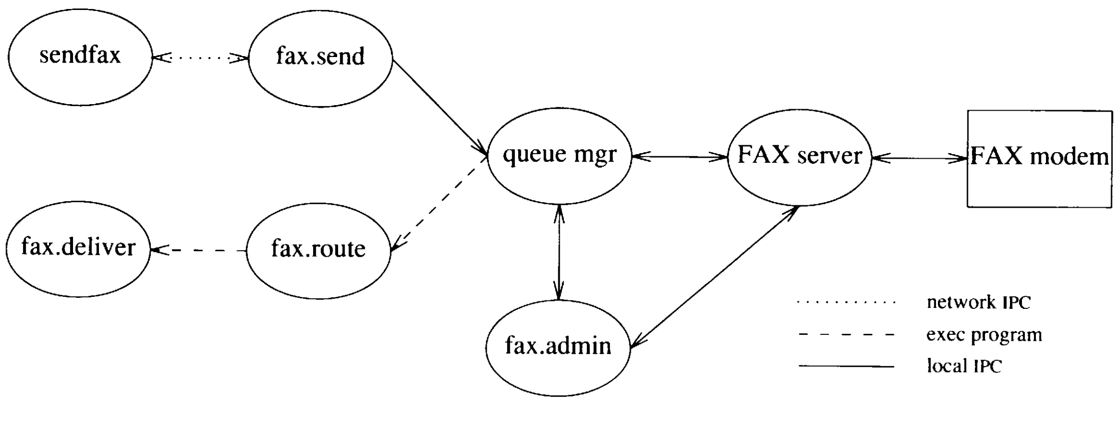


Figure 1: *FlexFAX Architecture*

3.1. Transmission

To send a facsimile a user invokes the *sendfax* application, specifying a set of files and a set of destinations (possibly through a mailing list). *sendfax* establishes a network connection to a transmission agent (*fax.send* in Figure 1) and submits the job for transmission. The information passed to the transmission agent includes the destination facsimile machines (i.e. phone numbers), the time to send the documents, and the information required to generate cover sheet(s). The documents to be transmitted are passed to the server either pre-formatted for transmission or in a format that requires imaging support to generate a facsimile. Formatting the documents on the client end usually reduces load on the server. In some cases however, it may be preferable and/or necessary to format and image a document on the server side; for example, when an imaging service is available only on the server. There are also issues related to generating a match between the imaged document and the capabilities of the receiving FAX machines; e.g. matching paper width and/or length.

The transmission agent places the documents to be sent in a spooling area and crafts a *job description* file that is then sent to the queueing agent (*queue mgr* in Figure 1). The queueing agent then schedules the job for transmission, potentially at some time in the future. When a job is ready to be sent, it is passed to a FAX server which manages the transmission process. As documents are transmitted by the FAX server the queueing agent is notified and the associated job's state is updated. Jobs that need to be retransmitted or rescheduled are handled by the queueing agent.

3.2. Reception

An incoming facsimile is handled initially by the FAX server process that services the modem on which the call is received. This process answers the phone, manages the Group 3 receiving protocol, and passes the document pages to the queueing agent. The queueing agent then spawns a program that is responsible for routing incoming facsimile to a user. One of the important features of FlexFAX is that it can automatically route an incoming facsimile directly to a user by scanning the facsimile for addressing information. This information takes the form of a bar code that usually appears on the cover page of a facsimile. A database of registered bar codes is used to map a bar code to a user who then receives the facsimile as an electronic mail message (section 6 discusses the routing scheme in more detail.) If a received facsimile can not be automatically routed to its destination, it is dispatched to an administrator named the *FaxMaster*. (The *FaxMaster* is modeled after the *PostMaster* user associated with electronic mail systems.)

Received facsimile can be viewed on a bitmap display and printed with various utility programs. The efficient printing of facsimile on PostScript printers turns out to be an interesting problem; it is discussed in section 5.

3.3. Document Storage

The Group 3 facsimile protocols specify a compressed format to be used in sending and receiving data. This format is encapsulated in the Class F recommendation [Tec89a] of the Tagged Image File Format (TIFF) [Cor88a]; a file format that augments the compressed data with information about the structure, photometric interpretation, and origin of the data. Multiple document pages may be stored in a single TIFF file, simplifying management of documents.

4. Imaging Facsimile

Aside from handling the CCITT facsimile protocols, the main effort required to build a facsimile service is in providing tools to image outgoing facsimile. FlexFAX handles normal ASCII text, output from the *nroff* and *troff* family of text processing tools, as well as colour and greyscale images in a variety of formats.

4.1. Imaging Text

The conversion of text-oriented input is easily accomplished by canibalizing a screen-oriented previewer. Public domain previewing programs that handle *Device Independent Troff* (DIT) [Ker84a] or that handle TeX DVI [Knu84a] are readily available. The main stumbling block in realizing high quality output is the availability of high quality fonts. Publicly available fonts, such as those included with the X Window System [Sch86a], often do not include high quality, high resolution fonts, or these fonts lack certain glyphs. It is laborious to cleanup and hand-tune appropriate fonts; consequently most vendors charge a lot of money for these fonts. Automated font machinery such as the Adobe Type Manager (ATM) [Cor90a] Folio font manager, [Cor89a], or Apple TrueType font system [Com90a] is a desirable alternative to bitmap fonts, but is also rather expensive. The cost of add-on fonts and font technology is a particularly good reason to image all facsimile on the FAX server machine rather than the client machine. This way only a single copy of the fonts and software need to be purchased, although licensing may restrict this type of use.

Another issue in imaging text is the font *face* to use in imaging the facsimile. Typical laser printers have a 300 dots/inch (dpi) output resolution while Group 3 facsimile machines support 200 dpi in the horizontal direction and either 100, 200, or 400 lines/inch (lpi) in the vertical direction. Each increase in resolution effectively doubles the time it takes to transmit a page. Consequently most facsimile are not transmitted at the highest resolution and so printing on a laser printer results in a loss of information.† A recent study by Adobe [Jey90a] considered this problem in terms of the most readable Adobe font faces to use when imaging a facsimile that is to be sent at various resolutions and then printed on a 300 dpi laser printer at the receiving end. Their results, while very subjective, indicated that Lucida and Lucida Sans fonts were the most readable, although several were found to show good results.

The *troff* conversion program used in FlexFAX was derived from a local previewer and uses a combination of NeWS, X, and Silicon Graphics-proprietary fonts. Both text and graphics commands are correctly processed so that the *pic*, *tbl*, *eqn*, and *grap* preprocessing tools can be used in preparing facsimile. For example, the following could be used to format this paper for transmission:

```
refer ?.t | pic | eqn | tbl | troff -ms | dit2tif -o fax.tif
```

(*dit2tif* is a conversion program that generates a Class F TIFF file from *troff* output.)

4.2. Converting Images

The handling of colour and greyscale images is more difficult than the conversion of text and text-oriented graphics. Since Group 3 facsimile are bilevel images this conversion requires the use of *halftoning* or *dithering* techniques [Uli87a]. The following scheme is used by FlexFAX:

1. Convert colour images to greyscale by taking a simple percentage combination of the red-green-blue components; by default 28-59-11 percent, respectively.
2. Resize the image to fit the page (or the desired area); a filtered scaling algorithm is used.
3. Apply a high pass filter to sharpen features.
4. Correct the greyscale colour based on an idealized scanner.
5. Convert from greyscale to bilevel with an ordered dither.

Step 2 uses a filtered scaling algorithm with blur (when scaling up); by default a triangular filter is used. Step 3 uses a 3 by 3 kernel; its' use is very important in producing good bilevel images. Step 4 passes the data through a transfer function to alter colour so that it approximates what would be obtained by scanning the image on an idealized facsimile machine. Step 5 uses an 8 by 8 ordered dither over a rectangular grid. The choice of an ordered dither was based on subjective experiments with a sampling of scanned and computer-generated images.

The default conversion scheme is done automatically by FlexFAX when it is given an image to transmit. Changes to the default parameters or alternative schemes, such as using dot diffusion [Knu87a] in step 5,

† Many Group 3 facsimile machines do not support 400 lpi, in which case there will always be a resolution loss due to transmission.

might be more appropriate for converting certain images, in which case the sender can select from a set of image processing tools that are part of the programming environment on a Silicon Graphics machine [Cor89b]. There are also public domain image manipulation packages available such as [Pos89a].

