Open Systems & UNIX

Sponsored by

UniForum
The International Association of UNIX Systems Users

1991 Conference Proceedings

January 22 – 24, 1991
INFOMART
Dallas, Texas

UniForum™
The International Conference of UNIX® Systems Users

1991 UniForum Conference Proceedings

January 22-24 1991
Infomart
Dallas Texas

Preface

The UniForum Conference is sponsored annually by UniForum, The International Association of UNIX Systems Users. In 1991, the three-day event is being held Jan. 22-24 at the Infomart in Dallas, Texas.

This document contains reprints of the technical presentations of the conference, including the names and corporate affiliations of the speakers. The papers are arranged in the order presented at the conference. Author, Keyword, and Plenary and Panel Speaker indexes appear at the back of the book, as well as comprehensive details and contact information on the Plenary and Panel Sessions.

For more information on individual papers and presentations, contact the respective author or speaker directly at the address noted. For details on UniForum membership and services, or to order additional copies of the Proceedings, contact UniForum at the address below.

UniForum
2901 Tasman Dr., #201
Santa Clara, CA 95054
Tel: (800) 255-5620
(408) 986-8840
Fax: (408) 986-1645

UniForum Program Committee

The following volunteers comprised the 1991 UniForum Program Committee. Under the guidance of committee chairman Ed Palmer, the volunteers coordinated the planning for the tutorials, conference sessions and workshops presented during the UniForum conference.

Nancy Batten
Sequent Computer Systems

William N. Bonin
Hewlett-Packard Co.

Ed Borkovsky
Unican Marketing Services

Brad Burnham
AT&T

Saleem Haider
Digital Equipment Corp.

Mike Hunter
IBM Corporation

Roger McKee
RLM Associates

Raanan Peleg
Hewlett-Packard Co.

Ray Swartz
Conference Tutorial Coordinator
Berkeley Decision/Systems

In addition, each person served on the UniForum Technical Review Team, which considered the technical papers presented at the conference to determine the 1991 "UniForum Best Technical Paper." The following people also served on the review team.

Jeffrey Haemer
Interactive Systems

Mary Hesselgrave
AT&T Bell Labs

Irene Hu
Digital Equipment Corp.

Frank Papierniak
Zoran Corporation

UniForum 1991
Conference Proceedings
Table of Contents

Application Development in the ANDF Model

Ronald G. Smith
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
(617) 621-8984
rsmith@osf.org

Application Development in the ANDF Model

Ronald G. Smith
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
(617) 621-8984

This paper describes the goals of an Architecture-Neutral Distribution Format (ANDF), the development environment, and the steps an independent software vendor or an application developer has to perform to develop applications that could be distributed in ANDF.

Introduction

Architecture-Neutral Distribution Format (ANDF) is a new technology for distributing application software to users. Today, current methods include linked or unlinked object code, and source code. Object code is platform specific; it is dependent on both the hardware and operating environment. If application developers want to distribute their code to multiple platforms, several object files must be built, tested, supported and stocked. Source code provides an alternative to multiple distribution copies. However, there is no protection for proprietary information such as algorithms and data structures. It also requires a compiler at the user site for the source language. Until recently there was no standard for C, the most popular UNIX® language, a fact which increased the risk of an application failing to compile.

ANDF provides a single means of distribution to multiple platforms. Under ANDF, the application developer uses an ANDF producer to translate portable source code into ANDF code. The producer is very similar to a compiler front-end; it performs syntax and semantic checking of the language and generates an intermediate representation of the source. The ANDF code is then distributed to end users. A user will install the application by running the ANDF installer on their system. The installer will complete the compilation process and create executable binaries that the user can run directly on the platform. The installer performs the same function as the back-end of a compiler, including storage allocation, code generation and optimization, and that of a linker, binding ANDF code to native libraries. See figure 1.

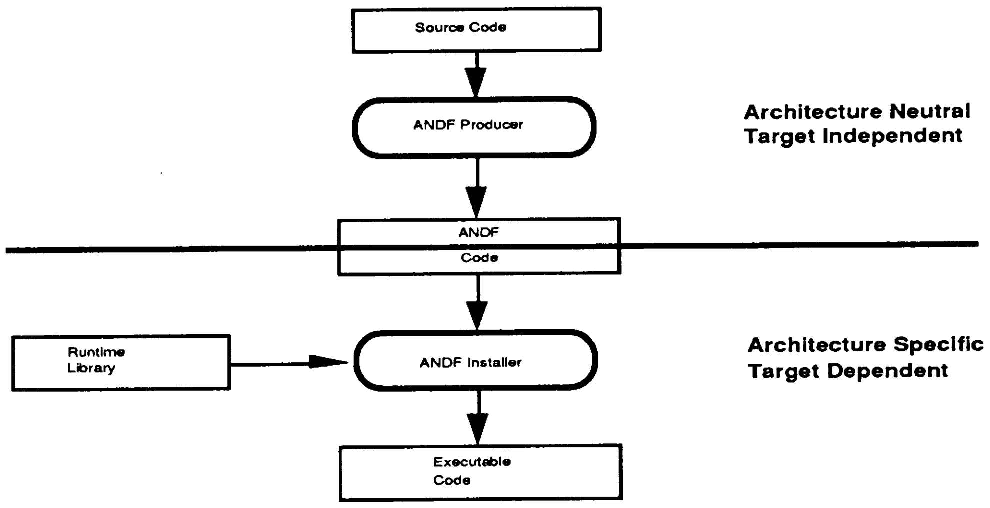


Figure 1. Overview

Goals of ANDF

The goal of the ANDF program is to promote growth of the open systems market by attracting a rich set of applications software to these platforms. This will be accomplished by the development of an architecture-neutral format for software distribution. ANDF will enable software developers who create portable source code to automatically convert it to ANDF, package it once, and have it run on all platforms supporting ANDF. The developer will be able to create and market software independent of hardware platforms. ANDF will also seek to reduce the level of effort to migrate the application to multiple hardware architectures and increase the lifetime of applications.

Increase Software Availability

Currently, software vendors may develop their applications in a portable way so they can easily migrate the software to many different hardware platforms. Because they distribute software in binary form and binaries are not portable, they must package and distribute their products in a different way for every platform or machine architecture on which the software will run. Vendors' software distribution is limited by the number of machines they can afford to support. As a result, they must decide which platforms are most important.

By providing a single software distribution format that is architecture-neutral, the promise of portability can be fulfilled: hardware and software truly can be separated. Software vendors will be able to provide their software on more platforms. The net result will be dramatically increased availability of software for open systems.

Facilitate Shrink-Wrapped Distribution of Software

A single, hardware-independent distribution format will enable software vendors to distribute their software in high volumes. It will also make possible shrink-wrapped, mass market software for open systems and provide the end user with all the associated benefits:

- personal, consumer software products
- low-cost
- retail purchase.

De-couple Software Purchase from Hardware

ANDF should enable a user to make software purchasing decisions without concern for the full range of hardware platforms they plan to use. In particular, the user should be able to purchase software for their workstations without concern for the underlying hardware architecture. Furthermore, they should be assured that the application will behave consistently across all architectures.

Reduce ISV Effort to Target Multiple Platforms

Related to the goal of increasing software availability is the goal of reducing the level of effort an ISV must make to support multiple platforms in the following areas:

- development,
- distribution,
- maintenance, and
- testing.

ANDF will also define what is needed to write an ANDF-compliant application and thereby aid the ISV in writing portable code that is truly hardware independent. It is also a goal to increase the consistency of function and behavior for C programs across these platforms. Maintaining one version of the application source will reduce the software vendor's maintenance load. Although we still expect that many vendors will test their applications on all key platforms, we believe the amount of overall testing required will be reduced and even eliminated in certain cases, such as the testing required to support new operating system releases.

Software Longevity

Hardware innovation will flourish and the spirit of open systems will be kept alive by providing end users all the associated benefits of scalability, portability, and interoperability. As such, ANDF will allow software to have a life cycle totally independent of the underlying processor technology.

Currently, many software products must be recompiled and redistributed whenever a new version or release of the operating system occurs. It is a goal of ANDF to isolate software from these changes and allow users to merely reinstall the software on their machines.

What about ABIs?

An Application Binary Interface (ABI) or Binary Compatibility Standard (BCS) defines the interface to the operating environment, instruction set and binary software conventions. These software conventions may include data alignment and size rules, function calling convention, error codes and binary object file format. The conventions may even include how the software is stored on tape. An ABI conforming application will install and execute properly on any certified ABI platform. There are ABIs for the Intel 386 and i860, Motorola 68000 and 88000, and Sun SPARC architectures.

Although ABIs are currently available, they are only part of the solution for shrink-wrapped software. ABIs are still operating system and processor dependent, so many versions of the same application are needed to support the entire market. Since ABIs are processor dependent, they need to be redefined for each new platform. With ANDF, the developer could develop a single version. ANDF does not preclude ABI's: ANDF installers will generate the appropriate ABI on a target platform that conforms to an ABI standard.

Development Environment Comparison

The development environment is slightly different for developing an application under ANDF versus a typical environment in use today. Today, the application developer may have target independent source, target specific source and maybe some key assembly language routines. Target independent source may use header files (both system defined and user defined) to isolate characteristics of the operating environment or functionality. Target specific or target dependent source contains knowledge of the underlying hardware architecture or services not available on all platforms. For example, assumptions about whether data is stored in "big-endian" or "little-endian" byte ordering, or whether a specific GUI is available. Usually, the more target independent code there is, the more portable the application is.

Traditional Environment

In the traditional environment, a compiler is used to translate the high level source language into binary objects which contain machine instructions for a particular platform. A linker is used to combine binary objects and libraries to produce an executable image of the application. Other executables may be combined with data files and install scripts to produce an application package. This package is distributed to the users. It is platform specific and it is usually for a single operating system. The whole process takes place at the developer's site.

The end user purchases the application and installs it on the system. This may include running a script and providing information during the process. Executable objects and libraries are moved into their expected location and the environment is set up to run the application. See figure 2.

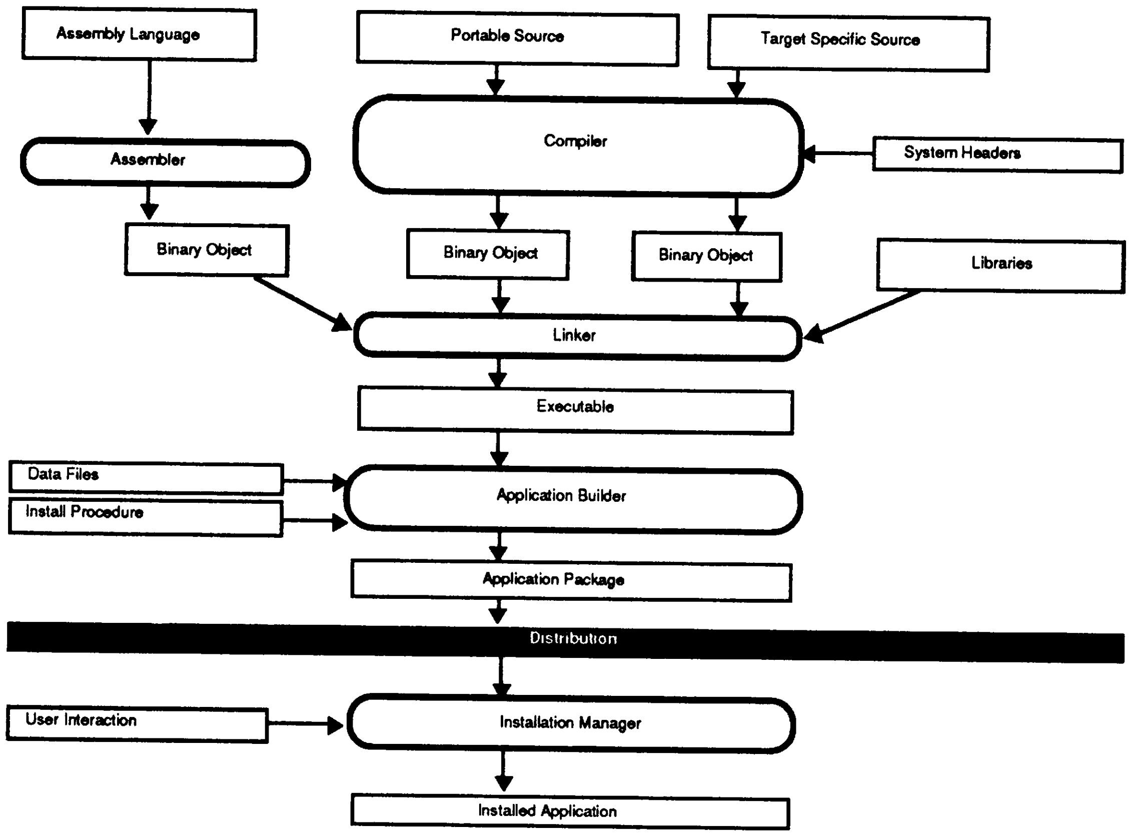


Figure 2. Traditional Development Environment

The application is tested by installing it on each platform and running a set of tests. The output of tests may be slightly different from one platform to another depending on the target environment. When the developer wants to debug a problem, the application is compiled with a debugging flag and run under a symbolic debugger.

ANDF Environment

Using ANDF, the steps an application developer performs in translating source code into an executable image and distributing it are different. The compiler will be replaced with two separate components, the producer and installer. The producer is invoked just like a compiler, giving it command line options and names of source files to process. The producer analyzes the source to verify that it conforms to the syntax and semantic constraints of the language. ANDF object modules are generated that represent the original. Although the producer never makes target dependent assumptions, a producer can generate ANDF from source which is explicitly target specific. A useful example might be code which assumes the availability of a particular GUI. Optionally, an ANDF linker could be used to link ANDF generated from different source files together to form a single ANDF object module and remove resolved external names.

The application builder will construct the ANDF distribution package from ANDF object modules, native code modules, data files and installation procedures. The ANDF application package is distributed to users.

The user purchases a version of the application in ANDF and runs the ANDF installer to install the software on the platform. The installer will complete the processing that is necessary to turn the original source code into an executable binary. The installer will also call the native linker to combine binary objects and native libraries. Linking may also be deferred to run time as with ABIs. See figure 3.

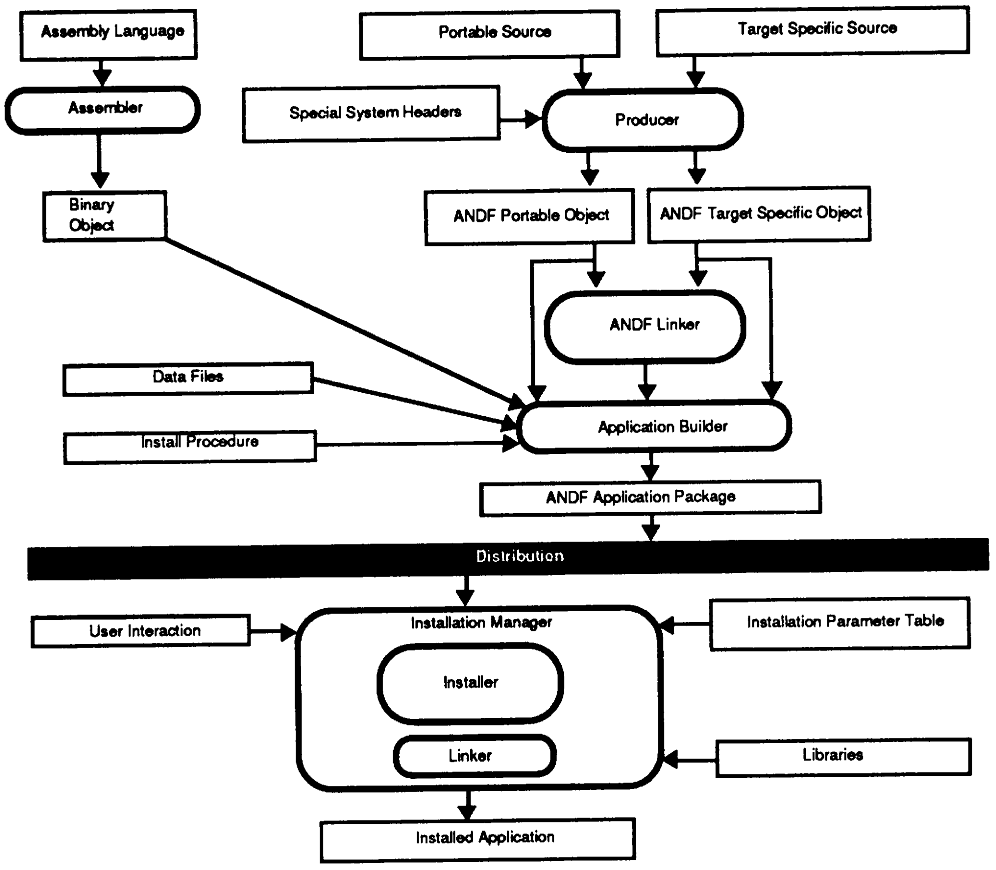


Figure 3. ANDF Development Environment

Testing and debugging is almost the same as in the traditional environment. Developers will continue to test their applications on platforms which are considered critical. The application will be installed on a test system and the test suite run. The developer specifies an option to the producer that will generate additional information for debugging in the ANDF object module. The installer will fill in the missing information such as variable locations. The application is then run under a symbolic debugger. After the bug is fixed, the developer turns off the symbolic debug option and creates another ANDF application package.

Evolution Plan for Releases of ANDF

The first release of the ANDF technology will support the ANSI C language. Installers will be available for most of the popular open systems platforms. Successive releases may add additional languages such as C++, FORTRAN and COBOL along with installers for additional platforms. Tools also will be developed to help the application developer write more portable source. These tools may include a portability checker (like a vastly improved *lint*) to flag possible non-portable use in source code and an environment checker to verify that the platform conforms to ANDF requirements.

As the developer modifies the source code to run under ANDF, the process also will make their application more portable in the traditional development environment. For today's developers, portable code enables them to reach the widest audience with the least effort. The application developer's effort to create portable code for ANDF is not wasted.

Why is ANSI C not enough?

If there is a standard language definition for C, why not distribute C source code? Two disadvantages have been cited previously: possible loss of proprietary information and the necessity for a development environment at the user site. The third disadvantage has to do with portability. Strict conformance to the ANSI C standard by an application does not in itself guarantee that the application is portable between architectures. Compilers are free to choose the behavior of implementation-defined areas in the standard. For example, ANSI C does not define whether a char data type is signed or unsigned (ANSI 3.2.1.1). In the following example, the variable i is either positive or negative depending on whether c is signed or unsigned.

```
char c;
int i;

c = SCHAR_MAX; c++;
i = c;
if (i > 0)
      printf ("char is unsigned\n");
else
      printf ("char is signed\n");
```

If the developer writes an application that depends on char being unsigned, it could fail in an environment where char is signed. ANDF will help solve this problem. The developer will use a single ANDF ANSI C producer to process his source code for multiple platforms. The producer will treat all chars as unsigned and provide an option to allow char to be signed if the original code is written assuming signed char. Using traditional compilers, application developers do not enjoy such levels of consistency.

Another area that the application developer has little control over is the size of data types. How large is an int? How long is a long? The ANSI C standard only specifies the minimum maximum values that C data types are required to have and the relationship between short, int and long. In the following code fragment, the signedness of an expression is determined by the size of int and long data types.

```
signed      long  sl;
unsigned    int   ui;

sl + ui;
```

The expression will be unsigned on platforms where ints and longs are equal size, but signed if longs are larger that ints. This behavior results from the C rule of "usual arithmetic conversions" (ANSI 3.2.1.5). As in the char example, the ANDF ANSI C producer could define the size of data types. This would lead to consistent behavior in this example. However, this would only lead to more serious problems. Discrepancies between the size of an int passed between an application and libraries could lead to subtle errors. The close() function has both an int parameter and a function return type of int, and the div_t structure contains two int members initialized by the div_t() function.* The solution is for the developer to properly cast mixed type expressions to ensure the desired results. The ANDF installer will allocate the proper sized data types so ANDF applications can share data with natively compiled code and libraries.

The ANSI C standard contains an appendix devoted to portability issues. Appendix F lists all the unspecified, undefined and implementation-defined behaviors contained in the standard. Many of these behaviors are in this section because they are dependent on the underlying architecture and operating environment. The standard was written to promote portability, not enforce it. It allows a developer to write nonportable code just as easily as portable code. It is up to the developer to follow self-imposed rules to insure portability.

ANDF Impact on Source Code

The requirements that ANDF technology places on source code are different between source classified as target independent or target dependent. ANDF application developers should maximize target independent source in their application; they should minimize and isolate target dependent source. When target dependent source can not be avoided, they must make sure that each target supported has a version of the target dependent source in the ANDF application package. Whenever possible, the application developer should supply a portable version of such target dependent code so that the ANDF application would execute on all ANDF targets.

Target Independent Source

There are very few restrictions ANDF imposes on the use of C. The application developer needs to conform to ANSI X3.159-1989, the language standard for C (ANSI C), and avoid some preprocessing constructs and a few of the portability issues (Appendix F) in the standard. ANDF provides an incentive for the developer to write more generic portable code than code which is customized for a specific platform.

The key requirement of target independent source is that it is written in a portable fashion. ANDF will not make unportable code portable. In the past, a portable application was one which could be moved to a different platform with very minimal effort. For ANDF, the developer must take a more proactive approach towards portability. Traditionally, each port to a new machine gave the developer an opportunity to slightly modify the sources if necessary. Portable C code for ANDF must be source which requires no modification, as it will be the single input to one producer which generates ANDF for many installers. Portability of the application is affected by dependencies on 1) machine architecture, 2) operating environment and 3) software and hardware implementations.

Portable code must not make any assumptions about the underlying machine architecture. For example, the range of data types, their representation and layout in memory are areas to avoid. The developer should avoid assuming that ints are 32 bits and that the floating point representation is IEEE. The ANSI C standard does specify the minimum range data types must have and the developer can safely rely on these ranges for ANDF applications.

The operating environment for ANDF applications is defined by IEEE Std 1003.1-1989 (POSIX) and ANSI X3.159-1989 (ANSI C). Any use of functionality outside these standards is not guaranteed to be present on every ANDF platform. ANDF does provide a mechanism to encapsulate dependencies on functionality not in ANSI C and POSIX. See Target Dependent code section.

* A program and its libraries must share a common idea of objects.