5. Printing Facsimile

Facsimile are large bilevel images; typically 1728 by 1066 lines per page.† While only 1 bit deep, this works out to ~225 Kbytes/page or ~457 Kbytes/page at high resolution. Most printers found in the workstation environment today are connected to a host through a 9600 baud or 19200 baud serial line. Assuming a 9600 baud serial line, it takes 192 seconds or 3.2 minutes to transfer the raw bits that make up a single facsimile page – assuming full transfer rates. At 19200 baud this rate drops to 1.6 minutes; a more tolerable interval.

Unfortunately for this application, many of the printers are PostScript printers and PostScript does not directly support the transfer and printing of binary images. Instead the image must be expressed in the PostScript language and, typically, presented to the printer as an ASCII representation.‡ The most straightforward scheme for printing an image is to use the PostScript *image* operator. When presented in this form, the data is most conveniently represented as an ASCII hexadecimal string that represents the binary image data. In this format twice as much data must be transferred to the printer as the raw format since each byte of data must be specified as two ASCII hexadecimal digits (two bytes). Furthermore, the PostScript interpreter must parse the input data, execute the PostScript language operation (the *image* operator), and, potentially, resize the image to fit the physical characteristics of the printer. Experiments with the printing of facsimile data in this manner indicated an average printing time of 15 minutes per low resolution page for a printer connected by 9600 baud line to a fast host. Time can vary significantly depending on the memory configuration, printer cpu, and whether or not any scaling is required.

The first solution to this problem that was tried was to implement the Group 3 facsimile decompression algorithm in PostScript. The decompression algorithm was then prepended to each print job and used to interpret raw binary data that was supplied to a PostScript *image* operator (as before). This solution represents a tradeoff between the time spent interpreting PostScript code and the time spent transferring data along the serial line. The PostScript code was carefully tuned to make the inner loop of the decompression algorithm as low cost as possible – all symbols were bound (to avoid dictionary name lookups), conditional and looping statements were minimized, etc. An average low resolution facsimile page comprised mostly of text (about 70 Kbytes), requires less than a minute to transfer at 9600 baud. By way of comparison, a low resolution copy of test chart No. 2 of CCITT Recommendation T.21 (mixed text and graphics) is about 55 Kbytes; or about 43 seconds of transfer time. A suite of pages was printed on several different PostScript printers of varying memory configurations and PostScript engines. The tests indicated that no page took less than 1.5 times the amount of time required for sending the decompressed data. Many pages took significantly longer than the raw transfer time; especially on the slower printers.

It was clear from this work that two factors were important in reducing the printing time: use the simplest straight-line PostScript code (no looping or conditional constructs) and reduce the data transferred on the serial line. The scheme that was selected is based on the observation that the facsimile data can be viewed as a series of horizontal *spans* of black pixels painted on a white background. (In fact, this is also the model on which the Group 3 compression algorithm works.) The spans are imaged using the PostScript line drawing operators. The coordinate system is setup so that a unit square corresponds to a single pixel of facsimile data. A page of facsimile data then becomes simply a series of PostScript *moveto*, *lineto*, and *stroke* operations with coordinates expressed in rows and pixels. Since all the lines are horizontal, most PostScript interpreters can image the operations very quickly. This satisfies the first of the two criteria for a fast printing method (straight-line code), but even with abbreviations for the PostScript operators a lot of data is required to express an image in this way.

To reduce the amount of data transferred, each page of facsimile data is preprocessed to build a compact encoding of the series of line drawing operations. Specifically, each span is treated as an <*rmoveto*, *rlineto*> pair. A histogram of all pairs that appear in a page is constructed and PostScript token names are assigned based on the frequency of their use. The token names are generated from the PostScript character set; avoiding those characters that either have special meaning, or that might result in a valid PostScript

† A standard ISO A4 page is guaranteed to have a reproducible area that is 281.46 mm long [CCI80b]. This translates to about 1083 lines at low resolution (3.85 line/mm) or 2166 at high resolution (7.7 line/mm).

‡ A forthcoming revision of the PostScript language includes an scheme for the efficient transmission of compressed image data [Cor90b].

operator name (e.g. *add*). The compressed image for each page then takes the form of a prologue to set up the page transformation and define encoding, followed by the encoded image data.

This technique works very well with text or mixed text and graphics but is poor for halftone images. Experiments indicate a printable page of text requires about 1.4-3.0 times as much data as the raw binary compressed Group 3 facsimile data. The time to print a page is entirely dominated by the time required to transfer the encoded PostScript. This is very encouraging as interfaces with higher transfer rates are becoming more commonly available.

A number of additional techniques can be applied to the problem to reduce and/or optimize this encoding technique. The algorithm assumes coherence in one direction. If the image is not oriented in the direction it is scanned, the coherence can be improved by rotating the image. Inverting the photometric interpretation of data (using white on black instead of black on white) can also improve the encoding. Preprocessing an image with a filter that removes some noise can significantly improve coherence, although it can also result in a loss of image detail. Finally, the encoding algorithm can be made more complex by considering strings of move-draw operations. The latter technique however is very difficult because the cost of specifying an encoded token requires that any improvements obtained from multi-pair encoding be significant. The time of additional processing intended to improve the encoding must also be weighed against the time to just transfer the data. Since the encoding process is typically done between the time a request is made to print a facsimile and the time it appears in a printer's output tray, reducing the amount of data through additional preprocessing may not reduce the overall time to print a facsimile.

6. Automatic Facsimile Routing

Existing and proposed facsimile standards define the protocols used for communicating data between FAX machines, but they do not specify any type of *envelope* or *addressing information* other than a phone number. Consequently, FAX systems that serve multiple users either use groups of phone numbers tied to a single phone line, Direct Inward Dialing (DID), or human intervention to automatically route incoming facsimile to a destination. DID requires special FAX modem hardware and requires multiple phone lines for an additional cost. Human intervention has the obvious downsides. FlexFAX uses a routing scheme based on a per-user *bar code* that is scanned from a cover page of each received facsimile. This scheme is independent of any hardware technology, but requires prior knowledge of the destination user's assigned bar code. This can easily be provided to a sender through an initial facsimile. It is expected that people will accumulate bar code symbols for specific people in much the same way that they collect business cards or addresses in an address book.

6.1. Bar Codes

Bar codes were introduced about 20 years ago as a means of labelling material goods. They are most commonly encountered in supermarkets and departments stores. The bar code industry uses the term *symbology* to denote each particular bar code scheme, while the term *symbol* refers to the bar code label itself [Pav90a]. There are many different symbologies, including UPC (Universal Product Code) [Sav75a] and Code 39 (three of nine), a scheme standardized by the United States Department of Defense since 1980 [Pal89a].

FlexFAX uses a Code 39-like bar code symbology for routing facsimile directly to users. The scheme used by FlexFAX differs from Code 39 mainly in the dimensions of the symbols. Code 39 was selected because it is widely known and has several properties that are important in recognizing symbols in facsimile:

- They are easy to recognize,
- They can be scanned in either direction,
- They are *self-checking* (i.e. single-bit errors can be immediately detected), and
- There is no specification for the arrangement of code words.

It should be noted that Code 39 is just one of many symbologies that would satisfy our needs.

6.2. FlexFAX Integration

FlexFAX users are automatically assigned a bar code symbol the first time they send a facsimile. These symbols are maintained in databases located at each server and typically are just the given name of the user (e.g. "Sam Leffler"). Name clashes are resolved by appending a unique digit to a clashing name. Alternative schemes, such as using the users' mail address ("sam@flake.asd.sgi.com") are possible with

Code 39; and would eliminate the need for an intermediate database to map user name to mail address. Mail addresses however tend to be longer than (shortened) user names and so require more effort to scan.

The bar code symbol for a user is normally included on cover sheets of outgoing facsimile sent by each user. These cover sheets are designed so that they may be turned upside down and used on any response.† It is also possible to place bar code symbols in the margin of pages. This scheme was rejected because the space needed to get a reliably scanned symbol tends to infringe on the working area of a page.

When a facsimile is received by FlexFAX it is passed to a scanning program that processes the document looking for a bar code symbol. If a symbol that maps to a known user is found, the document is converted to an ASCII format compatible with *uudecode* and dispatched to the user through the mail system. If no bar code is found, notification that an unrouted document exists is mailed to a distinguished user named the *FaxMaster*.

The bar code scanning algorithm sequentially searches documents looking for a bar code. Only the first page is typically scanned so that bar code symbols enclosed in a document do not give false results. This can be changed by a configuration parameter so that some number of pages are scanned. Unrouted facsimile may also be scanned in their entirety, although more commonly they are just viewed on-screen and manually dispatched.

7. Current and Future Work

The existing system would benefit greatly by direct support for PostScript imaging in sending facsimile. If a PostScript interpreter were available, all the input materials except images could be handled with a single imaging processor. Experiments using NeWS 1.1 as a PostScript interpreter have not been satisfactory. Public domain interpreters such as GhostScript do not appear to be a solution either. We are currently examining alternatives that would provide a good solution to this problem.

The use of bar code symbols for addressing appears to be a success. Some problems can arise in scanning bar codes on cover sheets that have not been copied onto plain paper. These problems are due to the ease with which the thermal paper normally used in FAX machines stretches during scanning, often causing distortion of the bar code symbols. The current bar code recognition algorithm handles scaling and rotation, but can be fooled by nonlinear warping. More sophisticated schemes that can handle nonlinear warping are possible, but incur more computation and may not be worthwhile. Alternatives such as re-orienting the bar code or using marking symbols other than bars to minimize the distortions are also a possibility.

Another issue associated with the bar codes is their content and related maintenance. It would be better to use a "White Pages" service to map names derived from bar codes to electronic mail addresses. This would avoid having to maintain databases and also avoid having to resolve name clashes (something that is not currently handled between FAX servers on separate machines.)

The integration between the mail system and the FAX service also needs to be improved. The current system uses a scheme based on Internet RFC1049 [Sir88a], but a more general scheme for handling multimedia documents would improve usability. As it is now, a user must explicitly view a facsimile enclosed in a mail message, rather than the mail system automatically recognizing the content of the mail message and presenting it in the most appropriate format. The mail reader for the NeXT machine is an example of a system that recognizes non-text enclosures and automatically presents them in their "native format."

Finally, there is a wide range of potential work associated with the CCITT Group 4 facsimile standards [CCI84a]. These standards are likely to be accompanied by the availability of greyscale and colour facsimile devices. The wide availability of Group 4 devices, however, appears to be a few years off due to their dependence on high speed communication facilities such as ISDN.

8. Current Status

FlexFAX is working, but still under development. It has yet to be made generally available to the computer users at Silicon Graphics, so it is too early to see if the original goal of replacing the FAX machines at Silicon Graphics will be attained. It has, however, demonstrated that it is possible to deliver a shared service that provides most all the services associated with a facsimile machine.

† The bar code scanning algorithm is designed to also recognize upside down symbols since this was found to be a common operational error.

Acknowledgements

Paul Haeberli designed the image conversion scheme. Mark Linton's feedback helped improve this paper. Eric Allman bought a FAX machine so that I could have a machine close by to work with (or so he said).

References

[Aba89a] Abaton, *InterFax Fax,* Abaton, Fremont, CA (1989). Facsimile system for Macintosh systems.

[CCI80b] CCITT, *Standardized Test Charts for Document Facsimile Transmission*, Recommendation T.21. 1980.

[CCI80a] CCITT, *Standardization of Group 3 Facsimile Apparatus for Document Transmission,* Recommendation T.4. Amended 1984. 1980.

[CCI84a] CCITT, *Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus*, Recommendation T.6. Amended 1988. 1984.

[Com89a] UniFax Communications, *UniFax,* UniFax Communications Inc., Vancouver, BC (1989). Facsimile gateway software for UNIX systems.

[Com90a] Apple Computer, *TrueType Font Manager*, 1990.

[Cor88a] Aldus Corp. and Microsoft Corp., *Tagged Image File Format, Revision 5.0,* Aldus Corp., Seattle, WA (August 8, 1988).

[Cor90b] Adobe Corp., *PostScript Language Reference Manual, 2nd Edition,* Addison Wesley, Reading, MA (1990).

[Cor89b] Silicon Graphics Corp., *4Dgifts Software*, Part of IRIX software distribution. 1989.

[Cor90a] Adobe Corp, *Adobe Type Manager*, 1990.

[Cor89a] Folio Corp, *Folio Font Manager*, 1989.

[Jey90a] Ross A. Jeynes, "Answers about fax fonts," *comp.lang.postcript*, A summary of an article by Adobe that appeared in the Fall 1989 issue of the Font & Function catalog. (March 1990).

[Ker84a] B. Kernighan, "Device Independent Troff," in *Documenters Workbench*, Bell Laboratories, Murray Hill, NJ (1984).

[Knu84a] Donald E. Knuth, "The TeXbook," ISBN 0-201-13447-0, Addison Wesley, Reading, MA (1984).

[Knu87a] Donald E. Knuth, "Digital Halftones by Dot Diffusion," *ACM Transactions on Graphics*, New York, NY **6**(4), pp. 245-273, Association for Computing Machinery (October 1987).

[Pal89a] R. C. Palmer, *The Bar Code Book,* Helmers Publishing, Peterborough, NH (1989).

[Pav90a] Theo Pavlidis, Jerome Swartz, and Ynjiun P. Wang, "Fundamentals of Bar Code Information Theory," *IEEE Computer*, pp. 74-86 (1990).

[Pos89a] Jef Poskanzer, *Extended Portable Bitmap Toolkit Distribution*, Included on MIT X11R4 software distribution. November 22, 1989.

[Res89a] PC Research, *FaxiX – Facsimile System for UNIX,* PC Research, Tinton Falls, NJ (1989).

[Sav75a] D. Savir and G. Laurer, "The Characteristics and Decodability of the Universal Product Code," *IBM Systems Journal* **14**, pp. 16-33 (1975).

[Sch86a] Robert W. Scheifler and Jim Gettys, "The X Window System," *ACM Transactions on Graphics*, New York, NY **5**(2), pp. 79-109, Association for Computing Machinery (April 1986).

[Sir88a] M. Sirbu, "A Content-Type Header Field for Internet Messages," RFC-1049, Network Information Center (March 1988).

[Sol90a] Solutions, *FAXGATE Plus,* FaxGATE Widget Company, Williston, VT (1990). Facsimile gateway software for Macintosh systems.

[Tec89a] Cygnet Technologies, *The Spirit of TIFF Class F*, December 1989.

[Uli87a] Robert Ulichney, *Digital Halftoning,* The MIT Press, Cambridge, MA (1987). ISBN 0-262-21009-6

The more I find out, the less I know?

NFS Fileserver Benchmarks (including one that works)

William Roberts

Department of Computer Science
QMW, University of London
Mile End Road
LONDON E1 4NS
England
liam@cs.qmw.ac.uk

ABSTRACT

The problems of benchmarking are many, and the problems of benchmarking a distributed system such as the NFS filesystem are more numerous still. This paper examines three existing benchmark programs, one trivial and two serious. These programs will be seen to fall into various pitfalls, but one of them manages to produce a single meaningful measure of NFS fileserver performance. The two serious benchmarks are examined in some detail and results of tests are given. The paper closes with some warnings about the successful benchmark, some thoughts on vendor support for benchmarks, and the relationship between benchmarks and the Uncertainty Principle.

1. A Simple Disk Benchmark

The first benchmark we examine in this paper is a disk read/write benchmark called diskperf, originally written by Rick Spanbauer (rick@sbcs.UUCP) to test the relative performance of various Atari and Amiga hard disks. It performs a number of file read and file write experiments using a range of different sizes for the requests (512 bytes through to 32K bytes) and measures the elapsed time to determine bytes per second throughput. Some sample output is shown in Figure 1.

Distributed as comments in the source code (posted to comp.unix.aux some time before December 1988) were results from a number of larger machines including Sun 3/50 and Vax 11/780. We (and others) used the diskperf benchmark to berate Apple over the performance of their disks under A/UX 1.0, and eventually demonstrated that their default mkfs parameters were slowing disk access by a factor of about 2.