As applications are developed, defects in the implementation of the compiler, operating system or even microcode are sometimes worked around. This can be very frustrating, particularly when porting an application to a platform and discovering a large number of defects in software other than yours. ANDF allows the developer to translate his source into ANDF code on a single system for multiple targets. Using a single producer will lead to more consistent application behavior. In the PC clone market, the market defines compatibility between different implementations. The application developer can expect the same from the ANDF market.

The ANSI C standard documents a number of portability issues in Appendix F. A strictly conforming program and a portable application should not depend on a particular behavior. Unlike a strictly conforming program, target independent source may rely on implementation defined behaviors which are guaranteed to be consistent across all ANDF platforms. For example, ANDF specifies the number of significant initial characters in an identifier without external linkage, the maximum number of case values in a switch statement and the value of an integer character constant that contains more than one character. ANDF will also allow the developer to select a particular behavior and guarantee that behavior on all platforms. Examples include the sign of the remainder on integer division (when one operand is negative), whether a "plain" char has the same range of values as signed char or unsigned char, and whether a "plain" bit field is treated as a signed bit field or as an unsigned bit field.

Target Dependent Code

In general, the application developer make tradeoffs when it comes to performance, functionality, and code portability. ANDF allows the developer to include target dependent (non-portable) code in his application. There are two methods: target dependent ANDF object modules and a conditional compilation capability similar to #if.

A target dependent ANDF object module can be created by either the native assembler or compiler, or by the ANDF producer with target dependent assumptions contained in the source. The ANDF application builder will label these modules as target dependent and attach a dependency requirement to them. The ANDF application builder will group together multiple target dependent modules that represent the same functionality. One of these modules could be labeled as a portable version. When the user installs the application, the ANDF installer will process only those modules whose dependency requirements are met. If, in a group of target dependent modules that are linked together, none meet the dependency requirement, the module labeled as portable will be selected. With ANDF, the developer could create a 100% portable application and continue to compete on platforms where performance or a particular user interface is necessary. See figure 4.

A well defined usage of conditional compilation (#if and #ifdef) is available to the ANDF developer. If the condition is known to the producer, the #if may occur anywhere. For example, "#ifdef DEBUG", where DEBUG is defined or not defined on the command line, can be resolved during the producer stage. If the evaluation of the condition must be deferred until install time, such as in "#if INT_MAX > 32767" where INT_MAX is defined in the header file limits.h and its value is unknown to the producer, the location and use of the #if might be more restrictive. For instance, ANDF might be able to handle conditional compilation only if it cleanly divides executable statements in the source. ANDF might not allow it to be used in the middle of statements or declarations. For example:

```
  i = i +
#ifdef MIPS
  100;
#else
  j * 10 - 3;
#endif
```

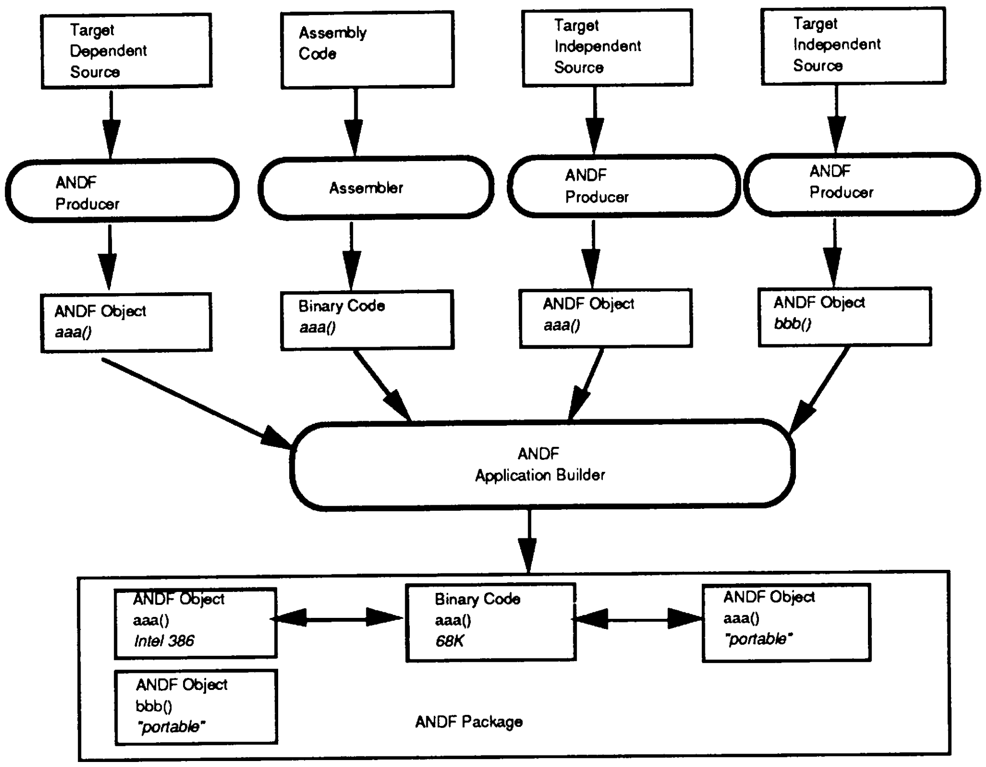


Figure 4. ANDF Package - with Target Dependent Code Modules 1

The ANDF producer will generate an ANDF code fragment and label it with the dependency requirement. Care must be taken to either provide a #else clause whenever necessary, or to assure that there is no effect on platforms whose target characteristics do not satisfy the defined condition.

In both methods, the ANDF installer will interpret a system wide installation parameter table (IPT) or use user interaction during the install process to determine the correct values for each dependency. The IPT is a simple database which will be available on all platforms supporting ANDF installations. It will describe in a standard manner the hardware and software attributes of the current environment, for instance endianness or identity of the graphical user interface.

Summary

Architecture-Neutral Distribution Format will promote growth of open systems by bringing a wealth of applications to a large number of these platforms. It will allow a developer to create a single version of an application that can be installed and run on multiple platforms regardless of the hardware architecture or operating system. Application developers will have the potential to compete in new markets and realize a reduction in both their development and manufacturing costs. Users will have a larger selection of applications to choose from and they will be insulated from the hardware life cycle. ANDF will work with other existing technologies for distribution, such as ABIs and portable source code.

The application developer's target independent source must be written in ANSI C, use only ANSI C and POSIX functionality, be portable, and refrain from making assumptions about the target environment. However, there is a well-defined mechanism, compatible with today's methods to include customized or target dependent code in the distribution package. Modified source for ANDF will continue to work under the traditional development environment.

The current ANDF technology does not address some issues particular to the open systems market. The distribution of software in shrink-wrap form is only one part of the solution. A single or limited set of physical medium to distribute software on is highly desirable. Until the industry can reach a consensus on a single medium, application developers will continue to distribute using the most popular media. Another issue ANDF does not provide a solution for is data interchange between platforms. Standards organizations such as POSIX have started to explore the issue of data interchange and possible solutions. For now, ANDF applications will continue to use the same workarounds or proprietary solutions in use today.

Future advances of ANDF technology might reduce the level of testing needed. It will be a significant achievement if application developers only have to test their application on a single ANDF compliant platform and have a high level of confidence that the application will work properly on any other ANDF compliant platform. ANDF installers could also start to appear on proprietary systems. As system vendors implement standard conforming functionality such as POSIX and ANSI C into their proprietary operating systems and provide popular GUIs and other functionality, ANDF compliant platforms and installers could spread beyond the limits of open systems.

Typically, the end user is looking to purchase software that is packaged and installed much as PC software is today. The biggest advantage that end users will see in an open systems market is they will find developers' applications conforming to a specification that permits execution in as many environments as possible. To this end, an intermediate format such as ANDF is the best potential answer to user needs.

1 This is one of the methods to include target specific code.

Beyond the ABI: Attaining "Shrink Wrapped" Software

Rita M. Anderson
NCR Corp.
3325 Platt Springs Rd.
West Columbia, SC 29169
(803) 791-6864
rita.anderson@columbia.ncr.com

Beyond the ABI: Attaining "Shrink Wrapped" Software

Rita M. Anderson

NCR Corporation, E&M Columbia
3325 Platt Springs Rd.
West Columbia, SC 29169
rita.anderson@columbia.ncr.com

ABSTRACT

Attaining a volume of "shrink wrapped" software is a key objective of many system vendors who are or will be shipping UNIX System V Release 4. This paper examines the contribution of the System V ABI towards this goal, the factors necessary to achieve this volume, and the methods which system vendors employ in the effort. The paper concludes with the recommendation that a reference port provides the optimal mechanism for evolving a "shrink wrapped" software environment.

The Value of "Shrink Wrapped" Application Software

There are few successful models of "shrink wrapped"† software which span multiple vendors' hardware platforms. The most prominant examples are MS-DOS and Santa Cruz Operations' XENIX / UNIX products.‡

Attaining a volume of "shrink wrapped" software is a primary objective of platform and system vendors who are or will be shipping UNIX System V Release 4. Establishing a base of "shrink wrapped" software provides a vendor a significant time-to-market advantage over his competitors. As Figure 1 illustrates, traditional product delivery processes require some form of Beta delivery to foundation software vendors for initial ports. These are languages, data bases, networking products, etc. Once these are available, then the platform can be made available to the myriad of horizontal and vertical software vendors to secure ports of those applications. The process may span the better part of a year, rendering the platform technology unavailable to the end user during that period.

† The term *"shrink wrapped"* implies software that is marketed such that the license is activated by opening the cellophane packaging. The term has been commonly applied to software which is made available on multiple vendor systems through referral catalogs without explicit ports to these platforms.

‡ MS is a registered trademark of Microsoft, Inc.

XENIX is a registered trademark of Microsoft, Inc.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

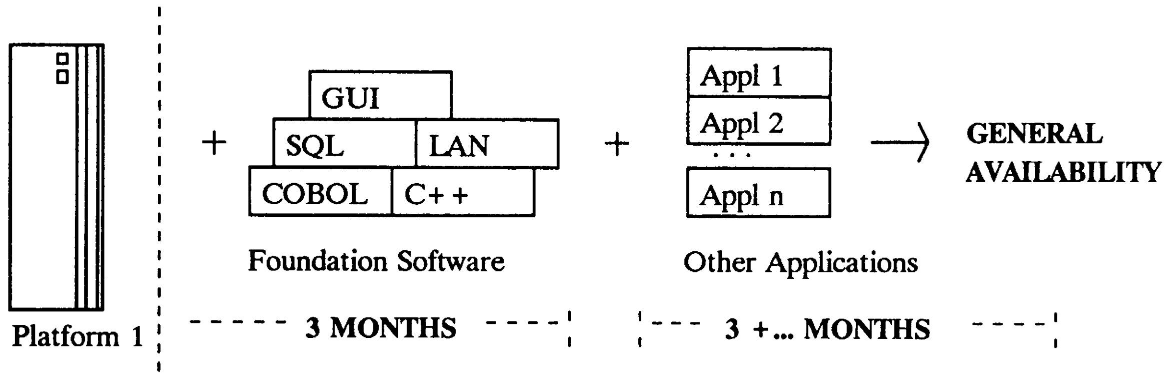


Figure 1. The Traditional Product Delivery Process

Figures 2 illustrates the competitive advantage of "shrink wrapped" software; the vendor who can leverage an existing distribution of "shrink wrapped" referral software is empowered to provide new platform technology to the marketplace as soon as it is released from the development process.

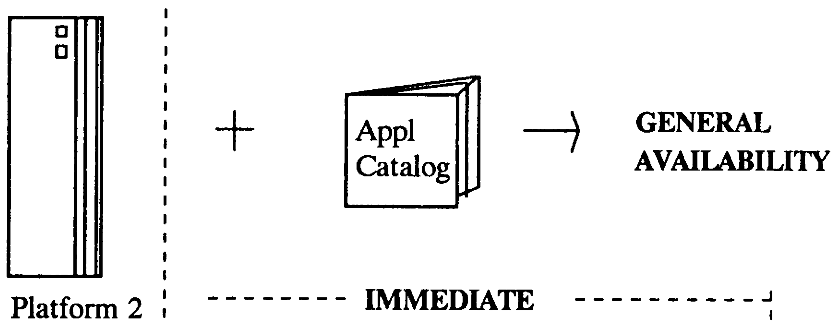


Figure 2. The Advantage of 'Shrink Wrapped'

The platform vendor may, in fact, choose to secure specific ports of foundation or other application software to take advantage of the new features or technology introduced with the platform. The difference is that the release of Platform 2 to the general user is independent of the completion of these activities.

UNIX International promotes System V as the only UNIX system to be described by an Application Binary Interface (ABI). The purpose of the the System V ABI is to facilitate application portability among compliant platforms. In fact, the combination of the generic ABI specification and a processor-specific specification facilitate the migration of application binaries among compliant platforms which utilize the same processor architecture.

Although the ABI is an excellent step towards establishing a "shrink wrapped" distribution of application software for UNIX System V Release 4 platforms, it can not, in and of itself, guarantee the availability of "shrink wrapped" software.

The System V Application Binary Interface (ABI)

Source Standards: The Derivation of the ABI

The ABI is actually derived from the System V Interface Definition (SVID) Issue 3. The intent is that compliance to SVID is the vehicle by which application source can port from one vendor's SVID-compliant platform to another vendor's SVID-compliant platform.

The System V Interface Definition defines both the *presence* and the *behavior* of the various components of the UNIX operating system. The presence of a utility, system call, library function, etc. is either required or not required for compliance. The run-time behavior of components is also described, including the definition of parameters, return values, and any effect that the component has on other system components.

SVID provides for base and extensions. A SVID-compliant system may contain only the base, but if the files for a particular extension are present, then the set must comply to the interface definition for that extension.

The base defines the minimal run-time operating system. Extensions include the following.

- The Kernel Extension (memory mapping calls, shared memory, ...)
- The Basic Utilities Extension (run-time command set: sh, cat, ...)
- The Advanced Utilities Extension (vi, mailx, ...)
- The Administration Systems Extension (fsck, mount, runacct, ...)
- The Software Development Extension (cc, lint, make, ...)
- The Terminal Interface Extension (terminfo, curses, ...)
- The Real Time Extension (timer and time of day system calls)
- The Remote Services Extension (RFS, RPC, XDR, ...)
- The Windowing System Extension (X Window System† Version 11 commands and utilities)

† X Window System is a trademark of the Massachusetts Institute of Technology.

SVID specifies the minimum directory tree, required system files such as */etc/passwd*, and the formats associated with these. Common device interfaces to the point of *ioctl's* are included as well as libraries such as *terminfo*; actual device names beyond the generic */dev/console*, */dev/null*, */dev/tty* are not included. Device interfaces to disks, tape drives, etc. are not necessarily common to all systems and are, therefore, not defined in detail.

The Intent of the ABI

The ABI was developed because the System V Interface Definition is not sufficient to guarantee that an application source compiled in a SVID-compliant environment can migrate from one system to another provided that the systems have the same processor and memory management architectures. SVID does not specify internal data formats, function calling sequences, and internal table formats. Interfaces which are beyond the UNIX kernel and library interfaces such as installation formats, supported media, etc. are omitted from SVID.

The intent of the System V Application Binary Interface is to leverage the existing source standards, including not only SVID, but X/Open XPG3 and the IEEE POSIX 1003 Base, to define both presence and execution behavior of the various operating system components and to provide a definition for low-level constructs. The ABI is divided into a generic definition and a processor-specific definition. The generic portion defines those components common to all architectures such as system call interfaces, library interfaces, system file formats, etc. The processor-specific section describes the implementation of the low-level constructs for that particular processor architecture. These include data type formats, stack frames, page sizes, etc.

The purpose of maintaining an ABI is that the binaries generated from application source **written to comply to the interfaces defined by the ABI** will be migratable across all conformant platforms.

The ABI Specification

Analogous to the SVID classification of *base* and *extension*, the ABI specifies *base* and *optional* components. An ABI-compliant platform must include the base components; optional components, if present, must conform to the interface as described.

The ABI does provide definition of system components not included in SVID. A summary of the operating system features included in the ABI follows.

- Software Packaging

  The ABI defines physical distribution, media formats, and layout of the required data and script files. The definition of the application software distribution is key to facilitating "off-the-shelf" software.

- Object File Formats

  The ABI specifies the Executable and Linking Format (ELF) which supports relocatable, executable, and shared object files. To increase portability, ELF provides for an architecture-independent definition of a word and utilizes no bit fields. The ABI defines the process of resolving dynamic references.

  Note that although System V Release 4 does provide run-time support for UNIX V.3 Common Object File Format (COFF) executables and relocatables, the ABI specifies only ELF.

- System Libraries

  The ABI specifies the basic system libraries as required: *libc*, the C library as defined by SVID, ANSI C, POSIX, etc., *libsys*, which supplies the traps to the system calls, and *libnsl* which describes the transport layer interface (TLI).† *Libx*, which defines the X Window System, is optional.

  These libraries are specifically required to be implemented as dynamic shared libraries and are, therefore, included in the ABI. Other libraries defined by the SVID extensions are not. Since they represent code which is linked into the application program, their run-time behavior is actually part of the application and not described by the platform ABI.

  Interdependencies between libraries are resolved by the provision that application executables must provide a complete dependency graph during execution.

- System File Formats

  In most cases, the ABI requires that conforming applications access all system files whose formats are not defined by SVID, via the programmatic interfaces provided.

  File formats included in the ABI are the general archive packages, the *cpio* archive format, and the *terminfo* data base.

- Networking Data Formats and Protocols

  These functions implemented in *libnsl* are optional, but if present, must comply to the ABI. Included are the external data representation (XDR), remote procedure call (RPC) protocols, and the Data Encryption Services (DES) authentication services. Note that the act of binding a client to a service is considered a higher level function and is, therefore, not included in the ABI.

- Application Environment

  A minimal application environment is included. A set of basic commands: *sh*, *cat*, *cpio*, *kill*, etc. are included and must be accessible via the default *PATH* environments.

  Also included for the application execution environment is required system functions and a rudimentary directory tree. (The directory tree does exceed the list of root level directories included in SVID.)

What the ABI Realistically Provides; What's Missing

UNIX International is justified in the efforts to advertise the ABI because a written application binary interface not only guarantees a "common denominator" between

† Other networking capabilities provided within *libnsl* such as remote procedure calls, authentication, and others are optional.

compliant systems, but facilitates source migration among different hardware architectures.

What adherence to the ABI can not provide is the assurance that **any** application which executes on one ABI-compliant system will execute on another.

Although one might argue that no paper standard can document every application programmatic interface, the ABI actually provides a very limited set of required directory structures, no specific devices names, no device driver interfaces, and less than 60 utilities (or shell commands).

For example:

- A fast backup tool that references a specific device node to access the tape drive can not be guaranteed to execute on another ABI-compliant system.
- Software developed on V.3 systems are, by definition, not ABI-conforming applications.†
- When a V.3 application is ported to V.4, it may compile successfully, but during its execution, an attempt to reference the */usr/spool* directory may fail since the V.4 equivalent directory is */var/spool*.‡
- An application which contains a shell script which uses the *awk* command is not guaranteed to execute on another ABI-compliant system. Although *awk* is specified in the SVID Base Utilities, it is not included in the list of commands required for ABI-conformant systems.

The ABI, therefore, specifies a somewhat restricted subset of system interfaces. The segments marked with vertical bars in Figure 3 illustrate the relative portion of UNIX System V which is described by the ABI.

† The ABI requires that application software utilize the Executable and Linking Format (ELF).

‡ Although not specified by the defining standards, the links to many of the standard V.3 directories are included in the current V.4 port, as delivered from UNIX System Laboratories today.

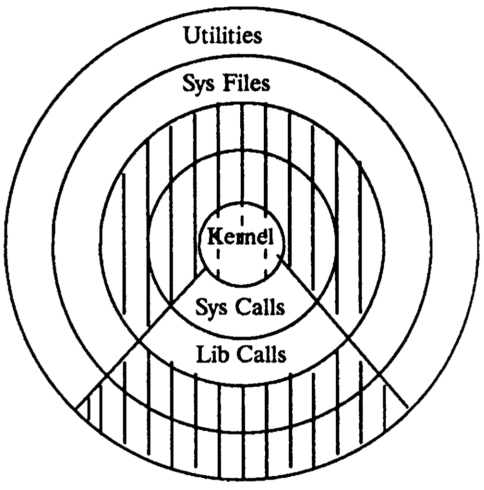


Figure 3. The System V ABI

Independent software vendors (ISV's) demand an assurance from system vendors that not only the base operating system be standardized via an ABI, but also the application environment be identical among compliant platforms. The application programming environment is best defined as the set of software required to execute the application that is not directly packaged on the same distribution media with the application. The availability of "shrink wrapped" software is most dependent on the degree to which this assurance is provided to ISV's by the system vendors and the volume of platforms represented by the vendors who are willing to commit.

Alternative Approaches to the Problem

Groups of platform or system vendors which share a common processor architecture have attempted to offer this guarantee via several methods.

- **Test Suite Verification**

  Platform conformance test suites have grown in popularity in recent years. ISV's can be assured that the set of platforms which support the application base are tested for conformance to SVID, the X/Open Programming Guide, and soon IEEE POSIX definitions via the test suite method.

  The test suite executes as an application on the platform and verifies that the implementation of each interface is consistent with the documented standard. Test suites are very effective means of building credibility; however, test suites may or may not cover 100% of the interfaces defined. What this method does assure is that there exists a common subset among the platform implementations which pass the suite.

  Several groups have attempted to measure the degree of an application's conformance to a particular standard via a test suite. At the source level, the test serves much as a compiler front-end providing syntactical and semantic analysis of the code to determine potential incompatibilities. Test suites which operate on the application binary typically provide a simulation of the operating environment. Such suites may appear to provide a strong assurance to the ISV community, but it is difficult to assess the degree of accuracy or coverage of these. Since the suite must be used for any application, it is virtually impossible to provide a suite which can detect **any** failure in compliance.

The ABI can best be interpreted as a contract among platform vendors to ensure a common operating environment. Given this perspective, it is incongruous to focus on the conformance of the application source developed by the ISV as opposed to the assurance of a standard operating environment provided by the platform vendor.

Although platform vendors can use the test suite method to demonstrate to the ISV that each platform implementation is consistent with at least a subset of the documented operating system, the test suite method does not address the larger scope, the application execution environment.

- **Porting or Testing Centers**

  Many groups of platform vendors which share a common processor architecture are establishing porting centers or testing laboratories. The centers serve as a convenient location where the ISV can test his application on any or all of the platforms installed at the center.

  Assuming that the majority of applications which are brought to the center do execute successfully on all platforms tested, the method can be very effective in assuring the ISV community that the set of platforms do represent a common application environment. The degree of coverage, and, consequently, the effectiveness of the assurance, is limited by the number of platforms represented in the center and what percentage of the total population that number represents.

  An additional disadvantage of this approach is that the burden of proof is placed on the ISV. It is the ISV who must bring his application to the center and satisfy himself that the application is portable to the various platforms available there.

- **Establishment of a Reference Port**

  If a group of platform vendors sharing a common processor technology can identify sufficient commonality among the system architectures to derive a reference hardware platform, then the group can utilize a reference port. The reference port is the minimal operating environment defined by the ABI and is, therefore, a tangible subset of UNIX System V Release 4 that is defined by the ABI. The reference port can actually can best be viewed as the standard test or application execution environment.

  The degree of coverage is, by definition, 100%, since the group is endorsing the reference port as a tangible ABI. More importantly, the burden of proof of conformance is now shifted to the platform vendor. It is the responsibility of the platform vendor that his system maintain compatibility with this minimal set; he must ensure that any added feature functionality provided in his packaging of System V contain the identical application interfaces. The ISV is, therefore, assured that if his application executes successfully on the reference port or test environment, then it will execute on all platforms which are compliant with the generic and the particular processor-specific ABI.

The reference port method has the added advantage that it can address the application execution environment beyond what is defined by the operating system standard: the reference port can evolve to encompass specific features or products as they emerge as *de facto* standards in the marketplace at a quicker pace than the documented standard will evolve.

The following section examines this process further.

Refining the Reference Port Concept

"Shrink Wrap" Successes Today

The establishment of a reference port embodies the two factors that have been key to the success of both MS-DOS and SCO UNIX or XENIX:

1. A standardized hardware architecture.

   Both MS-DOS and SCO UNIX / XENIX utilize the PC or PC clone system as the application porting base. Establishment of a reference port does require that there be enough commonality between the various hardware platforms supporting a particular processor ABI so that a majority of the systems can execute a common, minimal operating system. Modifications for installation, hardware initialization and management, and device drivers are assumed. The more the actual system architectures diverge, the more difficult the process of defining a common tangible ABI.

2. One operating system.

   Both Microsoft and Santa Cruz Operations have protected the concept of one operating system: there exist no variants of the standard as exists with the UNIX marketplace as a whole today. Microsoft and Santa Cruz Operations have complete control of the their respective products.

   The platform vendors, building systems based on System V Release 4, target various market segments and differentiate their product set by offering added value features and functionality to the UNIX operating system. Thus, it would appear that the concept of one vendor's providing the common operating system is, at best, naive.

   The ISV, however, is interested in the volume of potential sales for each application port; he, therefore, will tend not to rely on added value features if he can utilize one port for multiple vendors' systems.

   The reference port is that "common denominator" system which provides a port base for the ISV and which maximizes the number of hardware platforms and different vendors' systems for which he can sell his application software.

The Logistics of Utilizing a Reference Port

The reference port is, therefore, a tool for the ISV community; it does not represent a product for the end user community. It may or may not include the development tools.[†] The reference port is basically a standardized execution environment or testbed for the application.

The group of platform vendors supporting a processor-specific ABI can structure the reference port so that each vendor ships the port as a standardized test environment with the package of hardware, software, and documentation that they normally ship to their ISV's.

Another alternative would be to designate one vendor as the provider of the reference port. That vendor would serve as the primary ISV contact for the entire group.

Defining and Evolving a Reference Port

The starting point for determining the contents of the reference port is the ABI specification. The next step is to include those files necessary for booting and establishing a sufficient operating environment for the common hardware base.

The reference port may not necessarily be encumbered by how restrictive the ABI may appear; a reference port can be extended to include the primary source standards:

> As cited in a previous example, an application whose installation scripts depends on the *awk* utility is, by definition, not ABI-conforming, because *awk* is not included in the ABI. Since *awk* is included in the Base Utilities Extensions of the System V Interface Definition, and since most groups of platform vendors view conformance to SVID as a requirement, *awk* would naturally be included in a reference port. The benefit is that the ISV does not have to be burdened with determining whether to redesign his installation scripts to exclude the use of *awk*.

The true advantage of the reference port as a mechanism for promoting "shrink wrapped" application software over test suite execution or porting centers is the ease with which the "common denominator" between the platform, as defined by standards such as SVID and the ABI, can evolve to include the components of the application execution environment.

The components included in a reference port can also extend beyond the documented standards.

For example:

- **Inclusion of Graphical User Interfaces**

  The incorporation of graphical user interfaces in sophisticated application software is growing more prevalent. Although no one graphical user interface technology has emerged as *the standard* technology for UNIX systems such as Microsoft Windows has for MS-DOS, one technology may dominate that portion of the market targeted

[†] Many platform vendors may choose to differentiate their UNIX products with added value development tools, such as high performance compilers. Again, it is the responsibility of the platform vendor to ensure that these tools generate ABI-compliant software.

by a group of platform vendors who share a common processor technology. For example, a reference port for the SPARC processor might include the OPEN LOOK [†] Graphical User Interface product.

Other groups of platform vendors may choose to include two competing interface products, such as OSF/Motif and the OPEN LOOK GUI, thus allowing the ISV the choice of technology.

- **Preservation of Existing Application Environment**

  The announcement of the agreement between Intel, UNIX System Laboratories, and Santa Cruz Operations to extend the Binary Compatibility Standard to incorporate application interfaces inherent to the SCO UNIX products is significant to the platform vendors who plan to offer UNIX System V Release 4 systems based on the i386/486 architecture. The extension of the BCS provides a migration path to SVR4, allowing the existing SCO application base to be applied to i386/486 SVR4 systems.

  The inclusion of the extended BCS interfaces is a viable option for the platform vendors building SVR4 systems based on the i386/486 in defining a reference port for that particular processor base.

- **Inclusion of Common Device Interfaces**

  Where there exist common hardware or peripheral devices among the systems which share a common processor technology, it is likely that the interfaces to those can be standardized to the point of common device names, common interface structures, and common utilities. For example, many systems utilize 3.5" flex as the primary distribution media for application software. If the device naming conventions and the utilities to access the flex drive have already become *de facto* standards for systems based on a particular processor architecture, it is natural that the device names and the utilities would be reflected in the reference port for the processor architecture.

The reference port for a particular processor base can, therefore, evolve to include those portions of the application environment which are most important for the population of systems or platforms represented. Other examples are the standardization of a data base query language for front-end processors, inclusion of Kanji interfaces for the Japanese marketplace, networking interfaces for client-server systems, and others.

† OPEN LOOK is a registered trademark of UNIX System Laboratories, Inc.

Evolving the Documented Standard

Published standards such as the ABI, SVID, and others tend to document what already exists as "common practice" in the existing implementations of the UNIX operating system.

The establishment of reference ports allows the subset of "common practices" to evolve to encompass the features necessary for a complete application execution environment as applications become more sophisticated and more dependent on interfaces beyond the basic operating system interfaces.

The published standards must then also evolve to document these practices as they become *de facto* standards. The current revisions of the System V Interface Definition and the System V Application Binary Interface provide an excellent example of this process:

> The X Window System has evolved to be a *de facto* standard as a windowing system for UNIX platforms. Once the X Window System had become established in the UNIX community, the libraries were documented in both the SVID and ABI specifications.

Thus, the reference port for a processor-specific must reflect the ABI specification for that processor, and in turn, the ABI specification must evolve as the reference environment evolves.

Summary Statements

- The ABI represents an excellent step towards establishing a mechanism for "shrink wrapped" software for System V platforms.
- The ABI, as it is defined today, is not a sufficient mechanism for creating a "shrink wrapped" motion for System V Release 4 platforms.
- The group of platform vendors whose systems share a common processor base and a sufficient common architectural base are best positioned to create a "shrink wrapped" application motion for their products by endorsing a reference port for that processor.
- A reference port provides the ISV with a tangible ABI and an environment to which he can port his applications with the assurance that the applications will be portable to all platforms conformant to the ABI.
- As new technologies emerge, application software will become more dependent on interfaces beyond the UNIX system and library call interfaces. Examples of these today include graphical user interfaces, data base query commands, networking protocols, and others.
- The true significance of a reference port, beyond providing a tangible representation of the paper standard, is that it is the medium by which the standard operating system can evolve to include the necessary components of the application environment.

- The reference port must reflect the published ABI specification; the ABI specification must, in turn, evolve with the marketplace and include the *de facto* standards of the application environment.

Figure 4 illustrates the fact that the evolution of the ABI can be virtually unbounded as application technologies evolve.

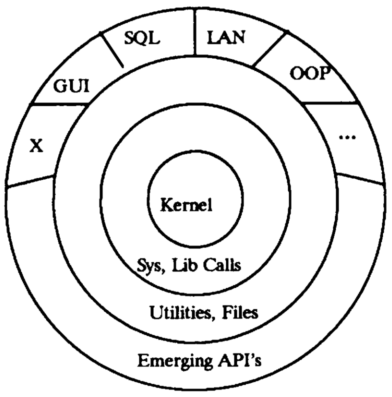


Figure 4. The ABI: The Application Execution Environment

References

AT&T. *System V Application Binary Interface*, 1990.

AT&T. *System V Interface Definition*, Industry Review Draft, Volumes 1-3, 1989.

Intel Corporation. *The Intel 386 Architecture and the System V ABI*, 1990.

Santa Cruz Operation. *SCO MPX*, January, 1990.

UNIX International System Interfaces Working Group. *ABI Creation and Evolution Process*, UI Internal Draft, February, 1990.

Porting Between Open Look™ and OSF/Motif GUIs

Paul E. Kimball
Digital Equipment Corporation
800 W. El Camino Real
Mountain View, CA 94040
(415) 691-4756

Introduction

Cost-reduction is a compelling reason for using UNIX®-based computer systems as opposed to computer systems based on proprietary operating systems. In a paper presented at UniForum™ 1990 (*Converting S/370 Batch Applications to Run Under UNIX,* UniForum 1990 Conference Proceedings, pages 37-46) we found that:

- High-performance UNIX-based computers have superior price/performance characteristics as compared to IBM® System/370™ computers (308X, 3090, etc.) that use a proprietary operating system.
- There are several COBOL compilers available today for use on UNIX-based computers. One of the compilers that was tested was very compatible with IBM's OS/VS COBOL and COBOL II compilers.
- Although file transfer and data conversion between a UNIX-based computer and a System/370 computer is more difficult than it is between UNIX-based computers, it is manageable and can be done on a day-to-day basis.

Continuing to explore the issue of porting System/370 COBOL applications to UNIX-based computers, this paper examines the following issues:

- Compiler Considerations
- CICS emulation
- Embedded SQL
- Use of NFS™ and TCP/IP
- RISC vs. CISC

Compiler Considerations

One COBOL compiler that is very compatible with IBM's System/370 COBOL compilers is the Micro Focus® compiler. The Micro Focus compiler is available on a wide range of hardware, including computers which use the MS-DOS™, OS/2™ and UNIX operating systems. The Micro Focus compiler supports the same numeric formats that the IBM COBOL compilers do.

The computer that we used last year for testing was a Sequent Symmetry® S27 computer. The S27 is a multi-processing computer utilizing two or more Intel® 80386 microprocessors. During last year's test, the Micro Focus COBOL compiler on the S27 could only produce programs which required run-time support. Stand-alone executable programs could not be produced. It was our feeling, based on the use of the Micro Focus compiler on other computers, that stand-alone executable programs would execute more quickly than programs which required run-time support. A compiler that could produce stand-alone executable programs was delivered just prior to the completion of this paper. A cursory study indicated that there was little, if any difference in performance between the stand-alone executable programs and those programs which required run-time support when running compute-bound programs. However, on the S27, when running disk benchmarks, the stand-alone executable code was consistently 10-20% faster than the run-time dependent code. (On another UNIX-based computer that was recently tested, there was no significant difference between stand-alone and run-time versions of the disk benchmark programs.)

Another observation that we made last year was that none of the currently available COBOL compilers could take advantage of the multi-processing architecture of the S27 to reduce the time required to run a single program (multiple programs which execute concurrently are automatically load-balanced by the S27). This situation has not changed since last year. Single programs can take advantage of multi-processing if they are re-written (probably to C). However, re-writing a COBOL application to take advantage of multi-processing today will certainly reduce porting ease, not only between a System/370 computer and a UNIX-based computer but between UNIX-based computers as well because of the lack of multi-processing standards.

CICS

System/370 computers utilize a teleprocessing monitor to manage the interaction between application programs on one hand and terminals, printers and other computing devices on the other. The two teleprocessing monitors most commonly used with IBM's MVS™ operating system are TSO (Time-Sharing Option) and CICS (Customer Information and Control System).

The following are some of the components necessary to build a CICS COBOL application:

- BMS source code files (also called maps). These files describe the layout of the screens used by CICS programs. BMS source code files are compiled and linked into executable programs.
- Copybooks. Copybooks are similar to the .h include files used in the C programming language.
- COBOL programs compiled and linked into executable programs.
- File Control Table (FCT). This table relates dataset names to VSAM file names and key paths. It also defines the location and properties of those files.
- Terminal Control Table (TCT). Older versions of CICS required that any terminal which used CICS be defined via the TCT.
- Program Control Table (PCT). This table relates Transaction Identifiers and AID key values to application program names.
- Processing Program Table (PPT). This table is used to identify the programs that are run under CICS.
- CICS makes use of VSAM (Virtual Storage Access Method) files. VSAM file formats include ESDS (Entry Sequenced Data Sets), KSDS (Keyed Sequence Data Sets) and RRDS (Relative Record Data Sets).

VIS/TP™. VISystems Inc. offers a program called VIS/TP which provides CICS functionality on UNIX-based computers. VISystems claims that VIS/TP is compatible with Release 1.7 of CICS. The current version of CICS (as of October, 1990) is 2.1. VIS/TP requires that the following steps be performed in order to run a CICS COBOL program on a UNIX-based computer:

- The File Control, Terminal Control, Program Control and Processing Program tables (as well as other CICS tables) can be defined to VIS/TP (these tables are interactively created - they are not down-loaded from the System/370 computer).
- Data files that are used must be converted using a VIS/TP utility. In order to use this utility, the corresponding copybooks must be available.
- Using the *vismpgen* utility, BMS maps are compiled and linked into executable modules. During this process, each BMS statement is analyzed (parsed) for validity based on syntax and level of support. Statements failing the analysis are flagged with a message.
- COBOL programs are translated and passed to the C compiler. Linking is then performed. (COBOL programs must conform to the ANSI 85 standard.)
- Development and debugging of CICS COBOL can be done using the VIS/TP package. A debugging tool similar to CDEF is available, although the VIS/TP implementation is not as functionally rich as the actual CICS product. Debugging is

done at the COBOL and CICS source level.

Two applications that were tested using VIS/TP were: 1) the Special Vehicle Handling application and 2) the Broadcast System application. The main purpose of this testing process was to determine what place (if any) such a product would have at Chrysler in the future. There was no intention to immediately port these production applications to a UNIX-based computer.

Special Vehicle Handling. This application is used to handle special vehicle orders (fleet orders, customized vehicles, etc.). This application consists of:

- Six COBOL programs and the associated copy books with 14,750 total lines of source code.
- One large BMS source file (map) with 1200 lines of source code.
- One VSAM data file.

All of the above files were transferred to the Sequent S27 computer using RJE (remote-job entry) as the file-transfer method. The following are our observations in porting this application to the S27:

- The *vismpgen* utility converted the BMS map file without any problems.
- The data file was converted for use under UNIX using the VIS/TP file convert utility without any problem.
- One of the COBOL programs used the variable "DAY-OF-WEEK", a reserved word under COBOL ANSI 85. It was not reserved in previous ANSI COBOL standards. Because of the reverse byte-ordering for words used by Intel microprocessors, a part of one of the COBOL programs that performed a alpha/numeric byte-check had to be changed. (This was a conscious decision on the part of VISystems and not a bug.) Once these two changes were made, the COBOL programs compiled and executed without a problem.
- It took 1-1/2 man-days to port this application, including the time necessary to create the various control tables that were required.

Broadcast System. The Broadcast System is used to create a build order for use at Chrysler's assembly plants. This application consists of:

- Eight COBOL programs (four main programs and four supporting programs) and five IBM System/370 assembler subroutines with 43,680 total lines of source code.
- The copybooks for this application were incomplete. This was unfortunate since one of the copybooks had over 300 different record formats defined in it.
- Six BMS source files (maps) with 1500 total lines of source code
- Five VSAM data files (using both fixed and variable record types)

Once again, RJE was used to perform a file-transfer between the System/370 computer and the S27. The following are our observations in porting this application to the S27:

- The *vismpgen* utility converted the BMS map files without any problems.
- In order to use the VIS/TP file convert utility, the corresponding copybooks are required. In the case of the broadcast system, these copybooks did not exist and additional programs needed to be written so that VIS/TP could use the VSAM data files which were downloaded. In order to work around this, five COBOL programs on the System/370 computer and five COBOL programs on the S27 (all batch) were written.

- One of the COBOL programs used the variable "DAY-OF-WEEK", a reserved word under COBOL ANSI 85. It was not reserved in previous ANSI COBOL standards. The verb TRANSFORM needed to be changed to INSPECT (another COBOL version change).
- There were five System/370 assembler routines. VIS/TP does not support the use of System/370 Assembler code. At the time that this paper was written, not all of these assembler routines had been converted to COBOL.
- This system made extensive use of CICS features. Some of these features were added in recent CICS upgrades. As is the case in similar situations, there are two moving targets. VISystems continues to work on VIS/TP while at the same time IBM is releasing new versions of CICS.

Based on our experience, we believe that the following conditions make it more likely that a CICS COBOL program can be ported using VIS/TP:

- Good system development practices were followed (i.e., complete copybooks exist, COBOL code follows relatively recent COBOL standards, etc.).
- Programs do not use System/370 assembler code.
- A list of the CICS features used is maintained so that it can be compared to the list of supported VIS/TP features.

DB2

One of the issues that was raised during the presentation last year was the porting of COBOL programs which contained embedded SQL statements accessing DB2, IBM's relational database management system. While there are UNIX-based solutions that provide SQL functionality, careful consideration should be given to the available alternatives.

VISystems. Although VISystems provides CICS functionality, the current release of VIS/TP does not yet support SQL.

Micro Focus. Currently, Micro Focus does not support SQL on its products for UNIX-based compilers. The vendors that provide SQL functionality with the Micro Focus MS-DOS compiler (Gupta Technologies and Software Systems Technology) have announced their intention to support UNIX-based computers in the future.

At least two other RDBMS vendors currently provide the capability to embed SQL statements in COBOL programs.

We have not tested any of these products. However, as was the case with our COBOL programs which accessed VSAM files, batch COBOL programs with embedded SQL will be easier to port than those that require CICS. It is also certainly easier to port COBOL programs which use VSAM files as opposed to DB2 files.

NFS and TCP/IP

The main feature of NFS (Network File System) is the ability for a client computer to "import" a file system from a server computer. While the term "import" is used (suggesting a one-time exchange of data), NFS really "virtualizes" file systems so that a file system on the server appears to the client computer like a file system of its own. The exported file system can be made available to multiple clients on a network and multiple servers can be defined on the network. A computer can serve as both a server and a client. If a System/370 computer (running MVS or VM) had the ability to act as an NFS file server, then a UNIX-based computer would be able to transparently access the IBM file systems. Unfortunately, this capability is not available today, mainly because of the very different nature of the file systems involved. However, there are numerous TCP/IP and NFS solutions that permit some degree of inter-operability between UNIX-based computers and System/370 computers.

A typical System/370 computer system uses the following components to communicate to other computing devices:

- The System/370 computer itself.
- Front-end processors which are "channel-attached" (a high-speed direct connection) to the System/370 computer.
- Communications links on the front-end processor to remote computing devices.
- Computing devices of some sort which may or may not serve as a hub for even more computing devices. (See Figure 1.)

Most implementations of TCP/IP on a System/370 computer require TCP/IP software on the System/370 computer. Some implementations (including IBM's) support NFS, however a separate file space is allocated for NFS use. From that point, the following options are available (this list is not exhaustive):

- An IBM 3745 front-end processor can be remotely connected via X.25 to a UNIX-based computer with X.25 and Ethernet™ communication capabilities.
- An IBM 3745 front-end processor can be remotely connected via an SNA network to a device that supports an Ethernet network (Mitek's 2130 is an example of such a device). Because many organizations already have an extensive SNA network, such an option may be less disruptive than the options which require an X.25 communications link.
- A front-end processor can be directly connected to an Ethernet network (IBM's 3172 and 8232 front-end processors are examples of this).

Amdahl offers a Multiple Domain Feature™ (MDF)™ which allows multiple operating system environments to be run on their systems (even single-engine computers). In this environment, it is possible to run UTS™ (Amdahl's implementation of UNIX) in one domain and, for example, MVS/XA in another domain. File transfer between the UNIX domain and the MVS domain take place at channel speeds. Perhaps the most important feature of MDF is the ability for programs running under MVS and UTS to access the same files on UTS utilizing APPC (Advanced Program-to-Program Communication) calls. Multiple domains and Amdahl's TCP/IP software can be used on other System/370 plug-compatible computers.