On the slim grounds that an NFS fileserver is in some sense pretending to be a disk, we were going to try this benchmark on NFS file servers as well. The test was finally abandoned when a with-it Sun technical support person observed that the "local disk performance" of the SPARCStation1 exceeded the bandwidth of the SCSI bus connecting the disk to the CPU, again by a factor of 2!

In fact, the caching strategy of SunOS 4.0 means that the high activity of the files used in the test causes them to remain almost entirely in memory, scarcely touching the disk at all. This kind of caching is not used in micro computers such as the Atari or the Amiga (which use small write-through caches) and so the benchmark is making no attempt at all to defeat or allow for this kind of cheating.

```
File create/delete:   create 18 files/sec, delete 60 files/sec
Directory scan:       2532 entries/sec
Seek/read test:       4262 seek/reads per second
r/w speed:            buf 512 bytes, rd 1883669 byte/sec, wr 159925 byte/sec
r/w speed:            buf 4096 bytes, rd 3495253 byte/sec, wr 191229 byte/sec
r/w speed:            buf 8192 bytes, rd 4695116 byte/sec, wr 2419790 byte/sec
r/w speed:            buf 32768 bytes, rd 4839581 byte/sec, wr 2476951 byte/sec
```

Figure 1: *The output of the diskperf benchmark for a SPARCStation1*

2. What is cheating?

Cheating is anything which improves the benchmark result but which isn't related to the things supposedly under test. The SunOS 4.0 caching is a good thing for speeding up compilations, but it is "cheating" in the context of the simple disk benchmark because it improves the performance figures in a way which is independent of the disk under test. If you don't like the pejorative word "cheating", use "obfuscating" instead: either way it results in benchmarks that don't measure what they claim to be measuring.

The quintessential benchmarking problem is not how to get some numbers by running a few programs, but what those numbers mean. Beyond the simple issue of what an individual result means, there is the additional problem of how two runs of the benchmark should be compared. The benchmarking myth which must be believed if the whole activity is to be worthwhile is:

There exists such a thing as a repeatable load.

Single system benchmarks are intended for either system tuning or system comparison: they usually incorporate a mixture of "typical activities" and aim to measure "overall system performance". All computer magazines have their own favourite collection of tests (often including things like "mail merge and print 200 letters"), and the recent SPEC benchmarks are a noble attempt in this vein [SPE89a] aimed at typical tasks for a scientific workstation. By measuring "overall system performance", the question of cheating usually doesn't arise and so a caching strategy which reduces disk access is a fair technique to improve the benchmark result. Conversely, there is usually little way of knowing what use such benchmarks are when comparing machines if you don't do the standard workload.

Other benchmarks, for example Whetstones and Drystones, concentrate on a typical program structure and remove minor components such as disk access etc, leaving a benchmark which tests just a single aspect of the CPU: Whetstones test floating point performance and Dhrystones test integer performance. Dhrystones illustrate the problem with this approach in that the Dhrystone result depends strongly on the quality of the C compiler and should only vary slightly between similar hardware designs: indeed the most reliable use of Dhrystone measurements is to compare different C compilers for the same machine, rather than comparing machines.

2.1. Issues when benchmarking NFS servers

In a multi-machine system, the problems become much worse. Looking at NFS in particular, a benchmark might be measuring CPU performance in client or server, disk performance in the server, network performance, effectiveness of the caching strategy in the client or server, or any combination of these things. A "typical load" benchmark run on a variety of client machines may produce varying amounts of load on the server depending on purely client-related issues, for instance the amount of memory in the client.

The complexity of what can happen when the read system call is used to read data from a file stored on an NFS file server is illustrated in Figure 3: The point to note is that simple benchmark programs may indeed make some fixed number of read system calls, but that this doesn't necessarily result in the same number of NFS read requests sent to the server. The same is true of write system calls, with the important difference that writes which actually go to the server will be written to disk immediately rather than just written into the server cache (i.e. for NFS requests the server cache is considered write-through rather than write-behind).

In the light of preceding discussion, we need to decide what is and isn't cheating when we are trying to benchmark NFS file server performance. Our answer is

The result should depend on the server, not the client.

Anything the server does is allowed.

There are some properties of a correctly implemented NFS file server which aren't very clearly stated in the definition of the NFS protocol, for example the "synchronous write" property mentioned above. If we take for granted that the server obeys all the rules, both written and unwritten, then anything else which the implementors can do to improve the server performance is fair and should improve the benchmark rating of their server.

The next two sections each examine a respectable NFS server benchmark: the NFSSTONE benchmark produced by Encore Systems [She89a], and the NHFSSTONE benchmark produced by Legato Systems Inc [Inc89a]. These names are too similar, so for the rest of this paper we will refer to the Encore benchmark and the Legato benchmark respectively.

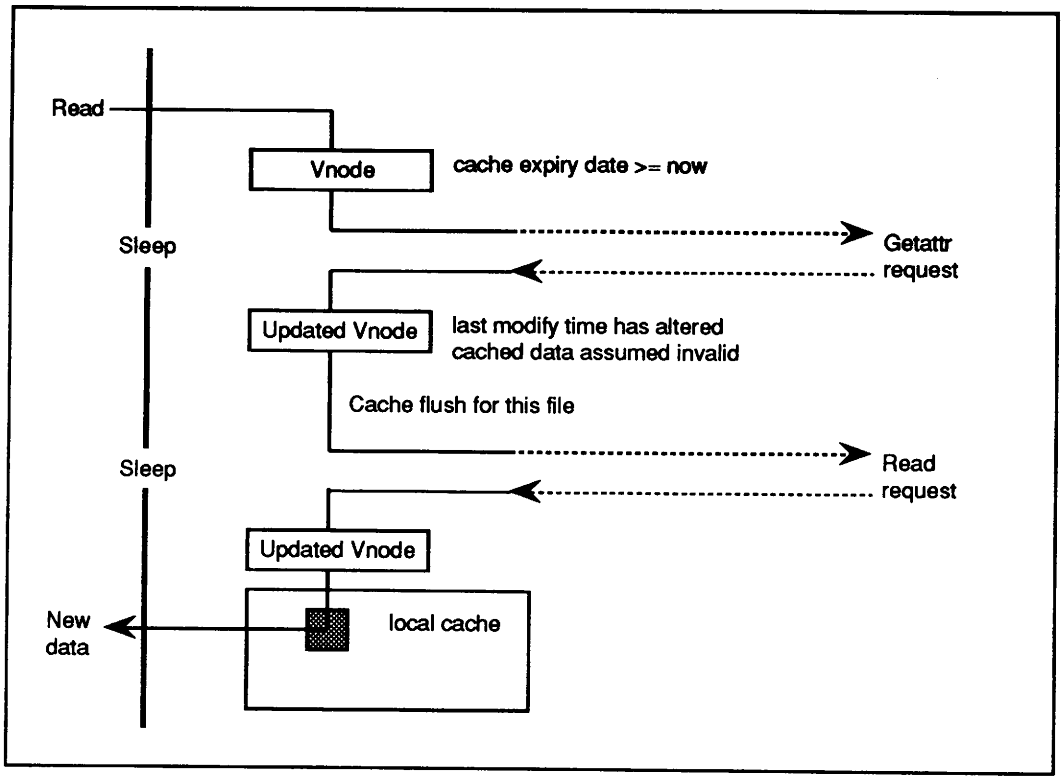


Figure 3: *A complex example of NFS activity triggered by a read system call*

3. The Encore NFSSTONE benchmark

This benchmark is described in [She89a] and was written by Barry Shein, Mike Callahan and Paul Woodbury. We obtained our copy of the source code and the paper from Paul Bixby at Encore (bixby@pinnochio.encore.com).

The Encore benchmark produces a fixed sequence of operations with each run of the benchmark forking 6 child processes to generate load in parallel on separate subdirectories. The load generation consists of a mixture of file creation and deletion, some directory scanning, and routines called seq_write, seq_read and nseq_read which create substantial files and then read them back; the nseq_read routine reads the file in a non-sequential fashion intended to overcome the normal UNIX read ahead [Bac86a] and to simulate non-sequential file access in general. A simple mechanism is used to achieve approximate synchronisation between several clients running copies of the benchmark, making it easy to generate a larger simulated load by using several client machines.