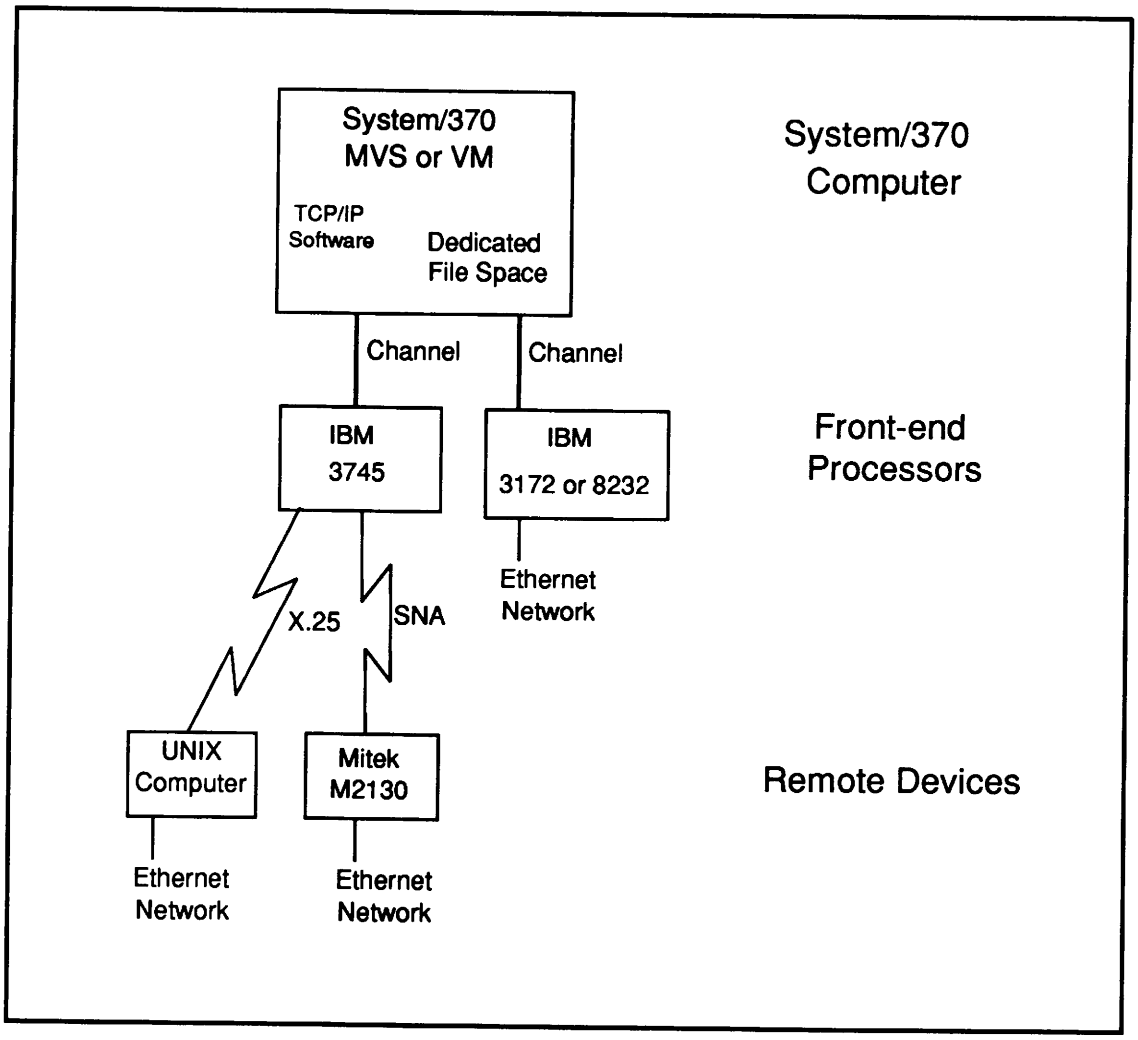
There is a wide range of TCP/IP and NFS products available today for the System/370 environment. These products differ in the functionality they offer, the performance they offer, the amount of System/370 computer resources that are required and their cost.

RISC vs. CISC

In the past 18 months, we have tested UNIX-based computers from ten different vendors. Some of these computers were aimed toward the engineering/scientific market and others were aimed toward the traditional business minicomputer market. Among the computers tested were several RISC computers. In two cases, RISC and CISC computers from the same manufacturer were tested. Based on our benchmarking tests and the use of these computers, we can make the following observations about the RISC vs. CISC issue:

- The RISC-based computers were, as a whole, faster than CISC-based computers when performing single compute-bound tasks. However, as more tasks were added, the RISC-based computers tended to lose their performance advantage. (While we tested computers with recent RISC microprocessors, we have not tested

Figure 1:
Typical System/370 TCP/IP Implementation



computers which use either the Intel 80486 or the Motorola 68040 microprocessors.)

- RISC computers did not show any performance advantages when doing disk-intensive operations.
- Executable program files were larger on RISC computers vs. CISC computers.

How important is the issue of RISC vs. CISC? For our applications (business applications as opposed to scientific or engineering applications), the issue is not very important. Certainly, when selecting a computer to port System/370 applications to, software availability (COBOL compiler, communications software, etc.) requires a higher level of consideration that the RISC vs. CISC issue.

Conclusions

Based on our experience over the past two years in porting IBM System/370 COBOL applications to UNIX-based computers, we suggest that the following steps be taken for a successful porting effort:

- Form a team that will be responsible for porting efforts. The team should have expertise in the following areas:
  - Typical UNIX hardware platforms (Intel 80X86, Motorola 680X0, etc.)
  - System/370 computer operation, including communications, file structure and areas applicable to the programs being ported (CICS, MVS, VM, TSO, DB2, etc.)
  - UNIX
  - COBOL
  - C
  - Multi-processing (helpful)
  - MS-DOS computers (helpful)
- The team should be cross-educated. That is, a person on the team should be knowledgeable about both C and COBOL. A person on the team should be knowledgeable about how UNIX-based computers and System/370 computers interface with terminals. This cross-education is important because the terminology and the technology of the UNIX-based computer environment is very different from that of the System/370 environment.
- Expectations about future porting success should not be based on early porting experiences. There is a learning curve involved in becoming proficient at porting programs.
- Once the team has become proficient at porting IBM System/370 computer programs to a UNIX-based computer, a porting profile should be developed. This dynamic profile should detail the likely costs/benefits of porting certain types of programs (batch vs. non-batch programs, programs which use VSAM files vs. programs which use DB2, etc.). This profile should be used to determine the probable cost/benefit of porting candidate applications.

Nobody today suggests that IBM System/370 applications can be put into a magic "black box" which would turn out the same application on a UNIX-based computer. However, the increasing popularity of UNIX has encouraged vendors to bring products to market which make it possible to port certain types of System/370 applications to run on UNIX-based computers with a relatively low expenditure of effort. This trend is likely to continue and the benefits for those organizations which can take advantage of this trend include lower computing costs, greater flexibility and faster system implementation cycles.

This paper should not be construed as an endorsement of any of the aforementioned products.

Trademarks

Amdahl is a registered trademark of Amdahl Corporation.
Ethernet is a trademark of Xerox Corporation.
Intel is a registered trademark of Intel Corporation.
Micro Focus is a registered trademark of Micro Focus.
MS-DOS is a trademark of Microsoft Corporation
Multiple Domain Feature and MDF are trademarks of Amdahl Corporation
MVS is a trademark of the International Business Machines Corporation
NFS is a trademark of Sun Microsystems, Inc.
OS/2 is a trademark of the International Business Machines Corporation
SPARC is a trademark of Sun Microsystems, Inc.
Sun is a trademark of Sun Microsystems, Inc.
Symmetry is a registered trademark of Sequent Computer Systems, Inc.
UniForum is a trademark of UniForum
UNIX is a registered trademark of AT&T
UTS is a registered trademark of Amdahl Corporation.
VIS/TP is a trademark of VISystems Inc.

Multimedia Document Workstation: Future Data Terminal Equipment

Jonathan Z. Ma
Accurate Information Systems, Inc.
3000 Hadley Road
South Plainfield, NJ 07080
(201) 754-7714
jma@accurate.com

Multimedia Document Workstation
Future Data Terminal Equipment

Jonathan Z. Ma
jma@accurate.com

ACCURATE Information Systems, Inc.
3000 Hadley Road,
South Plainfield, N.J. 07080

ABSTRACT

A multimedia document consists of text, images, graphics and, possibly, speech. In a computer-assisted environment, the ability to compose and to exchange such a document via data communications networks has the potential to significantly increase the effectiveness of communications. The benefit of multimedia in communications is clear; by exchanging information in multimedia, one can not only acquire the information, but also experience it. For example, an animated multimedia presentation of a surgical demonstration would give medical students much a stronger impression than a text book does. Also, a multimedia conference, although it cannot completely relace human face-to-face meetings, will certainly help to reduce the barrier introduced by geographical separation. However, like any emerging technology, multimedia communications capability also introduces a number of technical challenges. This is why, until recently, most of the multimedia document systems have only been available in research laboratories.

This paper discusses the technical challenges in constructing a multimedia document system and how the current technology can be applied to resolve these challenges. It presents the architecture of a multimedia document workstation which integrates all the necessary functional components and provides multimedia communications capability. The paper will also discuss how a multimedia document workstation, which may become a future Data Terminal Equipment (DTE), can be integrated with wideband data communications networks.

1. Introduction

A multimedia document consists of text, images, graphics and, possibly, speech. In a computer-assisted environment, the ability to compose and to exchange such a document via data communications networks has the potential to significantly increase the effectiveness of communications. However, like any emerging technology, multimedia communications capability also introduces a number of technical challenges. This is why, until recently, most of the multimedia document systems have only been available in research laboratories.

Firstly, a multimedia document, by its nature, often contains a large volume of data. Consequently, to exchange such documents requires high bandwidth networks. With the advent of FDDI and ISDN, efficient and economical exchange of multimedia documents becomes feasible.

Secondly, composing and processing a multimedia document often requires higher processing power. Recent development in high performance micro-computers paves the way for commercially viable, cost-effective multimedia workstations.

Thirdly, to construct, store and manipulate a multimedia document requires mass storage and sophisticated input/output devices. Recent progress in optical disk storage, scanning facilities and printing devices provides the necessary ingredients for the construction of a multimedia workstation.

Lastly, perhaps the most challenging task in constructing a multimedia document system is the provision for interchangeability of multimedia documents in heterogeneous environments. Needless to say, the interchangeability of multimedia documents requires all the systems in question to be open systems. The OSI [1] model provides the desirable architecture. However, an open system alone, even if it is in conformance with the OSI standards, does not make the multimedia documents interchangeable. Since physical characteristics of peripheral devices vary widely, information formatted for the originator's devices may appear quite differently on the recipient's devices; sometimes, it may even be illegible. Documents that are to be interchanged should have a certain structure and follow some interchange format, so that when required, they can be easily reprocessed and reformatted on the recipient's machine and properly rendered on the recipient's devices. The rules for defining the structures in documents are collectively called the document architecture model, referred to, by ISO, as the Office Document Architecture (ODA) [2].

This paper discusses the technical challenges in constructing a multimedia document system and how the current technology can be applied to resolve these challenges. The emphasis of the paper is on document processing and integration techniques required in building a multimedia system.

Today, multimedia are still a raw frontier in which tools and skills are mixed freely without having to follow any standard architecture. The complexity of existing multimedia systems varies from a simple CD-ROM player, through a sophisticated authoring system, to an integrated multimedia system [3]. The multimedia system discussed in this paper refers to such a system which is able to compose, store, retrieve, and to represent machine-processable information expressed in multimedia. The multimedia information should be organized in a structure conforming to the Office Document Architecture (ODA) [2]. In addition, the system should be able to exchange such a document which is in compliance with the Office Document Interchange Format (ODIF) [2] in a heterogeneous environment via wideband networks. Hopefully, such a system will become a forthcoming Data Terminal Equipment (DTE) and widely applied in future information and data communications systems.

Prior to discussing the architecture and details of such a system, the terminology and concepts related to the Office Document Architecture need to be described. Section 2 introduces the basic concept of ODA and ODIF. The architecture of such a multimedia system is discussed in Section 3. Two key components of the multimedia system, namely a structure editor and a structure formatter, are discussed in Section 4 and Section 5 respectively. Section 6 is concerned with the presentation of a multimedia document. Storage organization and database techniques used for multimedia document storage and presentation are covered in Section 7. In Section 8, a possible implementation scheme under UNIX® System V (Release 3 or greater) is discussed.

2. Office Document Architecture

An electronic document can be interchanged in an image form or a processable form. An electronic document received in an image form can be directly presented on the recipient's device(s) should the devices are compatible with or identical to the originator's devices. Unfortunately, in real life, this requirement cannot always be satisfied. In general, an electronic document, especially a multimedia document, needs to be presented in a processable form so that when it is exchanged in a heterogeneous environment it can be easily reproduced on the recipient's machine. ODA and ODIF define the architecture and interchangeable format of a machine processable electronic document. According to the ODA standards, the content of a document is physically and logically structured and the structure of the document is reflected by its logical and layout structures. Both the logical and layout structures are hierarchical and together they provide a different, but complementary view of a document.

The logical structure divides the content of a document into a hierarchy of logical objects, e.g. chapters, appendices, headings, paragraphs, footnotes, figures, etc.

The layout structure divides the content of a document into a hierarchy of layout objects for positioning and rendition on presentation media.

A group of similar objects is called an object class. A definition which describes the structure of an object class is called a generic object definition -- generic logical definition or generic layout definition. The structures that are particular to a given document are referred to as specific logical and layout structures. A document may also include a document profile, which is separate from the structure and the content of the document. A document profile contains information for handling the document as a whole.

3. Multimedia System Architecture

The attractiveness of a multimedia system is obvious, however to achieve the spectacular results it promises, it has first to overcome a number of technical challenges summarized below:

- Data capture and data acquisition: a multimedia document may include information from various sources. Conventional input devices, such as keyboards and disks, by themselves are not sufficient. Special data capture devices, such as a scanner, a voice encoder, and a motion video board, are required to capture the information presented in various media and convert it to digitized and formatted data so that it can be processed by a computer.
- Data presentation: to achieve the effectiveness of multimedia communications, multimedia information needs be presented, often simultaneously, to diversified devices.

® UNIX is a registered trademark of AT&T.

For example, to present a multimedia training tutorial, while text, graphic and image are being presented to a graphic monitor, the sound explanation may need, at the same time, to be sent to the attached speaker. Also, during the data presentation process, issues associated with data conversion and synchronization need to be addressed. For example, the output from a computer's VGA adapter is incompatible with the interlaced signal defined by the National Television Systems Committee (NTSC) standard for broadcast television [4]. Consequently, an image capture by a video camera has to be converted before being presented to a VGA monitor. In addition, to present multimedia information comprised of image and voice, a proper synchronization is required.

- Large storage and data compression: audio, image and/or video information when digitized, generates large volumes of data. A scanned image (assume that scanning resolution is 300 dots per inch and data is not compressed) of one A4-size paper requires about 1M bytes of storage. A one minute digitized motion picture, before it is compressed, requires 2 gigabytes storage [5]. Audio data is more compact. A one second quality segment of digitized sound needs about 44k bytes of storage. While data compression can considerably reduce the required storage space, the sheer volume of multimedia data still creates a storage problem. Optical disks, with storage capacity measured in gigabytes, provide required mass storage. An optical disk can be a CD-ROM, a WORM (Write Once, Read Many) or an erasable optical disk. The choice depends on the requirement of applications. Optical disks can be organized in Jukeboxes to provide very large storage. Often large data redundancy is observed in scanned image and digitized audio and video data. An efficient compression algorithm can considerably reduce the volume of data. Data compression/decompression can be conducted on a workstation by software or on an add-in board by firmware. Perhaps the most efficient way to implement data compression/decompression functions, is to incorporate these functions in firmware and free the main processor(s) to handle other computing tasks.
- Structure construction and presentation: the logical and layout structures of a multimedia document and the temporal relation between different media needs to be specified. Thus, at the receiving end the multimedia information can be properly reconstructed and rendered to appropriate devices.
- Efficient retrieval: in general, digitized audio and video data requires large volumes of space and often is not formatted. This kind of information is not appropriate for a database. The information can be stored on a separate storage device, e.g. an optical disk, instead of the same device where the database resides. However, efficient and versatile indexes should be built on the database, so that the media dependent information can be efficiently and easily retrieved.
- High communication bandwidth: exchanging a multimedia document requires high communication bandwidth and appropriate communication protocols so that adequate communication performance can be achieved.

To meet the above mentioned challenges, a multimedia system should be comprised of the following functional components:

— Special I/O devices.

— Structure editor.

— Structure formatter.

— Structure presenter.

— Database management system.

— Efficient data communications sub-system.

However, unless these functional components are integrated together and presented via a friendly user interface, multimedia systems will only be in the hands of specialists.

The architecture presented below provides a platform whereby all the functional components can be integrated together. As depicted in Figure 1, the proposed system will consist of the following functional modules:

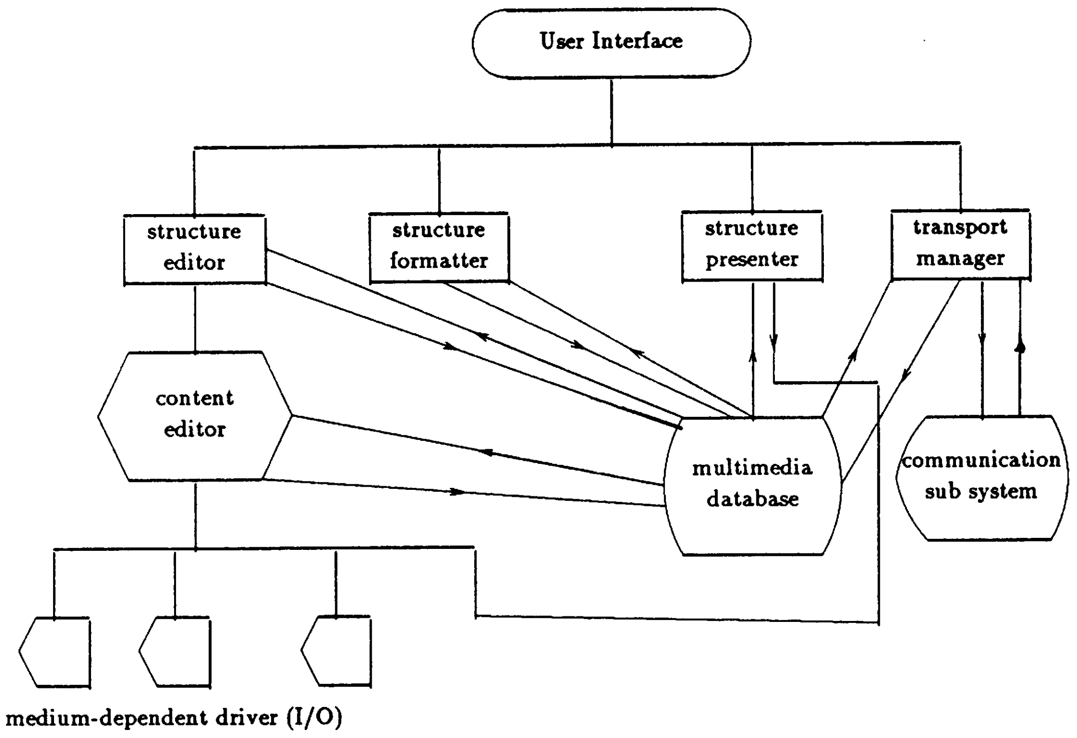


Figure 1. Architecture of a Multimedia System.

1. User Interface: this module is the user's access point to the system. From here, a user can either invoke the structure editor to compose a multimedia document, the structure formatter to format a pre-composed document, the structure presenter to interpret the formatted structure of the multimedia document and to present the document on suitable media, or the transport manager to exchange a multimedia document with a remote partner. The user interface module should be a menu-driven interface with pop-up and pull-down menu capability to allow a user to solicit any of the underlying functional modules. Function keys and icons should also be provided to assist access of the system by non-technical users.

2. Structure Editor: this module is responsible for composing a multimedia document. This module takes input from the user and the database, where generic logical structures are saved, produces a composed multimedia document. When needed, the structure editor will also call the content editor to create and allocate a content

portion for a chosen basic logical object. In creating a content portion, a special I/O device may need to be called.

3. Formatter: this module is responsible for producing the specific layout structure of a composed document.
4. Structure Presenter: this module helps a user view a composed multimedia document.
5. Multimedia database: the database system which stores and maintains multimedia documents and associated information such as document profiles, generic logical structures and generic layout structures. Indexes to contents may also be saved in the database, although the contents themselves may be stored on a separate storage device.
6. Transport manager: this module is responsible for exchanging a multimedia document with a remote system -- a host or a database server.

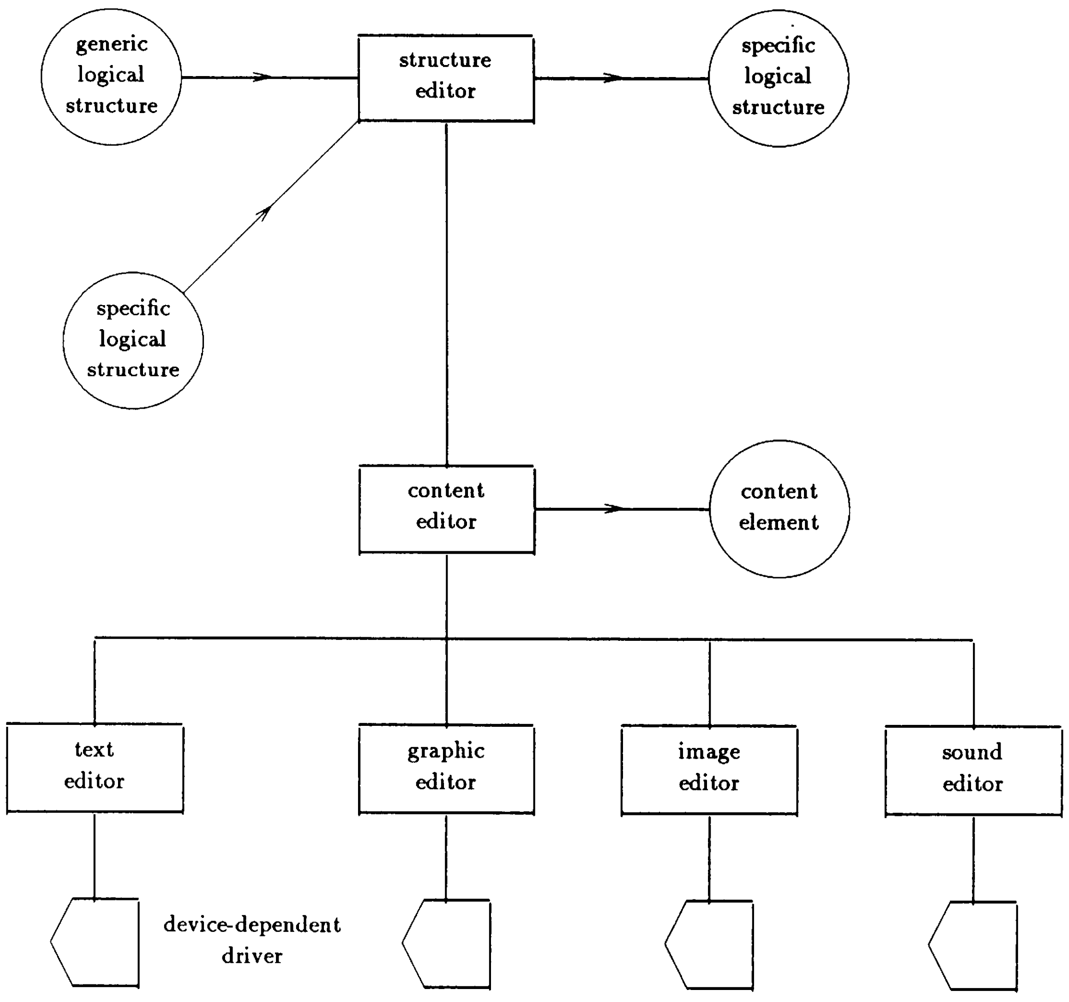


Figure 2. Interaction between Structure Editor and Content Editor.

4. Structure Editor

The process of editing a multimedia document includes document creation and document revision. From an architecture perspective, these activities are indistinguishable. The document editing process consists of the logical structure editing process and the content editing process. The logical structure editing process is concerned with the creation of a specific logical structure and the modification of a previously created logical structure. The content editing process deals with the creation of a new content element or the modification of an existing content element. The interaction between these two processes is depicted in Figure 2.

During the logical structure editing process, the inputs fed to the structure editor are generic and, possibly, specific logical structures. The generic logical structure is used as a template to guide the creation of a specific logical structure. The generic logical structure for a memorandum might be the one illustrated in Figure 3. A windowing system will make this editing process much easier. For example, while one window is used to display the generic logical structure, another window can be employed to create the specific logical structure. During this process, window oriented utilities such as cut and paste, can be utilized to speed up the editing.

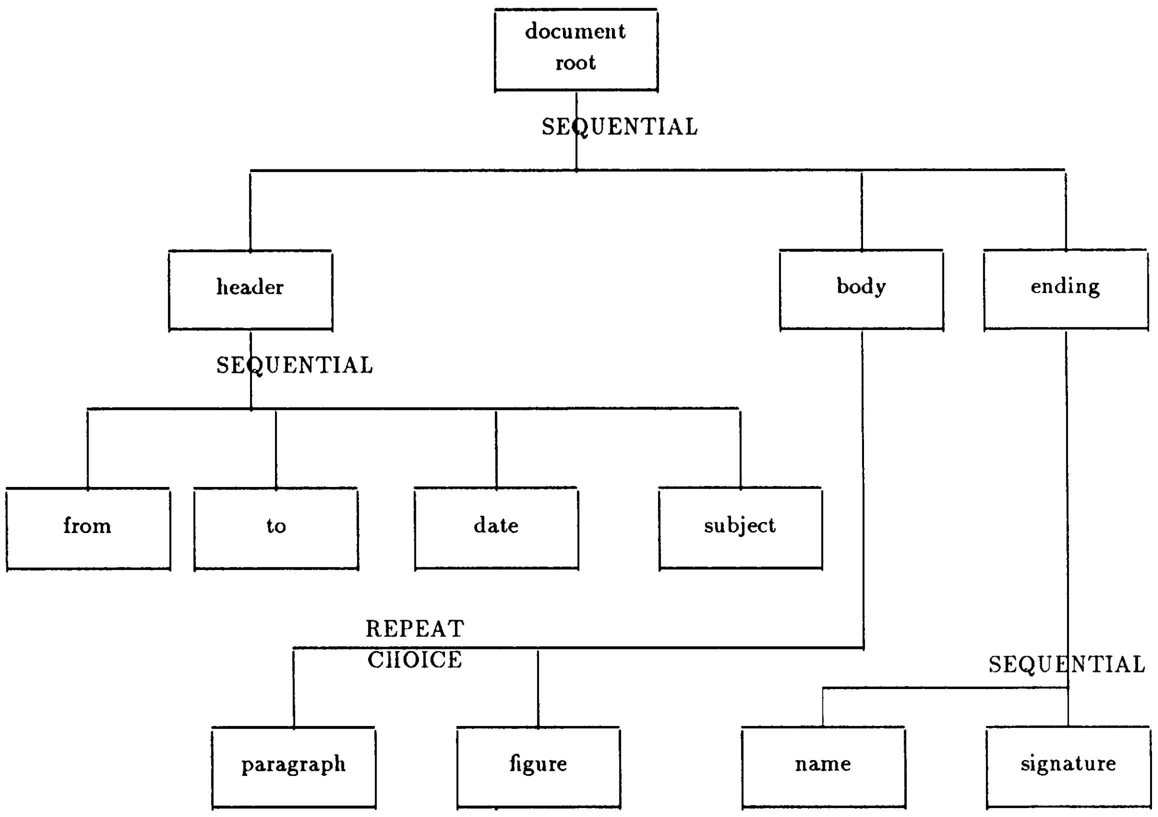


Figure 3. A Generic Logical Structure for Memorandum.

Once the specific logical structure is created, the content editing process can be started to allocate content elements for each of the basic logical objects. Again, if an appropriate windowing system is employed, the user can simply move the cursor to a content element and click a mouse button to activate the content editor.

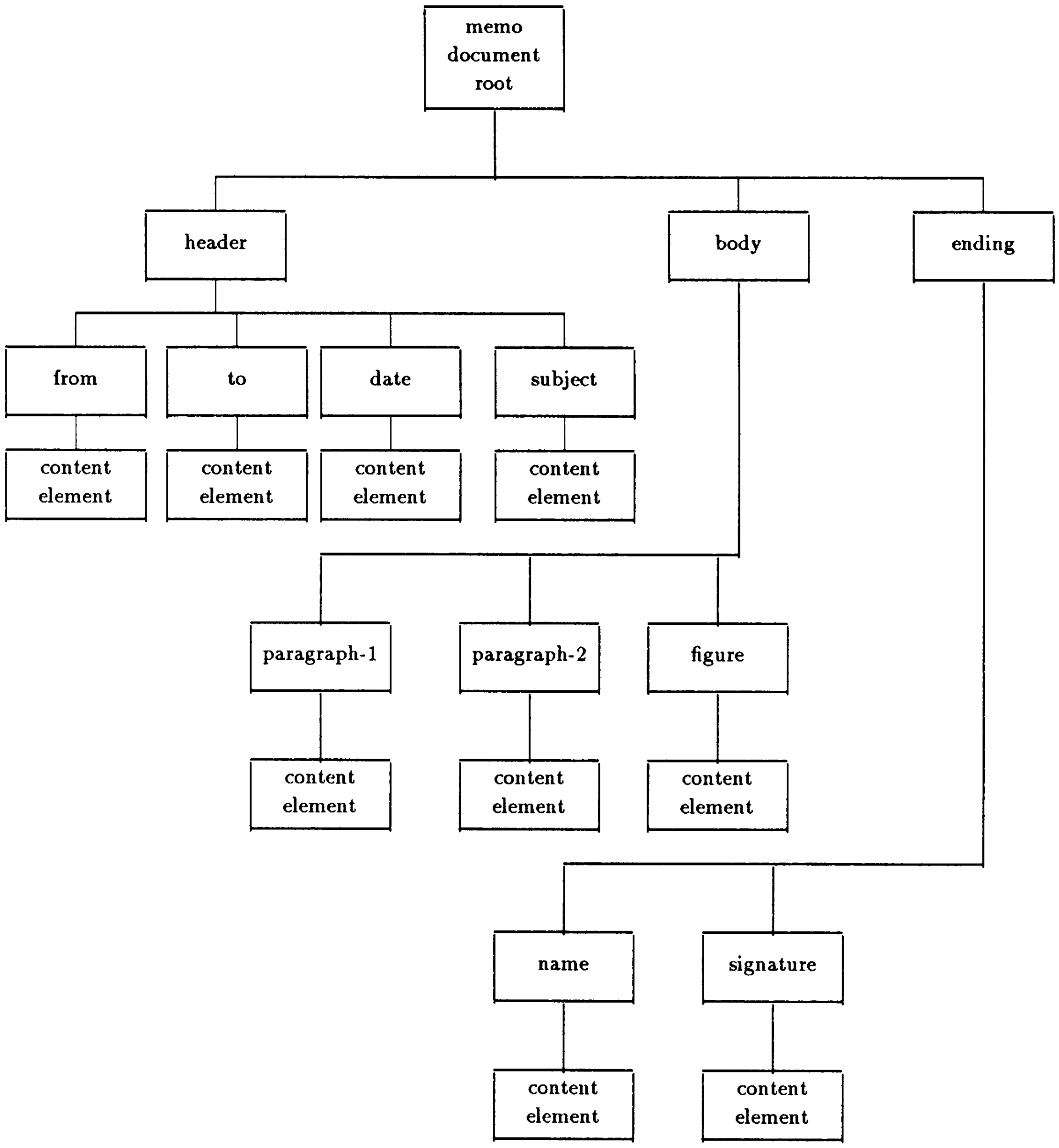


Figure 4. A Specific Logical Structure for a Memo.

From the content editor, the user can call any medium-dependent editor. For example the user may invoke the text editor to create the first paragraph in the memo body. He may invoke the graphic editor to create the illustrating figure and the image editor to pull out the scanned image of his signature. The final specific logical structure for a memo is illustrated in Figure 4.

The output of the structure editor -- specific logical structure and the corresponding content portions -- can either be fed to the structure formatter for further processing, saved in a database, or sent to a remote machine.

5. Structure Formatter

The structure formatter is called to format the document during the layout process. The layout process includes the document layout process and the content layout process. These processes are concerned with the creation of a specific layout structure which can be used by the imaging process to present the document in human perceptible form on the presentation media. The specific layout structure generated by the structure formatter should be consistent with the corresponding generic layout structure. The specific layout structure should also be in line with the layout directive attributes specified in the corresponding logical objects. During the document and content layout processes, the structure formatter will create layout objects into which the content of the sequence of basic logical objects is to be laid out. The layout objects follow a hierarchical structure. For example, a specific layout structure may form a tree structure, which consists of a layout root, page sets, frame sets, and blocks. The leaf nodes of the tree are blocks -- basic layout objects, and attached to each block is a content element.

The document layout process controls the allocation of the areas within a frame or sequence of frames into which the content of each basic logical object is to be placed. It will also defines constraints on the area(s) into which the content may be laid out. The content layout process is responsible for formatting the content into the allocated area taking into account the constraints imposed by the document layout process. The content layout process determines the dimensions of the basic layout objects. The document layout process is responsible for determining the position of these basic layout objects within the frame which they belong to. The document layout process is also responsible for determining the dimensions and positions of frames. The structure formatter should provide all the necessary utilities in assisting a user to complete the document and content layout processes. The output of the structure formatter -- specific layout structure and laid-out content -- can be sent to the structure presenter or to a remote machine for presentation.

6. Structure Presenter

The responsibility of the structure presenter is to interpret the specific logical and layout structure and present the document on appropriate media. During this process, data conversion and filtering may be carried out to convert the layout directives to a pertinent device-dependent control language understood by the device driver being used. For example, if the document is to be presented to a PostScript printer, the layout directives should be converted to PostScript commands.

ODA defines the logical and layout relationship between document objects. The only temporal relationship specified by ODA is the sequential order of document objects. The sequential order defined by ODA is such that each object in the structure is succeeded by all of its immediate subordinates, before any other objects with the same immediate superior. Each of the immediate subordinates is followed by all of its immediate subordinates, before proceeding to the next immediate subordinate in sequence. ODA does allow the imaging order to determine the precedence of layout objects for imaging. Thus, the imaging order can be different from the sequential layout order. However, the imaging order is still sequential. This over simplied temporal relationship is not always adequate to delineate the temporal relationship of document objects in a complicated multimedia document. Often, two content elements need to be simultaneously presented. For example, motion video and sound explanation need to be synchronized and text and figures need to be accompanied by corresponding voice annotation. The coordination of such a presentation is essential for its understanding. The required synchronization should be built within the document structure. Two new attributes, a temporal_relationship and a logical_ID_list are recommended. The temporal_relationship attribute specifies the temporal relationship between the logical object in question and other related logical objects. The value of this field can be 'sequential', 'independent' or 'simultaneous'. When the value is "simultaneous", the logical_ID_list field should contain all the IDs of the logical objects which are to be presented simultaneously. During the presentation process, the structure presenter should examine the temporal relationship attributes and determine when to present a content element.

7. Storage Organization and Database Management

A multimedia document consists of information in various media. During document presentation, manipulation, and processing, medium-dependent information needs to be retrieved, processed and updated. Consequently, the storage organization has a direct impact on the performance of document processing and its presentation. Since a multimedia document can be very large, it is not practical to store the entire document in memory. Hierarchical storage is required. Fast retrieval memory should be used to store the most recently accessed portion of the document. Hard disks should be used to save the frequently accessed documents, and optical disks, which are slower but have larger capacity, should be used for long term storage and archival. The contents of a multimedia document may be scattered among different storage devices, the structures of the documents and indexes to the contents should always be stored and managed by the database system. A possible database user schema is depicted in Figure 5.

According to this user schema, a user can retrieve a document by key words, a author name, a title or the document creation date. From these indexes the user can get an ID -- the unique identifier to the document. Using the ID as a primary key, the user can then retrieve the logical/layout objects and their associated content elements. The database server may be physically separated from its clients, such as a structure editor or a structure presenter.

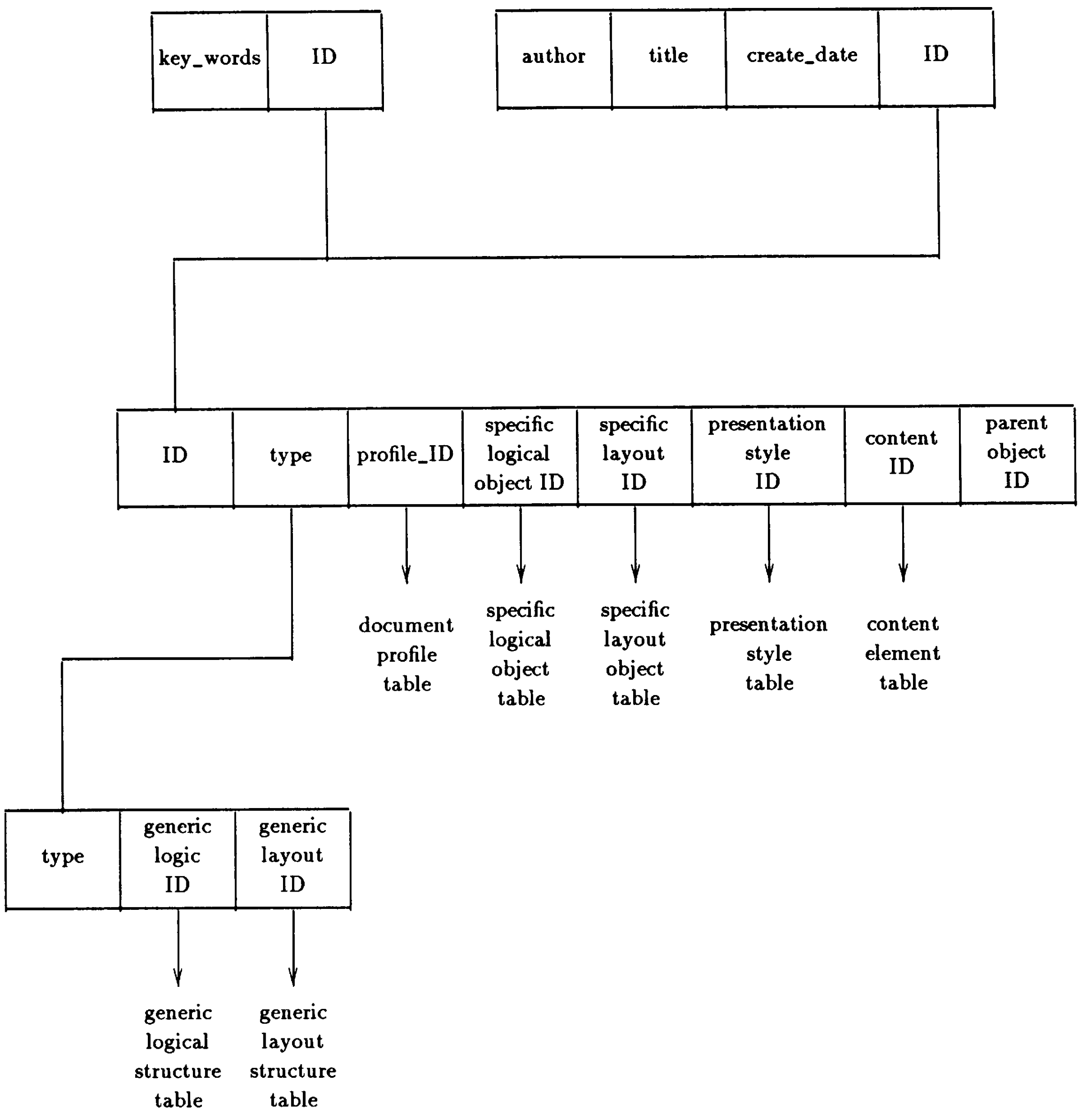


Figure 5. A Database User Schema.

8. An Implementation Scheme

A multimedia system can be implemented on a micro-computer, a workstation, or mini-computer running under various operating systems. However, the UNIX operating system is always a good platform for building a multimedia system due to its sophistication, flexibility and versatility. A possible implementation scheme for a multimedia system implemented under the UNIX operating system (System V Release 3 or greater) using the STREAM [6] mechanism is presented in Figure 6.

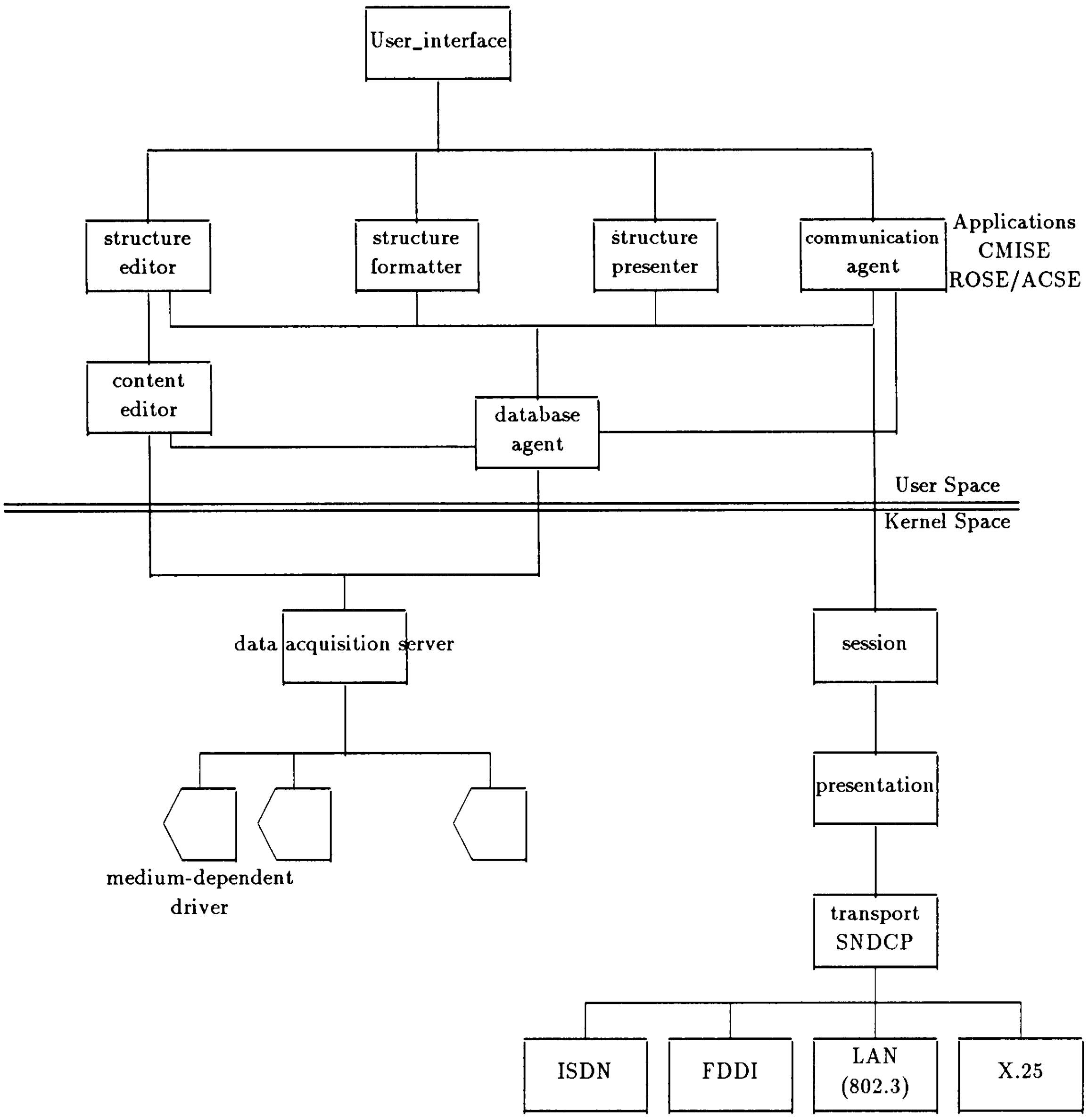


Figure 6. A UNIX Implementation Scheme for a Multimedia System

Here the implementation is split between the user space and the kernel space. Running in the user space will be the user interface, structure editor, structure formatter, structure presenter, content editor, database agent and communication agent modules. Running in the kernel space will be the communication stack, the data acquisition server and medium-dependent drivers. The user interface module is responsible for interacting with a user and assisting the user in obtaining the services provided by the structure editor, the structure formatter, the structure presenter and/or the communication agent. The database agent module accepts the requests initiated by structure editor, content editor, structure presenter and/or structure formatter modules for data retrieval, deposit or updates.

The database agent module will first decide whether the information in question is stored locally or remotely. If the information is saved locally, the database will interact with the data acquisition server and retrieve/save the information on the appropriate medium. Thus the database agent will perform all the functions required by a database management system such as data integrity and data consistency checks. If the information is stored remotely, the database agent will act as a client to the remote database server. The interaction with the remote database server is achieved via the communication agent module. The communication agent module running in the user space provides the required application services and application service elements, such as the Common Management Information Service Element (CMISE) [7], the Remote Operations Service Element (ROSE) and the Association Control Service Element (ACSE) [8]. OSI upper layers [8], i.e. session, presentation, transport layers and SubNet Dependent Convergence Protocol (SNDCP), and lower layers, namely network dependent communication protocols, will run in the kernel space as STREAM drivers and modules.