The result of the benchmark is the number of operations generated, divided by the elapsed time for the run (including all of the child processes terminating and the parent removing the directories afterwards). The "number of operations generated" is computed from the repetitions of each particular routine, for example, the seq_read() routine performs BLOCKS_PER_FILE read requests each for BYTES_PER_BLOCK bytes and so contributes BLOCKS_PER_FILE operations to the overall total (these values are #defined and so fixed at compile time). The default settings were chosen to produce NFS requests in a proportion similar to those described by [San85a] using a Sun 3/60 client and an Encore server with multiple CPUs (they don't say which model).

The benchmark is intended to be useful in comparing different server configurations and even different remote file system implementations. This means that Encore deliberately avoid trying to measure lower level protocol performance or examining the NFS statistics recorded by the client and server kernels.

The design of the benchmark is intended to defeat various aspects of client caching. In particular the total size of the files generated during a single instance of the benchmark is over 12 Megabytes, and the code contains comments warning that on SunOS 4.0 systems this may need to be increased to ensure that the client cannot hold all the files in memory.

Client	Results				Comments
Sun 4/60	173.5	176.4			
Sun 3/60	96.2	96.8	96.5	98.9	
Sun 3/60	97.3	97.0			Cache tweaking
Sun 3/50	61.8	61.6	59.1	58.4	

Table 1: *Raw Encore results for a Sun 4/260 server*

Operation	Target%	Sun 4/60		Sun 3/60		Sun 3/50	
lookup	53.0	77	77	54	54	53	53
read	32.0	2	1	32	32	32	32
readlink	7.5	10	10	7	7	7	7
getattr	2.3	3	3	1	1	2	2
write	3.2	4	4	3	3	3	3
create	1.4	1	1	1	1	1	1
Rating		173.5	176.4	96.5	98.9	59.1	58.4

Table 2: *Nfsstat percentages for different clients against the same server (Target figures taken from* [She89a], *following* [San85a])

3.1. The Encore benchmark in practice

To try out the Encore benchmark, we ran several separate trials using a variety of client machines and a single fixed server. The machines involved were:

greyeye	The server, a Sun 4/260 with 327 Megabyte SCSI disk and 32 Megabytes of memory, running SunOS 4.0.3
io	A SPARCStation1 (Sun 4/60) with 16 Megabytes of memory, running SunOS 4.0.3
titan	A Sun 3/60 with 4 Megabytes of memory running SunOS 4.0
zarniwoop	A Sun 3/50 with 4 Megabytes of memory running SunOS 3.5

Each experiment consisted of running the Encore benchmark several times with a single client. The basic results were as shown in Table 1.

The "cache tweaking" experiments with the Sun 3/60 client involved changing the mount command options acregmin and acregmax (to 60 and 600 respectively): this extends the length of time that cached data is considered valid and reduces the frequency of checks made before using such cached data (see Figure 3 above). As can be seen, the variability of the results is such that the difference between the Sun 3/60 results doesn't seem significant, so the Encore benchmark is probably doing fairly well at overcoming the caching on this machine.

However, when we arrange to look at the client NFS statistics on the client workstations by executing the command "nfsstat -csz" before and after each run of the Encore benchmark, we see the mixture of operations shown in Table 2.

From these figures is it clear that the Sun 4/260 server is being asked to do different things when the Encore benchmark is run on the Sun 4/60: in particular it does more lookup operations and fewer reads when compared to the Sun 3/60 and Sun 3/50 experiments.

Statistician: Excuse me, but what you have just said is wrong.

Author: It can't be wrong, I spent a lot of evenings obtaining those measurements.

Statistician: You are comparing percentages, i.e. relative proportions of some total, without making sure that the totals are actually the same.

Author: So I need yet more figures?

Statistician: No, just ignore the percentages and look at the actual numbers: your experiment was supposed to be a repeatable load so the numbers should be the same each time, shouldn't they?

It is very easy to make this kind of mistake when trying to interpret benchmark results: it is also easy to make such mistakes with designing the benchmark experiments in the first place. Table 3 shows the NFS operation counts and the theoretical expectations (based on the system calls made by the benchmark).

Operation	Syscalls	Sun 4/60		Sun 3/60		Sun 3/50	
getattr	0	1112	1103	573	575	1097	1099
read	15000	695	543	14774	14811	15027	15026
lookup	25002	25058	25060	25058	25059	25086	25087
readlink	3498	3498	3498	3498	3498	3498	3498
write	1500	1500	1500	1500	1500	1506	1504
create	504	504	504	504	504	504	504
remove	18	18	18	18	18	18	18
rename	6	6	6	6	6	6	6
symlink	6	6	6	6	6	6	6
mkdir	8	8	8	8	8	8	8
rmdir	8	8	8	8	8	8	8
Total	45522	32416	32413	45953	45993	46764	46764
Rating		173.5	176.4	96.5	98.9	59.1	58.4

Table 3: *Nfsstat numbers for the Encore results in Table 2*

This analysis shows why comparing percentages is wrong: far from doing "more lookups and fewer reads", the Sun 4/60 is simply doing far fewer reads and so making significantly fewer calls to the server. To be fair, comments in the source code for the benchmark did warn that a SunOS 4.x machine with more than 12 Megabytes of memory would be able to cheat by holding the files in its local cache, but the benchmark program itself doesn't detect this problem at run-time.

3.2. Why such a variation in results?

The numbers are fairly close for repeated runs of the same benchmark on the same machine, so the "repeatable load" is indeed repeatable. However it is clearly not a repeatable load on the server alone, rather a repeatable load on the client/server combined system.

In the case of the Sun 4/60, its large local memory subverted the benchmark read attempts by holding the information in the local cache; it made far fewer read requests and a few extra getattr requests to check cache validity.

The Sun 3/60 and the Sun 3/50 machines both have the same amount of memory, so why does one give only 60% of the performance of the other? To identify the problem we have to delve even deeper into the NFS implementation on the client and examine more of the nfsstat figures, shown in Table 4.

The number of NFS calls reflects only the number of request/reply exchanges between the client and server. At a lower level in the protocol, the RPC mechanism will retransmit requests if they are not answered within a timeout period. If the server was simply busy the response to the original request was just delayed and the server may well repeat the work to answer the duplicate request. The first answer with the correct request identifier (XID) will be accepted, and any answers with old XIDs will be ignored.

The top part of Table 4 shows the number of actual RPC calls (there may be more than one for some NFS calls), the number of retransmissions (retrans), the number of replies received that had out-of-date XIDs (badxid), and the number of timeouts. It is clear that the Sun 3/50 is having rather more timeouts than the other clients and the number of badxids in those experiments show that the problem was not due to network traffic being lost.

The most striking difference is the nclsleep figure which is over 10% of the total number of calls in the Sun 3/50 case. It is a little known fact that SunOS reserves a fixed number of "client structures" in the kernel

Statistic	Sun 4/60		Sun 3/60		Sun 3/50	
rpc calls	32416	32257	45961	45997	46775	46767
retrans	27	14	19	8	641	613
badxid	0	0	1	0	541	534
timeout	27	14	21	9	631	604
nfs calls	32416	32413	45953	45993	46764	46764
nclsleep	0	0	0	0	5089	5072
Rating	173.5	176.4	96.5	98.9	59.1	58.4

Table 4: *More nfsstat details from Encore benchmark experiments*

which are used to handle outstanding NFS requests. To send an NFS request to a server, the process making the request (or a biod in the case of readahead or delayed writes) must first obtain one of these client structures: if none is available then the process increments nclsleep and then sleeps until one becomes free. The nclsleep figure shows that the Sun 3/50 is actually being limited by having too few client structures declared in its kernel, which is manifestly not a property of the file server under test.†

3.3. Encore benchmark conclusions

The Encore benchmark is testing the client/server combination. Results obtained with different clients on the same server are not directly comparable because so much could be going wrong and the benchmark itself doesn't warn you.