Medium-dependent drivers will also be implemented in the kernel space as STREAM drivers and modules. The strengths of the STREAM lie in its capability for maintaining adequate real-time response and, at the same time, providing desirable flexibility. Both data communications and medium dependent devices require real-time response. Consequently, the software that handles real-time signals needs to remain in the kernel. On the other hand, the specific system configurations may vary from installation to installation. Using the STREAM mechanism, application developer(s) can easily reconfigure the kernel to accommodate the requirement of a particular system. For example, when ISDN becomes available, one can simply add an appropriate STREAM driver to handle the corresponding lower communications protocols without impacting the upper layer implementation.

9. Conclusions

The attractiveness of a multimedia system is obvious. It can significantly improve the effectiveness of communications. A multimedia system requires a wide spectrum of high technology. Special data acquisition devices, large storage systems, sophisticated database systems and efficient and high performance data communications systems are the necessary ingredients. Recent development in these technical fields have made the technology becoming cheaper, more common and mature. Although multimedia systems rely on the technology of their components, the integration of components into an integrated system is more important and just as challenging as the technology employed in the components. Perhaps the most challenging task in constructing a multimedia document system, is the provision for interchangeability of multimedia documents in a heterogeneous environment. To achieve this goal, a number of new document processing facilities, namely a structure editor, a structure formatter and a structure presenter, are required. The integration of these new document processing facilities with other functional components is essential to success. Clearly, a friendly user interface is another important measure to liberate multimedia document systems from the hands of specialists and to pave the way to much wider application. Hopefully, when all the required technology and facilities are in place, multimedia systems, as future Data

Terminal Equipments on tomorrow's data communication networks will become just as popular and pervasive as telephones on today's telecommunication networks.

REFERENCES

1. ISO 7498, 'Information Processing Systems -- Open System Interconnection -- Basic Reference Model' (1984).

2. ISO 8613, 'Information Processing Systems -- Text and Office Systems -- Office Document Architecture (ODA) and Interchange Format' (1989).

3. 'Multimedia: the Next Frontier for Business?', Robin Raskin, PC Magazine, July, 1990.

4. 'Putting Your PC on Tape', Lori Grunin, PC Magazine, July, 1990.

5. 'Document Imaging System: Technology Overview', Datapro Reports on Document Imaging Systems, June, 1990.

6. 'UNIX System V STREAM Programmer's Guide', AT&T, 1986.

7. ISO 9595, 'Information Processing System -- Open System Interconnection -- Common Management Information Service Definition', Sept. 1989.

8. 'Data Communication Networks Open Systems Interconnection (OSI) Protocol Specifications, Conformance Testing', Recommendations X.220 - X.290, CCITT, Nov. 1988.

Selecting a Graphical User Interface Strategy for Maximum Portability

Paul Shearer
Tektronix, Inc.
P.O. Box 1000 MS 60-850
26600 SW Parkway
Wilsonville, OR 97070
(503) 685-2137
paulsh@orca.wv.tek.com

Selecting a Graphical User Interface Strategy for Maximum Portability

Paul Shearer

Interactive Technologies Division
Tektronix Inc.

ABSTRACT

This paper discusses strategies which facilitate porting software applications between APIs and GUIs. The use of standards is recommended and the strengths of the toolkit paradigm which partitions code into user interface components (widgets) and application callback routines is emphasized as a means of separating user interface code from application code, which in turn improves portability.

The author describes the porting effort involving a modern application with a highly interactive graphical user interface (TekColor Editor) which has been ported to two X toolkits widget sets (Athena and Motif) and to an X terminal without toolkit support. The tasks involved in the latter port are outlined and it is shown that the main interface layers between the application and the toolkit were reimplemented in an attempt to minimize changes to the applications custom widgets and callback routines. This porting strategy proved successful in reusing application code and reducing the size of the GUI layer, which was the main requirement for the target Tektronix TekXpress X Terminal. The total size of the toolkit based application is over 1 MB (1089 KB) whereas the ported X terminal version is under 200 KB.

Introduction

This paper identifies a strategy for maximizing the portability of software implemented with graphics languages or graphical user interfaces (GUI), primarily those implemented using the X Window System. Advantages of software portability go beyond economic justifications. Requiring software to be portable has always had the effect of making that software better designed, better documented, and more thoroughly tested, all of which add to the software's reliability and maintainability. But recently porting software to GUI environments has also increased user productivity.

A consistent, easy-to-learn GUI provides a more productive environment for computer users. A recent study sponsored by Zenith Data Systems and Microsoft Corp. showed that microcomputer users who work with a GUI versus character user interfaces (CUI) are less frustrated and tired, and take advantage of more software features. The experienced GUI users finish 58% more correct work and novice GUI users finish 48% more correct work than CUI counterparts [5]. For these reasons software developers need to be prepared to port applications to the preferred GUIs of their users.

Nineteen years ago the IFIP Working Conference on Graphic Languages had a panel discussion entitled "Are we anywhere near a universal graphic language?" "The answer is no," was the opening reply by graphics expert Andries van Dam [1]. A lot has changed in the last nineteen years, although many experts would argue that the answer is still "no." Today there are far more choices available to graphics application programmers than there were nineteen years ago. However, those choices do include many standards and methods that increase our ability to write portable applications.

Before describing these choices, it is appropriate to define portability. Historically the term portability described the ability to transport software to different machines and/or operating systems. Today portability involves a broader spectrum of issues. The application programmer now must also decide what GUI environment the software will be ported to, what application programming interface (API) will be used, and how the application will communicate with other applications in that GUI environment. In this paper I will discuss the characteristics of portable software which minimize design changes and maximize the reuse of code when moving from one API or GUI to another.

GUI and API Defined

The graphical user interface (GUI) is what a computer user sees and interacts with while using a software application. From the user's point of view, the GUI is defined in terms of look and feel (appearance and behavior).

The GUI software functional specifications, written from the user's perspective, define these GUIs and provide a standard for the application programmer to follow. Examples are the OSF/Motif Style Guide [2] and the OPEN LOOK Graphical User Interface Functional Specification [3].

The application programming interface (API) defines the actual function calls and data structures the programmer must incorporate into the application to implement the user interface. Selecting a specific GUI does not necessarily restrict the application programmer to one API. Examples are the three toolkits, SunView2, At+, and NDE, each of which provides a different API to the OPENLOOK GUI [4].

Designing an application to be portable between different GUIs introduces new issues for programmers, some of which are: GUI Policy, Level of API, and GUI Certification.

The application programmer must decide if the application is to retain its own user interface look and feel or if it is to adopt the same style as the new GUI. With few exceptions, the choice will usually be to adopt the new GUI style. If the choice is to retain a single look and feel, the programmer has the choice of porting the application's internal graphics layer to a lower level API. However, to adopt the new GUI, the programmer needs to port to a higher level API (usually a toolkit), one that implements the look and feel of the GUI. This has changed the nature of the porting effort.

Finally, if the programmer wants to be able to use the GUI trademark in reference to the ported application, the application must receive GUI certification. This is a process that verifies the application as having the required feature set and following required interaction policies of the GUI. Currently both OSF/Motif and OPEN LOOK have such a certification procedure [6][7].

Internal vs External Control

Writing portable GUI software will almost always require the application to be designed using an external control model. This section describes the difference between internal and external control models. See figure 1 and 2. The next section will discuss Toolkit APIs that require the external control model.

Traditional applications utilizing a custom GUI have done so by defining a GUI library that was invoked internally throughout the application code. As graphic API standards were developed (Core and GKS in the late 70's and early 80's, and PHIGS+ and the X Window System Xlib in the late 80's), porting these GUIs meant porting them to whatever graphics library was supported by the target platform.

These early GUIs used an internal control model. Internal control from a user's perspective means that the application dictates the sequence of interaction. The user is prompted for input and has no external control of the order that input can be entered. Internal control from the application programmer's point of view means that calls to graphics input routines are made internally from within many application procedures. A single software procedure or function may have several calls to graphic input routines. This same function would have several blocks of code that define application actions for this required sequence of input events.

Current second generation GUIs tend to follow an external control model. External control gives the user control of the sequence of interaction. The application programmer writes small procedures (callback routines) which define the application action for a single event. The GUI toolkit or User Interface Management System (UIMS) software provides the main loop used to input and dispatch graphics events externally to the application code.

Layers of Choices

One of the toughest questions facing an application programmer is "What API should the application programmer use to maximize portability?" The answer will depend on how extensive the porting goals are (different window systems?), the customer requirements (different GUIs?), and the application requirements (graphics API or window system requirements). The subsections that follow group the API choices into layers according to the level of functionality or portability they provide. See figure 3.

Vendor-Specific Graphics

All applications using a custom graphics library can port to a vendor's machine by porting the custom graphics library on top of the vendor's supplied graphics library.

The biggest advantage at this layer is performance. Also the porting effort is isolated to the drivers in the custom graphics library.

A disadvantage is the cost of writing new drivers for a vendor specific API. Also the scope of the port is limited to the vendor's family of machines.

Examples at this layer are kernel-level windowing systems like SunViews and high performance vendor-specific graphics packages like the Tektronix OnRamp Graphics Library. Extensions to client-server windowing systems should also be considered in this layer. Until an extension is widely accepted, it may only be available on certain vendor's servers.

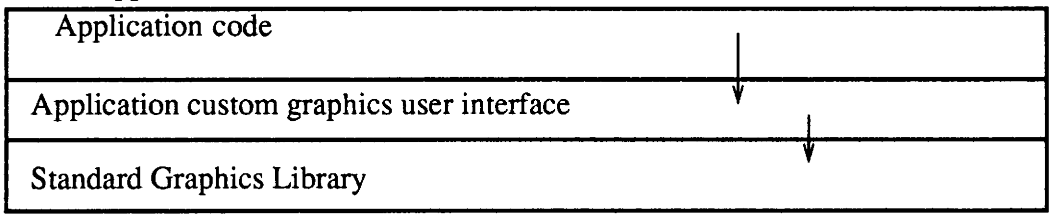
Window System API (Client/Server Protocol)

This layer identifies the lowest level API that an application programmer can interface to a window system.

An advantage at this layer is portability. The application will run on any vendor machine supporting that window system API (protocol). For client-server window systems like the X

Figure 1. Graphics API architecture.

a) Traditional application with internal control user interface


Application code
Application custom graphics user interface
Standard Graphics Library

b) Application with external control designed to be ported to different GUI toolkits

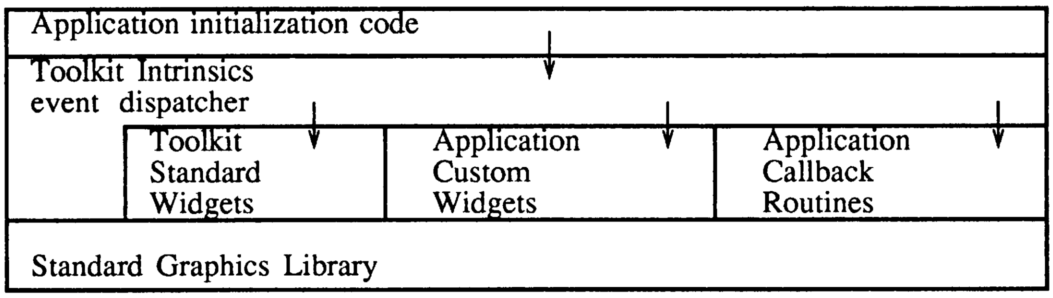
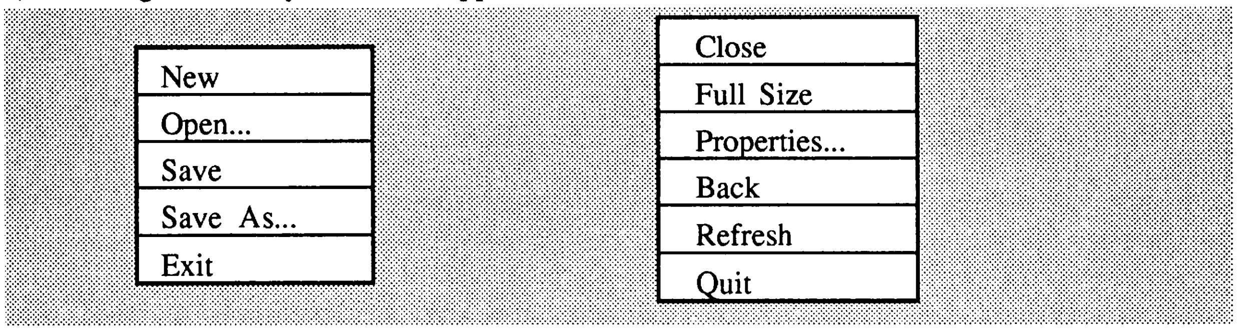

Application initialization code
Toolkit Intrinsics event dispatcher
Toolkit Standard Widgets
Application Custom Widgets
Application Callback Routines
Standard Graphics Library

Figure 2. Menus resulting from porting code to a graphics library vs. a toolkit.

a) Menus generated by traditional application interface, different GUI environments



b) Menus generated by application using different GUI toolkit widget sets.

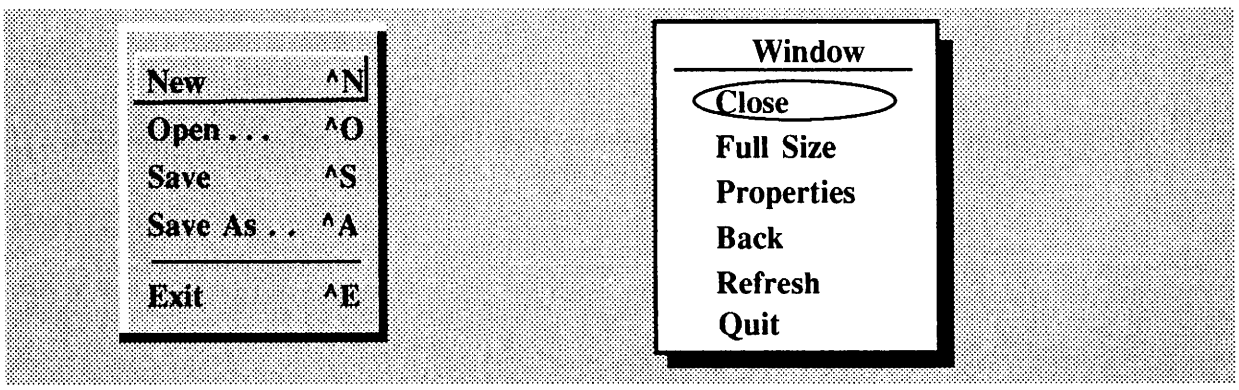
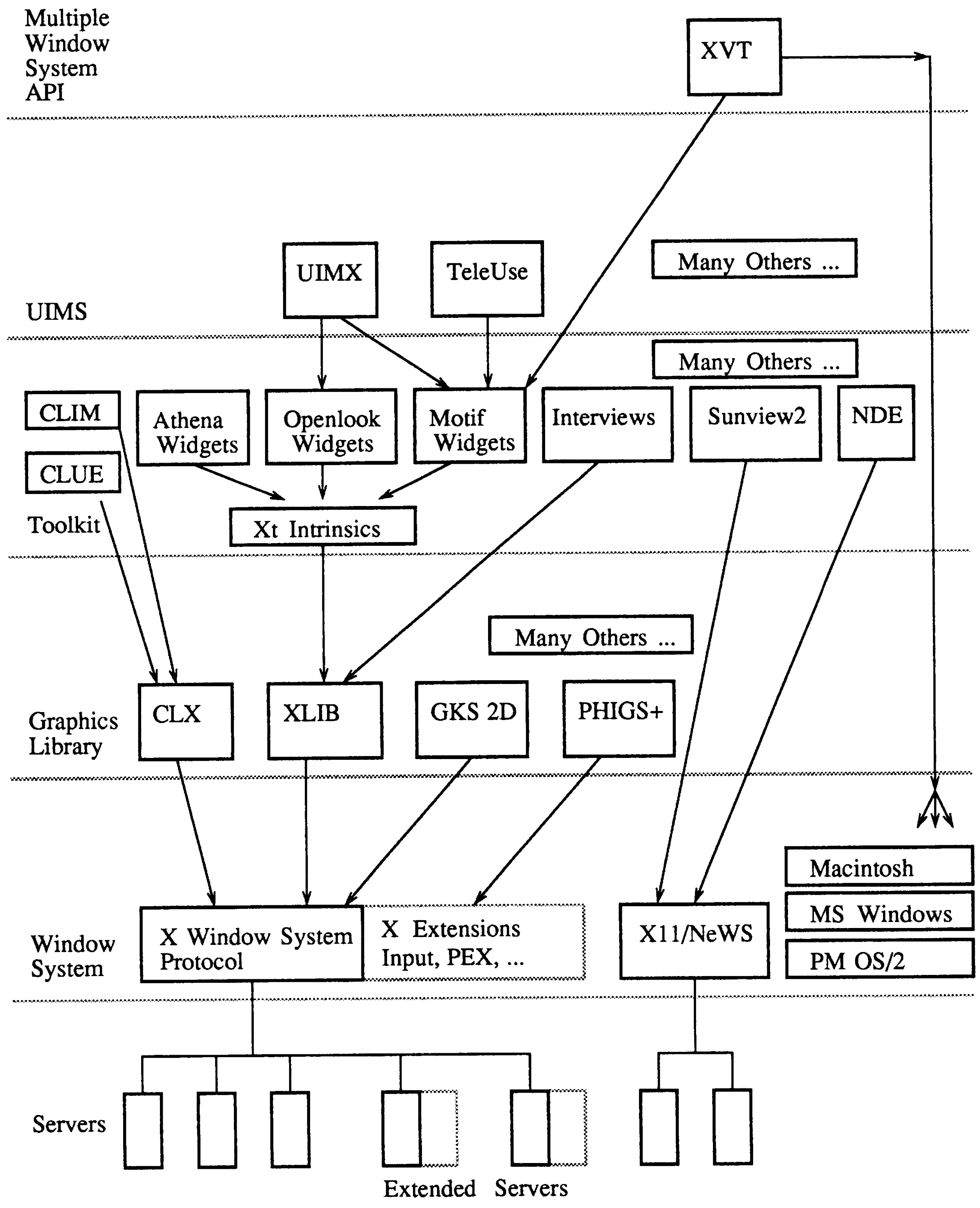


Figure 3. Examples of Different Graphics API Layers.



Window System, this means that the application can reside on only one vendor's machine and be displayed on any other vendor machine supporting the protocol.

Disadvantages are that applications interfacing to this layer will always have a custom look and feel rather than adopting the policies of the GUI environment they are used in. Since applications interfacing to this layer might use a traditional "internal control" design, with multiple calls to graphics input routines scattered throughout the application code, they may not be easy to port to higher GUI layers that use an external control design where the main loop that dispatches events resides outside the application code.

An example of APIs at the window system layer is Xlib, the C language API to the X Window System Protocol. For the purposes of this paper, I will also group APIs such as PHIGS+, GKS, and the supporting CGI standard in this layer.

GUI Toolkits

The next higher layer of API provides the programmer with an interface which adheres to the look and feel of a particular GUI. This layer includes the popular toolkits that have built on top of the lower level window system API. Some common components of these toolkits are [4]:

- Prebuilt graphical user interface components called widgets or controls.
- API that allows programmer to combine prebuilt widgets or create custom widgets.
- API that allows programmer to interface application code (callbacks) to the user interface built from the toolkit.
- Other communication services that interface with the window system or other cooperating applications using the same GUI.

One of the most important toolkit features from the programmer's perspective is that these toolkits provide an external control framework into which the programmer integrates the application code. This external control is provided by a main loop that receives events and dispatches these events to prebuilt components or to application callback routines. Thus the toolkit API discourages the application programmer from attempting to read and process events in a predefined sequence internally within the application code. To adhere to the external control model, the application code needs to be partitioned into callback routines that usually handle only one input event at a time. The application's custom widgets need to be written so they can be displayed and manipulated by the same toolkit API framework that interfaces to the toolkit supplied widgets.

Advantages of this layer should not be underestimated. An application written with the toolkit API functions with a specific GUI look and feel. At the widget level, this look and feel is free, since the GUI that the user sees is almost totally built from toolkit-supplied widgets. The application programmer only has to supply the callback routines that define the application actions resulting from the user's input. The application custom widgets and callback routines can be easily ported to other toolkits at this level that use the same external control design. For example, experience has shown that porting custom widgets and callbacks to toolkits based on the Xt Intrinsics (Athena Widgets and Motif Widgets) is fairly straightforward and much easier than attempting to port an application using the Xlib API without an external control design to an API at the toolkit level.

Other advantages include the added value toolkits supply such as the X11 R5 Xt resource loading conventions which will allow localization of messages. This will simplify the porting task of supplying different messages for different languages [8][9]. Toolkits with attribute-value APIs also allow a simpler interface, since the toolkit provides default values for attributes not specified by the programmer.

A disadvantage of the toolkit APIs is the redesign needed to port an internal control application based on a low level graphics API. Applications that are designed around the external control model may also need to keep track of more state information than applications that require internal sequencing of user input.

Examples of Toolkit APIs are Motif, Athena Widgets, and OPENLOOK Xt+, all based on the X Widow System Xt Intrinsic layer. Others include Interviews, Andrew Toolkit XTk, and OPENLOOK APIs SunView2 and NDE [4].

Multiple GUIs, Same Window System

The next level of API choices include those products that provide value-added features on top of the toolkits. These often include UIMS features like interactive WYSIWYG tools for building graphical user interfaces from toolkit-supplied components, high level descriptive languages used to define the interface, and libraries of convenience routines that provide still more functionality.

Advantages at this level are ease of use and fast prototyping. UIMS tools can also provide application templates (style guide prototypes) that adhere to the style of the supported GUI. One of the most important features of this level is the potential to provide a common API to more than one GUI. A user should not expect this goal to be 100% achievable for sophisticated applications. However, code at this level could have a common/specific split of about 80%/20%, depending on how many of the GUI specific features were being called from within the code.

A disadvantage is the necessity to learn yet another tool or user interface descriptive language. The APIs or output of these products will not be very portable between products at this level. As mentioned above, GUI-specific features must still be coded in a non-portable way. Although there are no current UIMS standards, the UIMS Working Group of OSF/SIGUEC (Special Interest Group User Environment Component) is addressing common interchange formats and other UIMS issues. However the fact that the programmer often supplies or can produce C code using these tools means they can utilize standards established at the lower levels.

Examples of UIMS products at this level include: UIMX, Guide, AutoCode, TeleUse, FaceMaker, NextStep, X-Pression, and Widget Creation Library. For a discussion of many of these products, refer to the May 1990 issue of UnixWorld [10].

Multiple GUIs, Different Window Systems

Finally, the most general API layer supports multiple GUIs and multiple window systems.

The obvious advantage of this API would be to support applications that are targeted for markets with different window systems. The portions of the application that can use this general API are completely portable to all the supported environments.

The disadvantage of such APIs is that they are forced to offer the lowest common denominator of functionality among the GUIs or window systems supported. This means that an application wanting to take advantage of a GUI or window system-specific feature (like hierarchical menus or event types) can't do so using the API provided, but must use non-portable hooks into the specific GUI or window system.

XVT is the best example of this layer of API. The XVT API can be used to generate applications that run on Macintosh, MS-Windows, OS/2 Presentation Manager, X Window System (Motif), or any character screen display. XVT is currently being considered as a proposed standard by the IEEE P1201.1 Working Group. The group had previously been working on standardizing the "N3 Proposal" from AT&T, which provided a common API to OSF/Motif and OPEN LOOK. However, the proposal was dropped when the API requirements expanded to include non-X

Window Systems.

Porting Software Between or Within Layers

When selecting a strategy for maximizing portability, it is important to know what API layers and which APIs or GUI within those layers the software will be ported to. Desired options may be to port across a layer, for example, to stay within the toolkit layer and port to different widget sets, such as Motif and OPENLOOK. Another option may be to port up to the next higher layer, having coded an application at the toolkit layer, to begin using a UIMS, and to reuse the code written at the Toolkit layer. Still another option is the ability to port down, to reuse code written with a UIMS at the toolkit layer, or to reuse code written at the toolkit layer in an environment supporting only the Xlib API.

The best strategy the designer can apply in all of these cases is to design the application user interface by partitioning the code into widgets and callback routines, following the API defined by the Xt Intrinsics. This code has the best chance of being ported (with minimal redesign) to other toolkits and UIMS systems built on top of toolkits. This is the main emphasis of this paper and will be repeated in the section on strategies below.

Extensions and Portability

The X Window System Protocol Version 11 has provided the basic foundation upon which many of the software layers described in this paper have been built. However some software layers require additional hardware support not covered by this protocol. This additional support can be added to X by defining extensions to the core X protocol. X extensions that are accepted as standards by the X Consortium will probably be widely integrated into future X servers. This will obviously increase the portability of any software that relies on the extensions. Many X servers already support the X input extensions. The Tektronix TekXPress second generation of X terminals uses this extension to provide support for tablets in addition to the standard mouse. The X extensions that will have the biggest impact on graphics applications are the PHIGS+ Extension (PEX), being implemented by SUN Microsystems Inc., and the Video Extension (VEX) being implemented by Tektronix Inc.

To date, no extensions have been proposed that extend the X server's ability to manipulate user interface components. This may be due to the original design goals to provide mechanism and not policy. Still it is tempting to consider these extensions, especially as more applications are developed using graphics-intensive interfaces.

Consider the example of a single five-item menu implemented using the Xt Toolkit and Athena widget set. To simply display the menu and drag the pointer across all choices requires over 13 client requests (270+ bytes) and many generate over 100 events (3000+ bytes) returned from the server. The network traffic could be reduced and the interactive performance increased if an extension was implemented to encapsulate the display and interaction of a downloaded menu with a single request. This feature is already standard in the NeWS window system which allows PostScript programs to be downloaded to the window server.

An alternate migration path is for user interface components to migrate out of the application to a user interface server (UI server), as opposed to the X server via X extensions. This does not eliminate the network traffic unless the X and UI servers are on the same host. However, it would eliminate the requirement that the user interface logic be linked to each application. And it would allow the UI Server to switch GUIs without requiring any change in the application. This would provide a portable GUI interface that did not even require the application to be relinked.

The UIMS software X-Pression from Unicad is already implementing UI server technology.

Case Study, TekColor Editor

We are often reminded of the saying, "There is no such thing as portable software, only software that has been ported." So to demonstrate the portability of widgets and callback routines, I will describe a case study involving the port of an application from the toolkit layer to the lower level Xlib layer.

The TekColor Editor is an interactive software application that allows the user to intuitively select screen colors and match screen colors to hardcopy colors. The TekColor Editor uses the Tektronix Color Management System (TekColor CMS) and the TekHVC (hue value chroma) color space [11], which allows colors to be described in a device-independent manner.

The TekColor Editor consists of a unique interactive graphical interface that contains both standard widgets (menus, buttons, arrowbuttons, text) as well as custom widgets (HueBar, HueLeaf). The user selects a hue from the HueBar, and then varies the color's value (lightness) and chroma (intensity) by dragging the mouse in a two dimensional widget called a HueLeaf. The changing color is displayed dynamically in the ColorPatch widget. See figure 4.

The TekColor Editor was originally written for the Apple Macintosh. It was subsequently rewritten as an X client using the Xt toolkit intrinsics and the Athena widget set. TekColor Editor was then ported to the Motif widget set running on a Tektronix XD88/10 Workstation. Finally, it was decided to support the Motif version as an X client running locally on a Tektronix second generation TekXPress XP27 X Terminal.

First a requirement specification was written to identify the customer and application requirements and the system constraints in the new environment. A brief list of the requirements and constraints follows:

- X Window System only.
- Match the user interface using the same custom widgets.
- GUI depended on customer site. (Many Motif users.)
- GUI certification NOT required.
- Native language support required.
- Xlib sufficient for all custom widget display routines.
- Complex editing features supplied in standard widgets NOT required. (No fancy text editing.)
- Maximum size of the runtime executable must be under 200 KB. (Original Motif version was 1089 KB).

Guided by the above requirements, the following design decisions were made.

The ported interface would resemble Motif since that was the appearance chosen for all local X clients running on the X terminal.

The main size reduction would need to be achieved by reducing or replacing the Xt toolkit intrinsic layer and Motif widgets, since that accounted for over 80% of the original application size. Two approaches were considered.

The first approach was to start with the original toolkit code and remove functionality, keeping only a minimum feature set. This may have worked, and increased portability, but the resulting window based widgets may have still consumed too much runtime allocated memory on the X server side.

Figure 4. TekColor Editor user interface

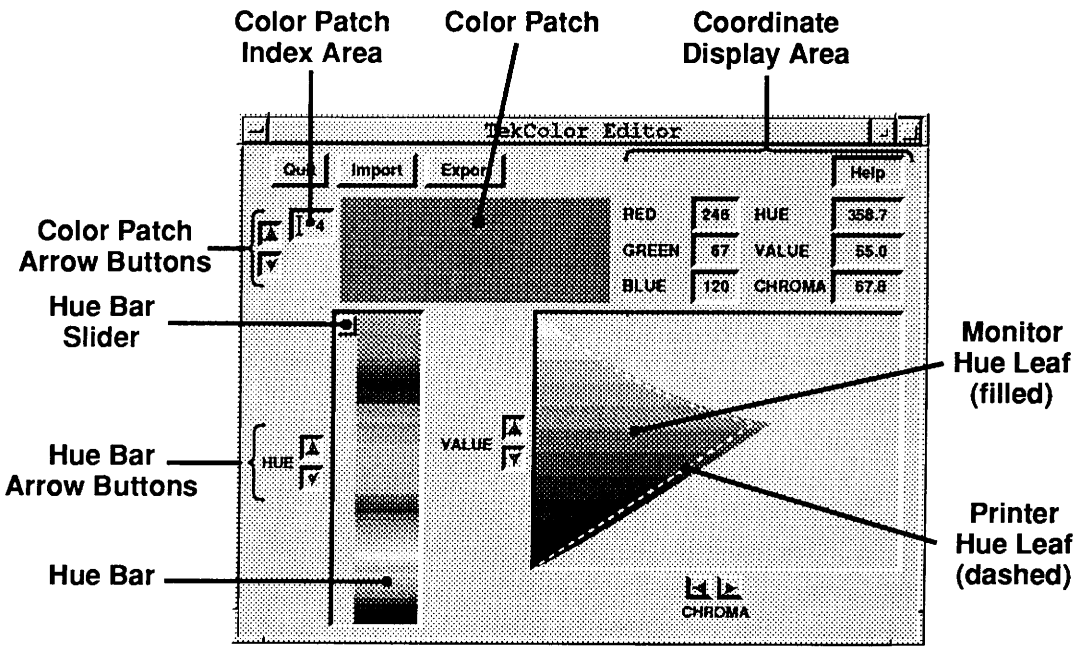
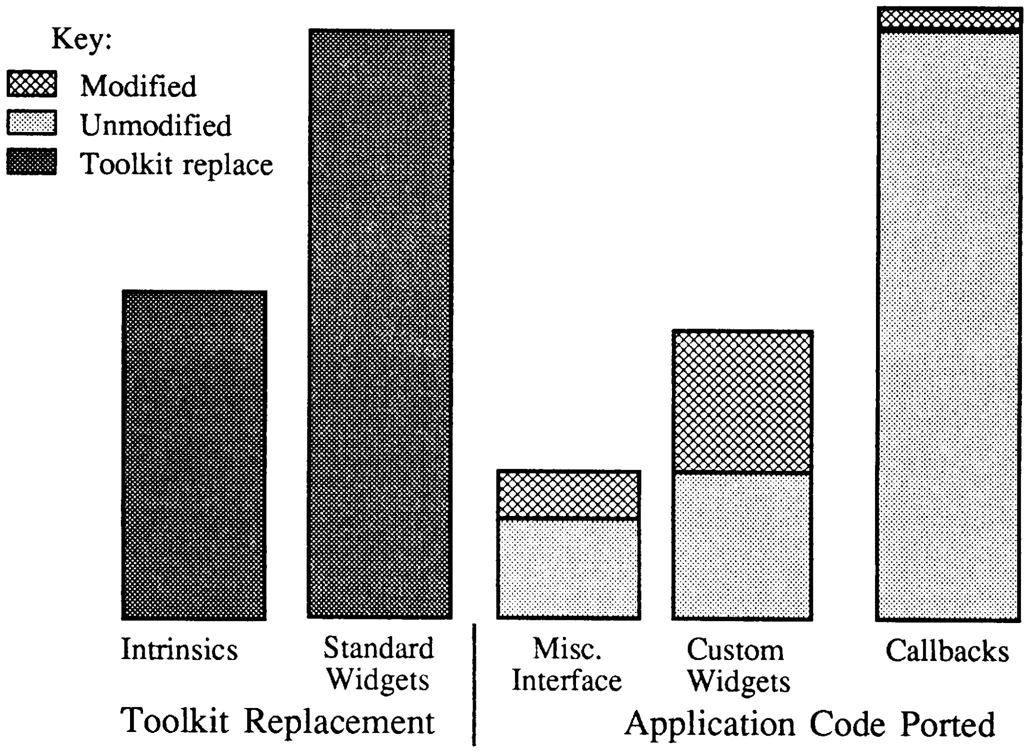


Figure 5. Relative code sizes of ported TekColor Editor application showing modifications



The second approach was to follow the same toolkit paradigm, but build a minimal feature set from bottom up. This approach was adopted since it allowed all original primitive widgets to be designed as windowless rectangular objects (gadgets).

Each original widget data structure was reduced to a subset of its original members, and changes were made to convert them to gadgets. Whenever possible data structures and procedure interfaces were left unchanged to allow reuse of code. Main features of the replacement intrinsic layer were mechanisms for traversing the gadget hierarchy, an event dispatcher, and state information to track things like which text gadget has keyboard focus. To illustrate the scaledown effort, the default "resize" routine used by all gadgets was implemented using a simple gravity mechanism requiring only 36 lines of C code.

Once the above intrinsic layer was in place, standard and custom widgets were ported, one at a time, in a straightforward manner. Reusing the toolkit paradigm made this easier since the code was already correctly partitioned.

The code size and ratio of ported to modified application code is shown in figure 5. The code used to replace the Xt toolkit intrinsic layer and standard widgets (window, composite, button, text) was reduced by an order of magnitude, but still made up 48% of the binary executable. From the original Motif application code, the ported client reused 97% of the original callback code, 50% of the original custom widget code, and 66% of other toolkit interface-related code. In total, 80% of the application code from the Motif based version was reused.

Most software developers would not consider these statistics very impressive. Indeed, the goal for code ported without modification should be higher than 99%. But given the memory constraints which necessitated moving from a toolkit to a lower layer API, the success of reusing 80% of the code is attributed to maintaining the toolkit design methodology (and API, where possible) of intrinsics, widgets, and callbacks.

The final X client was reduced to 179 KB and allocated far less runtime memory than the original Motif client. It should be emphasized that the author is not encouraging the naive claim "Look Mom, No Toolkit!" Using a toolkit is the preferred approach. Without the Xt toolkit intrinsic layer and Motif widgets, we sacrificed such important features as complex text editing, accelerators, and resource management. Since resource management was removed, native language support for this and other local X clients had to be provided using *xstr*, a string extraction preprocessor. In hind site, the lack of resource management is the weakest link in the design of the replacement toolkit. But we did succeed in the goal to reuse some the custom widgets and almost all of the original application callback routines to provide a user interface similar to the Motif version.

Successful Strategy for Maximum Portability

The following strategy is offered as a checklist to be reviewed when designing software applications that can be ported within and between the API layers described in the previous sections. The checklist should be viewed as a starting point rather than an exhaustive list. Some of the items mentioned below are only relevant for X client applications.

Requirements Phase:

1. Identify Customer requirements
   - Required Window System?
   - Required GUIs?

- Required foreign language support?

2. Identify Application requirements (API)
   - Window system features?
   - Prebuilt user interface widgets?
   - 2D bitmapped graphics? (Xlib)
   - 3D graphics, segment editing, transforms, pan and zoom (PHIGS+)
3. Review GUI style guides and certification checklists for required user interface features [2][3].
4. Review Inter-Client Communication Conventions Manual (ICCCM), an MIT X Consortium Standard, for conventions to be followed in the areas of selections, cut buffers, window management, session management, and resources [12].

Design Phase:

1. Design the application to meet the above requirements. Successful design is achieved if the application can be ported between GUIs with minimum redesign and maximum reuse of code.
2. Design the application using an external control model. Partition the application into user interface components (widgets) and application callback routines. The importance of this should not be underestimated. It is the key to portability between GUIs.
3. When possible, design user interface components from combinations of the standard widgets that are supplied with a toolkit. If additional custom widgets are needed, design them using the same toolkit API that interfaces with the standard widgets.
4. Identify areas of the application that are system-specific. System-specific now includes window system and GUI dependencies as well as machine and operating system dependencies. Partition these into modules that can be replaced easily when porting.
5. Identify the common features between different GUIs. Partition the application code so that these common features are handled in common blocks of portable code. For example, identify the X resources that are common to a standard widget (for example, the Text widget) in different GUI widget sets.
6. Avoid convenience functions that are not common to different GUI toolkits when there is a standard method that can be used for both. (Or alternatively, supplement one GUI toolkit with convenience routines from another. This can sometimes be done with simple cpp macros.)
7. Consider implementing the application using a combination of different APIs. For example, the application may require a PHIGS+ output model when outputting to and picking graphics structures from the application workspace. However, a standard GUI interface (Motif or OPENLOOK Toolkit) could be used to supply the basic input model and the PHIGS+ output routines could be called from within the application callback routines.
8. To increase portability in the short term, the application programmer should consider X server extensions accepted by the X consortium if they provide application requirements not met by the standard X core protocol. In the long term, keep an open mind to technology like the UI server which may offer programming solutions to GUI portability problems.

Conclusion

This paper described several layers of APIs that could be used for designing or porting applications with graphical user interfaces. A strategy for designing portable applications was presented that emphasized the following points:

- Portability is increased when standards are utilized as the building blocks of a graphical user interface (GUI).
- The toolkit external control paradigm, which partitions application code into widgets and callback routines, encourages modular separation of the GUI from the application-specific code. This separation facilitates porting the GUI to use other toolkits or environments without toolkits.
- Development of GUI X server extensions could increase an X applications interactive performance, as well as decrease network traffic. The portability of applications using any extensions depends on the availability of extended X servers.

Bibliography

1. "Panel Discussion: Are We Anywhere Near a Universal Graphic Language?", Proceedings of the IFIP Working Conference on Graphic Languages, May 22-26, 1972.
2. *OSF/Motif Style Guide Revision 1.0,* Prentice Hall, Inc., (1990).
3. *OPEN LOOK Graphical User Interface Functional Specification,* Sun Microsystems, Inc. (1989).
4. Richard Probst, "Blueprints for Building User Interfaces, OPEN LOOK Toolkits", *Sun Technology, (Autumn 1988).*
5. "Smile When You Say GUI", Computer Aided Engineering, (Sept 1990).
6. *OPEN LOOK Graphical User Interface Trademark Guide,* available from AT&T, OPEN LOOK GUI Trademark Quality Control Manager, 60 Columbia Turnpike, Room 129B-A208, Morristown, NJ 07962, (201) 829-8996.
7. *OSF/Motif Trademark Certification Checklist Level 1, Revision 1.0,* available from Open Software Foundation, Attention: Motif Desk, 11 Cambridge Center, Cambridge, MA 02142.
8. *X/Open Portability Guide, Version 3,* Prentice-Hall Inc., (1988).
9. Glenn Widener, "International Language Support in X11 Release 5: Building a Standard for Heterogeneous Network Computing Using Standards for Homogeneous Internationalization.", UniForum Proceedings (1991).
10. Alan Southerton, *Many Paths to X Window Programming,* UnixWorld, Volume VII Number 5, (May 1990).
11. *TekColor Color Management System Programmers Manual,* Tektronix Inc., Part Number 061-3799-XX.
12. David S. H. Rosenthal, *Inter-Client Communication Conventions Manual, Version 1.0,* MIT X Consortium Standard.

Author's Biography

Education

Bachelor of Arts Degree in Mathematics and Physics from Whitman College in 1977. Master of Science Degree in Computer Science from Washington State University in 1981.

Employment Experience

Currently Paul Shearer is a Software Engineer with the Interactive Technologies Division at Tektronix, Inc. His experience includes the design and implementation of UIMS prototypes for Tektronix workstations. Previous X experience includes participation in the implementation of two X servers on the Tektronix 4319 workstation and XN11 X Terminal. The paper presented refers to current work experience porting the the TekColor Editor graphical user interface from the Motif toolkit to a X terminal local client interface built on top of Xlib.

Paul has also worked for Bell Laboratories as a Member of the Technical Staff in the Residential Networking group prototyping videotex applications.

Trademarks

UNIX is a registered trademark of AT&T in the USA and other countries.

X/Open is a trademark of the X/Open Company Limited.

The X Window System is a trademark of M.I.T.

Open Software Foundation, OSF, OSF/Motif, and Motif are trademarks of The Open Software Foundation, Inc.

OPEN LOOK is a trademark of AT&T.

TekCMS and TekColor are trademarks of Tektronix Inc.

Microsoft is a registered trademark of Microsoft Corporation.

OS/2 is a trademark of the International Business Machines Corporation.

Presentation Manager is a trademark of the International Business Machines Corporation.

PostScript is a registered trademark of Adobe Systems Inc.

Porting Between Open Look™ and OSF/Motif GUI's

Paul E. Kimball
Digital Equipment Corporation
800 W. El Camino Real
Mountain View, CA 94040
(415) 691-4756

Porting Between OPEN LOOK™ and OSF/Motif™ GUI's

Paul E. Kimball

Digital Equipment Corporation
Independent Software Vendor Group

Abstract

One of the many choices faced by the UNIX™ software developer is the choice of a graphical user interface (GUI). Over the last two years, two proposed "standard" graphical user interfaces have emerged by consensus and are now vying for attention: OSF/Motif™ and OPEN LOOK™. This paper explores the difficulties encountered in building an application which could support either user interface style. The approach discussed involves the use of separate toolkits to implement the OSF/Motif and OPEN LOOK styles. First, the differences and similarities between the two styles and their supporting toolkits are summarized. Differences in individual widget functionality as well as overall application logic are considered. Generally-applicable workarounds for common application situations such as menus, pop-ups and graphics areas are proposed. Finally, other areas which require customized treatment are described. Out of this study, the reader should arrive at a clearer understanding of what is required in building a GUI-portable application.

1.0 Introduction

The application programmer developing for the UNIX workstation environment is besieged with choices when considering a software development platform. One of the many issues to be faced is the choice of a graphical user interface (GUI). Out of the olio of available GUI's, two proposed "standard" graphical user interfaces have emerged by consensus and are now vying for attention: OSF/Motif™ and OPEN LOOK™. Forced to choose between the two, an application supplier is faced with the risk that regardless of the choice, some group of users will not be satisfied. It would be a relief to the programmer if an application could be developed and supported which could port with relative ease from one user interface to the other.

Is this really possible? In fact, OPEN LOOK and OSF/Motif share striking similarities at a number of levels. Both are available as X-based toolkits implemented using the MIT X Toolkit Intrinsics. Based on the same Intrinsics, both toolkits may be manipulated and accessed using the same native application programming interface (API) and methodologies. There is also an astonishing degree of similarity in the set of user interface tools provided by the two widget sets. Even visually, there are correspondences between applications built with either toolkit. These similarities suggest that with careful coding, an application can be built which is at least reasonably GUI-independent.

At the programming level, however, the differences to be considered can be formidable. Areas which must be given attention when porting an application include:

- Individual widget callback and resource semantics
- Widget creation semantics and order in which tools are created
- Private window manager protocols which affect on-screen action
- Proprietary API extensions

- Binding of context-sensitive help
- Internationalization

This paper is a preliminary survey of the issues to be faced in building a GUI-portable application.

2.0 Approach

Three distinct approaches are available to the developer attempting to support both GUI's. First, it would be possible to implement either one or both with straight Xlib (or other) graphics library functions. While this approach offers the greatest degree of control over the final product and its appearance, it necessitates an intimate understanding of every aspect of visual appearance and behavior in both styles, and requires the most work.