The benchmark isn't completely useless; the following comment comes from Peter Bixby of Encore (bixby@pinnochio.encore.com) [Bix90a]:

> While I agree that you are in fact measuring the client, that should only be the case as long as the aggregate client load is small. The real test is to increase the number of clients until the server is truly being stressed. Then continue to increase the load measuring the responsiveness of the server vs. the increasing load. Any server (that's truly a server) should be able to handle a small load with reasonably good response time.
>
> What the benchmark is really intended for is to allow an end user to generate a reasonably repeatable load that can a) be compared to their current or planned usage, and then b) apply that load to a variety of server platforms.
>
> The customer having done a bit of homework can characterize his/her current or planned load, and then take that information to a number of vendors along with some standard piece of (hopefully portable) code and let the server supplier run the benchmark and show response time vs load curves. Initially, it had even been our hope that a benchmark (any fair benchmark – not necessarily ours) would become so standard that vendors would publish the numbers (we obviously felt that we would do very well on any fair benchmark).

We would add the further caveats that the customer must also specify the client machines to be used for the test, that results for different clients are not directly comparable, and that mixed client tests should not be used.

4. The Legato NHFSSTONE Benchmark

Legato Systems Inc. have produced a benchmark called NHFSSTONE (with a silent H) and made the source code publicly available. The benchmark is aimed solely at evaluating servers based on Sun NFS and expects to be able to read the same kernel data structures that are displayed by the nfsstat command.

In some respects, the Legato benchmark has a similar structure to the Encore benchmark: it consists of a program which forks some number of worker processes which generate the load on the fileserver. The difference is that the workers in the Legato benchmark are reading the nfsstat values from the client kernel and executing small load generating routines in an attempt to produce a specified mix of operations: if there appear to be too few read requests (as for example happened in the Encore benchmark on the Sun 4/60) then the workers will perform the read-generating routine more often to try to compensate.

Runs of the benchmark are parameterised by the desired mix of NFS operations, the number of worker processes to run, the desired load on the server (in calls per second), and either the desired number of calls or the number of seconds to run for.

Instead of using just one file per worker process, the Legato benchmark uses a much larger number (typically around 40) per worker, and cycles through them when performing particular operations. Its vocabulary of NFS request generating routines includes routines which use the fstat system call to generate getattr requests, and one using the statfs system call to generate fsstat requests: its coverage is therefore wider than that of the Encore benchmark.

The Legato benchmark can be provided with a description of the mix of NFS requests to generate: rather neatly, the description takes the form of the output of nfsstat, so the easiest way to generate the description appropriate to your site is to run "nfsstat -sz" on an existing server before and after a typical peak period.

† It also shows that (where biods are concerned) 4 is not a good number or even a bad number, just an irrelevant one....

```
% nhfsstone -l 1000 -c 20000 -p 10 -v
op          want       got     calls       secs  msec/call    time %
null          0%     0.00%         0       0.00       0.00     0.00%
getattr      13%    13.49%      2712     282.87     104.30     7.97%
setattr       1%     1.12%       227      45.51     200.52     1.28%
root          0%     0.00%         0       0.00       0.00     0.00%
lookup       34%    32.69%      6571     778.57     118.48    21.94%
readlink      8%     6.99%      1405     242.97     172.93     6.84%
read         22%    23.28%      4681     786.79     168.08    22.17%
wrcache       0%     0.00%         0       0.00       0.00     0.00%
write        15%    15.34%      3085    1020.19     330.69    28.75%
create        2%     2.03%       409     176.89     432.51     4.98%
remove        1%     1.01%       203      87.49     431.02     2.46%
rename        0%     0.00%         2       0.00       0.00     0.00%
link          0%     0.00%         0       0.00       0.00     0.00%
symlink       0%     0.00%         0       0.00       0.00     0.00%
mkdir         0%     0.00%         0       0.00       0.00     0.00%
rmdir         0%     0.00%         0       0.00       0.00     0.00%
readdir       3%     2.99%       602      94.09     156.31     2.65%
fsstat        1%     1.00%       202      32.67     161.77     0.92%
357 sec 20099 calls 56.29 calls/sec 176.53 msec/call
```

Figure 4: *Output from the default mix Legato benchmark (Sun 3/60 client, Mac IIfx server)*

The output of the Legato benchmark has two forms, of which Figure 4 shows the verbose form. It shows the desired mix of operations (as a percentage of the total), the actual number of operations and the actual percentages, and the average number of milliseconds elapsed time per system call.

Alas, the milliseconds per system call figure is valueless as a part of a pure server benchmark. It is obtained by performing a gettimeofday system call before and after the system call thought to generate the NFS request in question, but as Figure 3 above and the Encore analysis showed, system calls and NFS requests are only loosely related because of the decoupling effect of the local cache (not to mention trivia like context switching to other worker processes!). This part of the Legato measurement is like the Encore benchmark in that it measures combined client/server performance, but an attempt is made to associate these timings with individual NFS requests rather than reporting simply average times for particular system calls. We will ignore the milliseconds per call results in subsequent discussion.

4.1. Strengths of the Legato benchmark

Despite the strange attempt to measure the elapsed time for an NFS request just by measuring elapsed time of a system call, the Legato benchmark does have a number of important strengths.

It knows if it has achieved the correct mix of operations. The Encore benchmark failed essentially because it hoped for a one-to-one correspondence between system calls and NFS requests. The Legato benchmark repeatedly examines the kernel nfsstat statistics to find out what NFS requests have actually been achieved, and adjusts itself accordingly. If the final results do not match the desired mix closely enough, the benchmark declares the run invalid.

It checks that the client is idle before starting. Prior to letting the workers start, the benchmark waits for a few seconds to check that the test machine is not making any NFS requests other than those generated by the workers; this ensures that the NFS requests shown by the kernel statistics are all directed at the server under test. If there is other NFS activity then the test is abandoned immediately. The test can be run from a diskless client provided that the benchmark program remains in memory during the test: on an otherwise idle machine this is likely to be true since the benchmark code is very active, but executing the benchmark code from the fileserver under test is an alternative way of ensuring that any paging requests are part of the load on the server under test.

It uses real time and real statistics. By using the kernel nfsstat statistics the benchmark is tied to the actual effect of its actions rather than the intended effects. This allows it to compensate for large caches in the client and similar effects. If it becomes impossible to compensate then the benchmark detects this and declares the whole run invalid. If it is used to achieve a modest load on the server, the worker processes may choose to sleep briefly: this is done in time intervals which are multiples of 20 milliseconds (so no real-time clock is required) and uses select to achieve these short delays with low overheads.

4.2. Weaknesses of the Legato Benchmark

The Legato benchmark is not without its faults, most major of which is the way it computes the results. Each worker process is continually checking the kernel statistics and will stop after it detects that the desired number of calls have been generated, reporting its total elapsed time to the parent. The parent uses the average of these reported times, but its own independent measurement of the number of calls achieved. This is very dubious and may contribute some of the variability to the results.

The load generation operations are of a rather variable granularity. Most of the load generation routines will attempt to produce just one NFS request, for example a single statfs system call is used to generate a single fsstat request. The read and write operations, on the other hand, process a complete file which may be up to 256K in size and so generate over 30 requests. The write operation can also increase the size of a file, so this effect gets worse as the length of the test increases.

The load generating routines are labouring directly against the normal caching on the client workstation, since the Legato benchmark aim is to cause lots of NFS activity whereas the caching strategy is trying to eliminate NFS requests where possible. At present the benchmark doesn't report the number of system calls it used to generate the requests: this information could be used to indicate whether or not the results might be higher if the local cache size were reduced.

The validity of the experiment is determined by adding the differences between desired percentage and achieved percentage of operations. This doesn't give any weighting to different operations so that a 1% deviation in the number of fsstat calls (which can typically be processed at a rate of over 250 per second) is considered the same as all the others even though a reasonable mixture of operations gives results of less than 100 calls per second. A simple sum of differences in percentages doesn't seem like the proper way to measure the deviation from the desired experiment, but traditional statistical measures such as Chi-squared tests aren't very appropriate if you can get spontaneous events which you didn't really want (e.g. the occasional small numbers of getattr calls observed during tests which call for 100% fsstat operations).

4.3. The Legato benchmark in practice

It is clear that the results of a benchmark reporting "NFS calls per second" will only be comparable if the same mixture of operations is used, so all Legato results need to be qualified by reporting the mix of operations. To provide a direct comparison with the Encore results given above it is therefore necessary to use the appropriate mix of operations: for this we take the output of "nfsstat -c" from the Sun 3/50 experiments shown above (these correspond well to the original figures from [San85a] which form the basis of the Encore mix of operations). The results of the Legato benchmarks are shown in Table 5, obtained by asking for a load of 1000 calls per second (i.e. push the server as fast as you can) and running the experiment for 46000 calls with 6 worker processes. The table also shows corresponding figures from the Encore experiments described above.

The Legato benchmark does not currently work on SPARC machines, so the experiments were only carried out on the Sun 3/60 and Sun 3/50. *(Not very convincing then.)*

The figures for the RPC calls etc of the Legato experiments are obtained in the same way as for the previous Encore experiment: the benchmark program does not provide them so we run "nfsstat -csz" immediately before and after the benchmark run. In the Legato case however, the benchmark prepares a set of files before starting its measurements, so the nfsstat figures include this extra activity as well as the load that the benchmark itself generates. The Legato benchmark does display the load achieved during its intended measurement period and this matches accurately the mix of operations that we specified and the

Value	Sun 3/60			Sun 3/50		
	Legato		Encore	Legato		Encore
benchmark	86.5	87.2	96.5	82.8	83.8	59.1
nfs calls	46044	46075	45953	46035	46056	46764
seconds	532	528	471	556	550	770
rpc calls	47641	47651	45961	47217	47254	46775
retrans	11	7	19	124	141	641
badxid	0	0	1	112	125	541
timeout	11	7	21	124	140	631
nclsleep	0	0	0	5116	4499	5089

Table 5: *Results of Legato benchmark with the Encore operation mix*

desired number of operations. The Legato benchmark also provides the timing for its measurement. This means that in Table 5, the figures in rows "rpc calls" to "nclsleep" given for the Legato experiments are actually for a longer period of activity than the benchmark itself.

Hoping fervently that the detailed figures of Table 5 are still useful given the difficulty just described, we are now supposed to be comparing very similar loads on the server. The mix of operations is the same, the total number of operations is the same to within 0.25%, there are 6 load generating child processes in each case. The Legato measurements are very similar though the Sun 3/50 results are lower at about 95% of the Sun 3/60 results: the Encore results differ much more with the Sun 3/50 result being about 60% of the Sun 3/60 results.

In the earlier analysis of the Encore benchmark results, the Encore discrepancy was attributed to the extra nclsleeps, but the Legato Sun 3/50 experiments show a comparable number of nclsleeps. Unless these nclsleeps are all incurred before the Legato benchmark starts its own internal measurements (not that it measures nclsleeps anyway) then the 20% increase in the time taken to perform the Encore benchmark on the Sun 3/50 compared with the Legato benchmark must (might perhaps?) be related to the increase in timeouts and retransmissions.†

Despite the difficulty in explaining why the Encore benchmark gives such different results for these two clients, it is clear that the Legato benchmark is making a much better job of producing a repeatable load on the server and producing a measurement which is independent of the client.

4.4. Other Legato Benchmark Experiments

Given a parameterisable benchmark, it is possible to vary the experiment in interesting ways. Legato Systems Inc have used the benchmark to demonstrate that the most expensive NFS operation is write [San89a], and produced a non-volatile disk cache controller for Sun equipment that maintains the "stable storage" property of synchronous writing on the server, without actually forcing all of the altered data to disk immediately. This regains the speedups of the UNIX disk cache for the NFS server without losing the benefit of NFS clients surviving server crashes, so legitimately speeding up NFS writes.

At QMW we have use the operation mix that is the Legato benchmark default to compare a variety of possible purchases for central student fileservers. These servers are supporting a total of 100 "dataless" workstations: all personal files are stored on the NFS servers, but most operating system files, all swap space, and a reasonable amount of temporary file space are provided on local workstation disks. The default Legato mix is more appropriate in this case than the Encore mix because it has a much larger proportion of write operations.

This paper was written as a result of this search for good, cheap fileservers and so the results presented reflect a range of different experimental practices: Most experiments have been attempted with the Sun 3/60 as the client and adjustments indicated by the comments in Table 6.

It is worth noting that a server producing more than 50 calls per second under this mix usually produces runs with an incorrect mix on a SunOS 4.0 client, with mix wrong by about 15% and too few getattr calls. Longer tests make this even worse.

4.5. The Legato benchmark for testing server CPU only

An important question when comparing fileservers is the choice of disk system, particularly when the models available from the vendor for benchmarking purposes may not be the configuration you would actually buy. The statfs system call produces the infonnation shown by the "df" utility, namely blocks used and blocks free, which is actually taken from the filesystem superblock held permanently in the server memory. By using a mix which consists solely of NFS fsstat operation which implements the statfs system call, we can test only the server CPU and Ethernet performance independent of the server disks, producing the results shown in Table 7.

The pure fsstat measurement ensures that two server CPUs are broadly comparable, allowing a more accurate assessment of the virtues of particular disk systems or other server configuration issues. For example, Table 7 demonstrates that the use of SMD disks rather than SCSI disks on a Sun 3/160 makes an improvement of over 50% in the Legato benchmark rating using the standard mix.

† The Encore benchmark workers all start in unison and perform identical operations on different files. This includes all writing a large file simultaneously. Perhaps this sustained burst of writing causes an unusually large number of long timeouts: the Legato benchmark avoids this by producing an even mixture of operations throughout its load generation.

Server	Disk	Result(s)					Notes and Comments	
Sun 4/280	SMD 893M	96.2	98.4	98.3	99.3	99.7	TI	32 Meg RAM
Epoch-1	SCSI 690M	75.9	74.2				T	2 x 68020 dedicated server
Epoch-1	SCSI 690M	70.1						2 x 68020 dedicated server
Sun 4/330	SCSI ?	57.5	57.6	57.1	55.6	55.8	Y	
Balance 21000	SMD Eagle	61.1	61.6				TI	6 x 32332
Sun 4/20	SCSI 327M	70.2	69.5	68.8	67.0	69.4	TX	
Sun 4/60	SCSI 100M	43.0	42.3	42.0				
Sun 4/65	SCSI 1200M	89.0	88.8	87.8	88.2	85.7	TX	8 Meg RAM
Sun 4/65	SCSI 327M	66.8	67.1	68.0	70.1	70.4	TX	8 Meg RAM
Sun 4/65	SCSI 100M	44.4	44.2	42.8	41.4		TX	8 Meg RAM
Mac IIfx	SCSI 1200M	57.1	56.3	55.0	55.9		TX	A/UX 2.0b15
Sun 4/260	SCSI 327M	38.2	38.4	35.1			Z	32 Meg RAM
Sun 4/260	SCSI 327M	34.2	35.3					32 Meg RAM
Sun 3/160	SMD ?	37.7	39.0				T	SunOS 3.2
Mac IIcx	SCSI 600M	29.7	29.7	29.3			T	A/UX 2.0b10
HP 9000-300	SCSI ?	28.8	27.9					
Sun 3/160	SCSI 141M	27.4	26.9					
Sun 3/50	SCSI 141M	27.0	28.1	27.1				
Mac IIfx	SCSI 80M	23.0	21.6	20.4	22.4			A/UX 2.0b3
Mac IIcx	SCSI 80M	20.5	21.0	20.9				A/UX 2.0b3
Mac IIcx	SCSI 80M	7.4	6.8	7.4			F	A/UX 1.1.1
Mac IIcx	SCSI 80M	9.9	9.4				F	A/UX 1.1.1, DMA accelerator
WCW MG1	SCSI 250M	10.6	10.7					

T	= 10 worker processes rather than 6
I	= Invalid run due to too few getattr operations
F	= System V filestore with 1K blocks
X	= Sun 3/60 client running SunOS 3.5 rather than SunOS 4.0
Y	= Sun 3/80 client rather than Sun 3/60
Z	= Sun 3/50 client rather than Sun 3/60

Table 6: *Will these benchmarks stretch out to the crack of doom [Sha02a]?*

4.6. Legato benchmark conclusions

The Legato benchmark is a useful tool for comparing NFS fileservers. It produces comparable results despite reasonably drastic changes of client workstation and it can be used in a variety of ways to measure different aspects of server performance.

The way in which the Legato benchmark checks to see if its assumptions have been violated is invaluable: all benchmark suites that make assumptions should have something similar to filter out invalid results.

The Legato benchmark could be improved in a number of ways, giving it more range and removing the nonsense figures that it currently generates. The way in which it decides the duration of the benchmark is currently suspect and should be changed. It should get the additional statistics available from nfsstat and

Server	Result(s)			Std Mix	Comment
Mac IIfx	296.4	296.6	295.4	62.0	SCSI Wren7
Sun 4/65	273.8	273.4	273.3	89.0	SCSI Wren7
Sun 4/20	267.4	273.1		69.4	SCSI 327M
Sun 4/280	280.4	282.4	281.3	99.6	SMD disk
Sun 4/260	273.3	264.8		38.4	SCSI disk
Mac IIcx	257.2	258.1	243.8	29.7	
Sun 3/160	241.5	239.5	239.8	39.0	SMD disk
Sun 3/160	255.7	254.5		27.4	SCSI disk
Epoch-1	233.5	233.1	235.1	75.9	2 x 68020
Balance 21000	224.7			61.6	6 x 32032
Sun 3/50	218.2			28.1	

Table 7: *Pure fsstat mix with the Legato Benchmark*

include them in its assumption checking so that it can indicate when the client is running as fast as it can. It would also be useful to compare the local system calls made with the NFS requests actually generated in order to detect significant local caching.

A version of the benchmark capable of coordinating load generation from several clients (as per the Encore benchmark) would be valuable but should only be undertaken after some very careful thinking about how to establish a combined measurement.

5. Late News

Time and space do not permit a fuller expansion and justification of the following very recent results and observations using the Legato benchmark: perhaps more details at the actual conference presentation...

1. As the operating system on the client, SunOS 3.x gets higher results for the standard mix of the Legato benchmark than SunOS 4.x, presumably because it has a smaller cache and so more local system calls result in NFS requests.
2. For the pure fsstat mix, SunOS 4.x is the client operating system which gets better results, presumably because it can support more concurrent outstanding NFS requests.
3. Increasing the memory on the Ethernet card (used by the interface chip to store incoming packets) from 16K to 64K improved the Legato standard mix result from 35 to 60 calls per second.
4. Sun's Little Computer (Sun 4/20, aka the SPARCStation SLC) with a 1.2 GByte Wren 7 SCSI disk provides twice the performance of a Sun 4/330 with a 327 MByte disk, both tested with the standard Legato mix.

For NFS servers and the Legato mix, the disk performance is a critical factor. A lot of vendors write "server" on the biggest boxes without any real justification in disk performance: our results suggest that a CPU sold as a cheap workstation, plus a decent disk, usually has better price/performance and often better overall performance.

6. Conclusions

In many ways it is impossible to reach a real conclusion from benchmarking: the closer you look at the whole notion the less well-founded it seems (perhaps this is just cynicism borne of too many late nights slaving over a hot NFS file server). There are some clear positive recommendations which can be made, after which we will return to the philosophy.

6.1. Vendor support for benchmarking

One difficulty with benchmarking complex systems is the poor documentation for the additional tools used to cross check the benchmark. Investigating the true activity of the Encore benchmark required the use of the "nfsstat" program which provides a variety of different statistics. The usual nfsstat manual pages and supporting documentation don't explain the meaning of these numbers; they don't even explain the circumstances under which some of them are incremented! What *exact* circumstances lead to the "badcall" counter being incremented? The information given in this paper about the "nclsleep" figure is not found in any of the standard Sun documentation and in fact comes from the author's reading of the NFS 2.0 and NFS 3.0 kernel source code, backed up by occasional forays into the kernel with adb.

Calling all vendors: please give us detailed documentation on this kind of diagnostic program.

Accumulating timeouts and badxids by request type rather than simply as a total would make those particular numbers more useful. It would help in tuning an NFS client to generate as many calls as possible if the number of client structures in the kernel was binary configurable.

Another area in which most UNIX systems are lacking is provision for proper atomic collection of statistics. We should like to see a standard facility (possibly an IOCTL for the /dev/mem device) whereby a list of areas of kernel address space can be copied to a user process as an atomic action protected from interrupts if possible. This would help to ensure that statistics-gathering which currently involves several separate seek/read operations, for example getting both the NFS client statistics and the client RPC statistics, could be changed to get all of the information at the same time. It is not going to be possible (or even wise) to try to lock the structures to prevent data access, but it would still be useful to get all of the data as a single operation. As an extra flourish, the kernel would report the real time at which the copying took place.

6.2. The Uncertainty Principle

Modern physics accepts that it is not possible to know beyond a certain accuracy both the position and the velocity of an electron. This is an instance of a more general result called the Uncertainty Principle which comes from the fact that to measure some properties of a physical system you necessarily interfere with it and change some of its other properties. This interference effect is also true of performance measurement in computer systems: if you run the "ps" utility to find out which process is using most of your CPU time, the answer will be "the ps utility".

Benchmarking is affected by the uncertainty principle in a different way. The more precise and meaningful you want the answer to be (so that benchmark results are easily comparable), the more specific you have to be about the details of the test. The more you restrict the test the less it relates to activities which don't do exactly what the test does. Ultimately, like the physicist who knows the precise position of the electron at the time of the experiment, but has no idea where it went next, you know exactly how the system performs benchmark X but have no way of relating this result to its performance in other areas.

One solution to this dilemma is to accept that though Computer Science deals with small numbers of mathematically definable systems, the combined behaviour of these systems is so complex that a mechanistic approach is not appropriate. The statistical techniques of the biological and social sciences are aimed at interpreting experiments where the experimental subjects are too complex to be modelled completely mathematically, and all results are given as probabilities and confidence levels.

Predicting computer performance (which is usually part of the purpose of benchmarking) stands to gain yet further from the mathematics of biological systems: the computing resource available and the computing resources required are often interrelated in a way similar to predator/prey systems. A more powerful computer actually encourages people to do more complex things, or to do the previous things more often.

6.3. Where next for NFS server benchmarks?

Further work in this area should include re-writing the Legato benchmark to take into account the observations given elsewhere in this paper, and should concentrate on improving the statistical analysis of the results. Work should also be undertaken to explain fully the differences between the Encore results and the Legato results, to ensure that no further "surprises" await the unwary.

Acknowledgements

The author would like to acknowledge the generosity of both Encore and Legato Systems in making their benchmark code available to the public. Thanks is also due the various research laboratories at QMW who allowed their machinery to be used in these tests, and to the various vendors who loaned equipment to be evaluated.

Appendix 1: Where to Obtain the Benchmarks

The Legato benchmark is listed under NHFSSTONE in [Sti90a] as being available via anonymous FTP from bugs.cs.wisc.edu.

The Encore benchmark is available from Encore Systems via email?

References

[Bac86a] M.J. Bach, *The Design of the UNIX Operating System,* Prentice-Hall (1986).

[Bix90a] P. Bixby, *Private communication* (June 1990).

[Inc89a] Legato Systems Inc, *NHFSSTONE*, July 1989.

[San85a] R. Sandberg, "The Sun Network File System: Design, Implementation and Experience," *Sun Technical Report*, Sun Microsystems Inc. (Summer 1985).

[San89a] R. Sandberg and B. Lyon, "Breaking Through the NFS Performance Barrier," *SunTech Journal* **2**(4), pp. 19-27 (Autumn 1989).

[Sha02a] W. Shakespeare, *Macbeth*, 1602.

[She89a] B. Shein, M. Callahan, and P. Woodbury, "NFSSTONE: A Network File Server Performance Benchmark," *Encore Technical Report*, Encore Computer Corporation (1989).

[SPE89a] SPEC, "SPEC Benchmark Results," *SPEC Newsletter*, Systems Performance Evaluation Cooperative (Fall 1989).

[Sti90a] R. Stine (ed), "Tools for Monitoring and Debugging TCP/IP Internets and Interconnected Devices," RFC 1147, Network Information Center (NIC@NIC.DDN.MIL) (April 1990).

For further details, contact
The Secretariat

European UNIX® systems User Group

Owles Hall, Buntingford, Herts SG9 9PL, UK
Tel: + 44 763 73039
Fax: + 44 763 73255
Network address: euug@EU.net

ISBN 0 9513181 8 7