Another possible solution to the double-GUI dilemma is a single toolkit supporting both GUI's. Such a toolkit has been built by Solbourne Computer; from a viewpoint of general usability, its dependence on C++ puts it at somewhat of disadvantage, since most current commercial software is written in "C", FORTRAN or one of the other more popular languages. This situation is bound to change over the years, and such a multi-style toolkit definitely deserves consideration if a new application is being developed.

This project considered the third alternative: using two different toolkits, each supporting its own GUI style. Since "C"-based toolkits supporting OSF/Motif and OPEN LOOK styles are widely available, this approach offers the possibility that it could be implemented readily, if sufficient correspondence is found between the two styles and their components.

There exist at least two native toolkit implementations of the OPEN LOOK style. The first is produced by AT&T, and is based on the MIT X Toolkit Intrinsics. The second is Sun Microsystems' XView toolkit. XView is layered on the X Window System, and features a SunView-like programming interface.

The definitive toolkit implementation of OSF/Motif is based on the MIT X11R4 Toolkit Intrinsics. This toolkit is offered, with minor variations, in both source or binary distributions on a number of platforms. Vendor-specific distributions of the software add miscellaneous features, but the basic toolkit in all commercial implementations is essentially the same as that which is offered by OSF.

The mutual dependence of the AT&T and OSF toolkits on the MIT Intrinsics made a comparison between these two toolkits most attractive. On the assumption that it would be easier to support two toolkits which share a common API and philosophy, further consideration of the XView implementation was dropped for the purposes of this paper.

The study was developed in three sections. First, the distinguishing features of both styles were outlined. Significant areas of similarity as well as significant differences were determined. Next, the toolkit implementations of the two styles were compared to determine major API differences and similarities.

A toolkit API consists of the routines, data types and other publicly-defined symbols and procedures used to access and control a toolkit. Both toolkits considered in this paper use the MIT X Toolkit Intrinsics as a technology base, and the same high-level routines can be used to manipulate the widgets in each toolkit. In many cases, one-to-one comparisons of the widgets implementing a particular user interface tool were possible. Both toolkits also define toolkit-specific convenience routines, data types and symbolic constants. Convenience routines are used to perform functions which are outside the scope of the MIT Intrinsics (e.g. managing a clipboard) or provide a simpler interface to complex widget manipulations then the prototypical XtSetValues/XtGetValues mechanism defined by the Intrinsics. API issues considered were:

- The purpose and semantics of individual widget resources
- The purpose and semantics of individual widget callbacks
- The functions of analogous convenience routines; where provided
- The order in which widgets are instantiated in a tree

3.0 Background

Any comparison of these two popular GUI's is complicated by the fact that they evolved very differently. OSF/Motif is first and foremost a toolkit, while OPEN LOOK is a specification for a user-interface style.

OSF/Motif is a set of software tools and an application programming interface specification, which together define the user-interface segment of the OSF Application Environment Specification. As a product provided by OSF, Motif includes a number of distinct components:

- The OSF/Motif Style Guide
- A widget set which, by default, implements the Motif visual and syntactic style
- The Motif implementation of the X Toolkit Intrinsics (in Version 1.0 only; V1.1 uses the vanilla MIT R4 Intrinsics)
- A set of utility routines for the manipulation of compound strings, which encode language-specific text for use in international applications
- A clipboard facility and routines to support asynchronous cut and paste operations between applications
- The Motif Window Manager (mwm) application
- A presentation-layer development facility, consisting of the Motif Resource Manager (mrm), User Interface Language (UIL) and UIL compiler (`uil`) application

Because OSF/Motif is relicensed by a number of vendors, vendor-supported implementations may support additional tools or capabilities as well.

OSF/Motif merges code and technologies from several predecessor toolkits. The widget set itself is practically a direct merge of the Digital *XUI* and Hewlett-Packard *Xw* widget sets. Those who are familiar with either of these toolkits will notice many similarities in style, resources and widget usage. The clipboard, compound strings and user-interface language also have their ancestry in the *XUI* toolkit. Overall, the appearance and behavior owes much to a joint HP/Microsoft style submission, and is compatible with Presentation Manager style where feasible.

The style guide for OSF/Motif permits a great deal more behavior than it specifies, and cannot be viewed as a definitive *specification*. While defining a number of common dialog controls and features specifically, it often grants permission to wax creative when appropriate to an application or its intended use. A number of recommendations are made regarding facilities which an application should provide for usability, e.g. context-sensitive help. But the exact visual format or method for supplying these facilities is often left up to the programmer. The latitude found in the style guide is mirrored in the widget set itself. Though all the widgets implement the OSF/Motif style as a default, enough resources are provided to pretty much make Motif look like anything desired. It is also one of the largest toolkits currently available, providing a great range of data-management features and visual richness.

OPEN LOOK itself is not a widget set, but a detailed specification defining the visual appearance and behavior of workstation applications. The functional specification for the OPEN LOOK visual style was developed by Sun Microsystems and describes which user-interface tools are presented to the user (e.g. Menu Buttons, Check Boxes, Menus), but it explicitly does not state how these tools are to be implemented by the programmer.

A toolkit based on the X Toolkit Intrinsics represents only one option for building applications which conform to the OPEN LOOK Style. This is the chief distinction between OPEN LOOK and other X-based graphical user interfaces (GUI's). While OSF/Motif is defined by its widget set, as much as by its style guide, OPEN LOOK is defined independently of its implementation.

This paper considers the OPEN LOOK style as implemented by the AT&T OPEN LOOK GUI X-based toolkit. The AT&T offering includes a number of separate components, each of which is important in implementing the style:

- The OPEN LOOK Style Guide and Functional Specification
- A widget set which implements the OPEN LOOK style
- Utility routines for converting units of measurement, manipulating text buffers and other miscellaneous functions

- The AT&T implementation of the X Toolkit Intrinsics (in Version 2.x only; V4.0 uses the vanilla MIT R4 Intrinsics)
- The OPEN LOOK Window Manager (`olwm`) application
- The OPEN LOOK Workspace Manager (`olwsm`) application
- The OPEN LOOK File Manager application

Compared to other styles such as DECwindows or OSF/Motif, OPEN LOOK is much more highly codified. Adherence to the OPEN LOOK style demands close cooperation between the widget set and other elements in the workstation environment, including the OPEN LOOK Window Manager, Workspace Manager and File Manager applications. The distinguishing features of the OPEN LOOK application toolkit follow from this.

To start with, OPEN LOOK widgets export fewer user-settable resources and callbacks than those found in other toolkits. This follows from the highly-specified nature of the OPEN LOOK style. Fewer resources per widget offer less possibilities to "break" the OPEN LOOK style by mistake. Also, certain global resources such as foreground and background color are intended to be set by the user through the Workspace Manager, and the toolkit cautions against setting these at the Intrinsics level.

The average OPEN LOOK widget has more built-in policy and performs more codified layout chores than does its counterpart in Motif. This makes the support/enforcement of the OPEN LOOK style practically automatic in many cases. Departures from the style are in fact difficult to implement. Many of the OPEN LOOK Shells go so far as to create their own subtrees of widget children, lest you try to give them the "wrong" ones by mistake, and thus violate the style.

At the present time, the OPEN LOOK toolkit offers fewer data management features than Motif; comprehensive support for clipboards, user-interface builders and internationalized text is not provided. Still, the detailed codification of behavior will be considered an advantage by those wishing to support a "standard" style.

4.0 Elements of OPEN LOOK and OSF/Motif Style

This section examines major elements of the current OSF/Motif and OPEN LOOK styles. I have necessarily emphasized certain similarities perhaps more than the creators of the respective Style Guides would wish. As an apology, I offer that the programmer must necessarily look for similarities before discovering differences, if there is to be any hope that a GUI-independent application can be built.

4.1 Controls

Both styles define a fundamental set of user-interface *controls*. Each control performs some atomic user-interface function, e.g. setting the value of a variable, invoking an application function or accepting a single-line text entry. Usually, a number of controls are displayed simultaneously to implement a dialog with the user. There is a fair degree of correspondence between the basic controls defined by OPEN LOOK and OSF/Motif styles (see Figure 1); however, the two styles insist on referring to analogous controls by different names. Mastering the vocabulary is one of the chores facing the programmer. The style-defined names for analogous user interface objects are described in the sections below.

4.1.1 Static Text or Images (Motif *Label*, OPEN LOOK *Read-only message* or *Caption*)

Most applications have contexts in which a static piece of text or image data must be displayed as a label or message. Such a message usually features no input semantic of its own, but simply maintains its own appearance.

4.1.2 Buttons (Motif *Button*, OPEN LOOK *Button*)

Buttons are user-interface tools which appear something like the buttons one would find on an everyday electronic appliance, and are usually invoked by clicking a pointer button while the cursor is inside the area of the button. This gives the user the impression of having "pressed" the button. Buttons are used in several ways: to trigger application functions, to display application menus, to pop up secondary windows, or to toggle application settings "on" or "off". The visual style of a button and its label indicates the context in which it is employed.

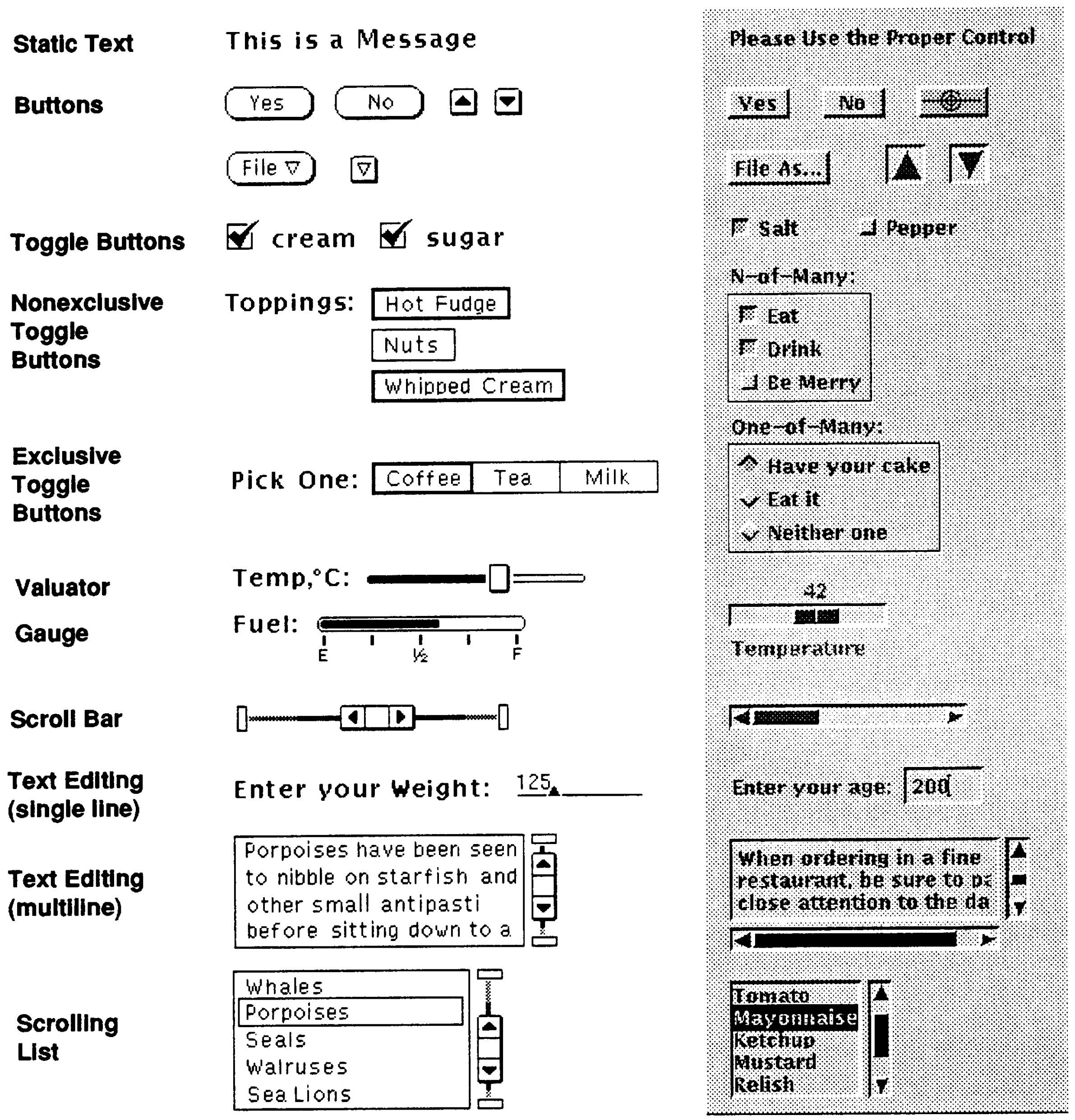


Figure 1 - OPEN LOOK and OSF/Motif Controls

4.1.3 Nonexclusive Toggle Buttons (Motif *Check Button*, OPEN LOOK *Check Box* or *Nonexclusive Settings)*

Toggle buttons maintain an internal state, which may be "on" or "off". They can occur singly, in order to control discreet application settings, or may be grouped together to indicate related settings. OPEN LOOK makes a semantic and visual distinction between settings, which group sets of related parameters, and check boxes, which simply represent individual controls that may be on or off. Items in settings are represented as rectangular buttons.

4.1.4 Exclusive Toggle Buttons (Motif *Radio Button*, OPEN LOOK *Exclusive Settings*)

When toggles are grouped, they may exhibit *exclusive* or *non-exclusive* behavior. If exclusive behavior is in force, only a single toggle in a group may be "on" at any time. This is desirable when a single option must be selected from a set of mutually exclusive choices, e.g. numbers representing baud rate.

4.1.5 Valuators (Motif *Scale*, OPEN LOOK *Slider* or *Gauge*)

A valuator allows a user to select a single value from a range of values displayed along a linear scale. The user selects a value by positioning a slider along the the scale. The OPEN LOOK style additionally defines a read-only valuator called a Gauge.

4.1.6 Scroll Bars (Motif *Scroll Bar*, OPEN LOOK *Scrollbar*)

A scroll bar visually represents the selection of a range of values from within a larger range of values displayed along a linear scale. Scroll bars are commonly used when an application displays some portion of a larger data area, e.g. a part of a text buffer, and must indicate to the user which piece is visible relative to the larger whole. By dragging a slider along the scroll bar, the user can ask the application to display different portions of the larger data area. Scroll bars in both styles provide mechanisms which allow the user to drag the slider continuously with the pointer, or move it in discreet increments. The OPEN LOOK scroll bar also presents a pop-up menu of scrolling options which can be added to by the application.

4.1.7 Text Input Fields (Motif *Text Entry*, OPEN LOOK *Text Field*)

In cases where free-form input is desirable, a user may enter textual data from the keyboard. In some cases this will be a single line: in others it will be a multiple-line text input area. Both styles define much the same semantics for text editing.

4.1.8 Scrolling List (Motif *List Box*, OPEN LOOK *Scrolling List*)

Scrolling lists present a group of related items and allow the user to select one or more of them. The OPEN LOOK definition of list behavior is much more complex than the Motif definition, and includes semantics for editing the list from the user interface, and controlling the behavior of hierarchical lists - essentially, list of lists. The OPEN LOOK list also presents a pop-up menu of list options which can be added to by the application.

4.2 Application Layout

The prototypical application layout in both styles is similar, though not identical. Every application has at least one primary window (Motif *Main Window*, OPEN LOOK *Base Window*) and may have one or more associated auxiliary windows (Motif *Dialog Boxes*, OPEN LOOK *Pop-up Windows*).

4.2.1 Window Manager Decoration

Many of the stylistic elements of the application primary window are provided by a window manager application; this is true in either environment. In particular, this includes window title bars, headers, resize handles, decorative borders, window management menus and controls, pushpins and other decoration. Both `olwm` and `mwm` are ICCCM-compliant, and implement the standard protocols and properties described therein. However, both also implement private protocols which control window decoration, resize controls, the OPEN LOOK pushpin, etc. These protocols are controlled in both toolkits by resources of the VendorShell. A similar set of resources are available to control resize handles and decoration.

4.2.2 Client Area Layout

Both styles suggest that an area at the top of the application window be used to present a set of application menus. Apart from this, each style defines several optional areas, as shown in Figure 2. Both endorse an optional message area at the bottom of the application window.

4.2.3 Paned windows

A pane is a sub-area within the application client area, within which a specific portion of the application data is viewed, or a dialog carried out. Both styles endorse the use of panes. Panes may be resizable, and since they often are used to display a view into a larger virtual space, may be equipped with scroll bars to adjust the area seen inside the pane. In cases where panes provide multiple views of the same data, OPEN LOOK specifies a number of operations which can be performed upon panes to split, combine and resize them.

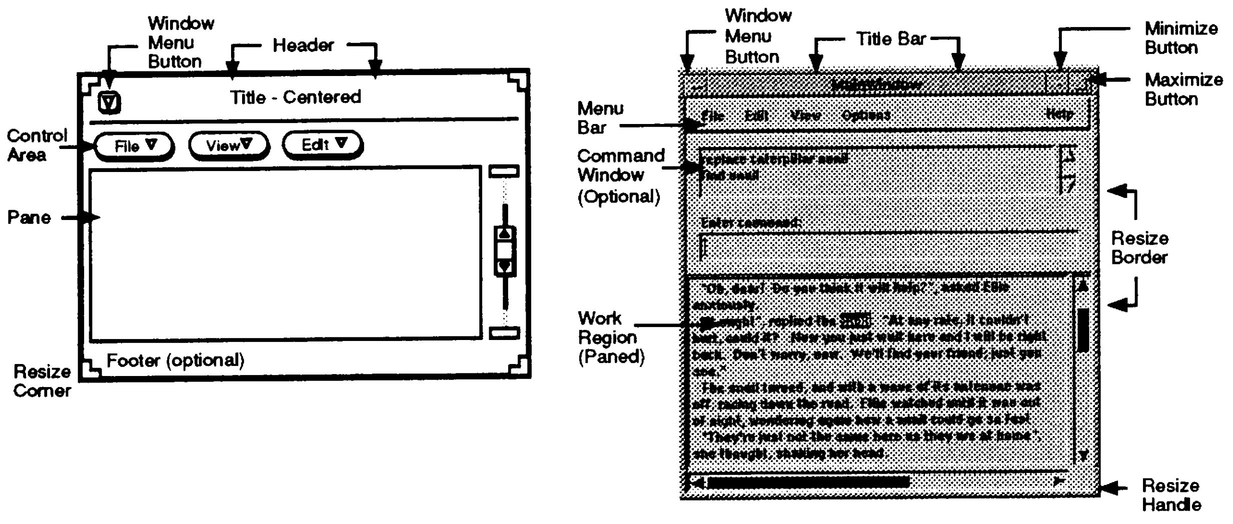


Figure 2 - OPEN LOOK and OSF/Motif Application Layout

4.3 Menus

Menus are an alternative for grouping controls together in cases where one or more functions are invoked frequently. Usually an item on a menu triggers some program function, and selecting from a menu can be thought of as entering a program command. Menus may also contain items which pop up dialog boxes, toggle application settings on and off, or invoke additional *cascaded* menus. Both styles endorse similar menu constructs.

Most applications present a set of menu buttons positioned at the top of the client area, each of which invokes an associated pull-down menu (see Figure 2); this stylistic element is endorsed by both OPEN LOOK and Motif. Motif refers to this set of menu buttons as the *menu bar*, while OPEN LOOK refers to it as a *control area*. Though OPEN LOOK does not specifically restrict the use of this area to presenting menu buttons, that is the use shown in almost all current examples of the style. The most general program options appear in these buttons, and lead the user to other more specific choices presented on pull-down menus. Both style guides suggest a similar set of standard menu choices, which are appropriate and consistent for many applications.

In either style, pull-down menus are activated by selecting the appropriate menu button. The associated menu then appears on the screen, and allows the user to select an item within it. Both styles support "click-move-click" and "press-drag-release" syntax for selecting items from pull-down menus. The major difference between the two styles is that OPEN LOOK requires menus to be invoked with the MENU button on the pointer, while OSF/Motif invokes them with the SELECT button. This difference is transparent to the programmer. Items on pull-down menus may themselves trigger additional menus in a menu cascade. A menu choice which triggers another pull-down menu is indicated in either style by a small arrow next to the item label.

Pop-up menus are invoked in either style with the MENU button on the pointer, and can be popped up anywhere in the client area. They present commonly used functions or options associated with the area under the cursor, or the currently-selected object, and are used in contexts in which they would save significant pointer movements, or provide a more intuitive interface.

Both styles offer a type of menu which always displays its current default choice. OSF/Motif refers to this as an *option menu*; OPEN LOOK calls it an *abbreviated menu button*. The currently selected choice is displayed in a small text area. When the area is selected with the pointer, the complete menu of options is popped up. A new selection may be then be made, which replaces the old selection in the display area. Option menus are usually found in dialog boxes, along with other controls.

Motif provides menu traversal using the arrow keys on the keyboard, and two other shortcuts. *Mnemonics* are single-character keystrokes which activate a visible menu choice. They can be used to traverse a tree of pull-down menus. The mnemonic used to activate a given menu item is underlined in the label for that item. *Accelerators* are keyboard sequences which invoke a menu item whether it is visible or not.

4.4 Pop-up User Interface Tools

Dialog boxes are auxiliary windows associated with a main window, the purpose of which is to carry out some specific, usually transient, interaction with the user. A dialog box groups a set of related controls, each of which implements part of the dialog.

Motif defines a number of convenience dialog boxes for use in certain common contexts. Motif *Message* dialogs convey timely information (warnings, error messages, informational messages, etc.) to the user in response to conditions discovered by a program. The visual rendition of these boxes depends on the nature of the message displayed.

Other predefined dialog boxes are brought up under user control, to enhance a specific dialog with the user. Examples include the *entry* (or prompt) dialog, which accepts text input, and the *file selection* dialog, which allows the user to peruse the file system and specify a filename.

Apart from these standard dialog boxes, OSF/Motif guidelines are very generous on layout and ordering of controls in dialog boxes. A common feature of many dialog boxes is a set of buttons along the bottom, which may be used to confirm or cancel the dialog interaction. One of these may be designated as the default button, which is invoked by the <Return> key and is visually distinct from the other buttons.

OPEN LOOK defines four distinct types of pop-up window. *Notice Windows* are analogous to the Motif Message Box, and convey information to the user. *Command Windows* accept user input and are roughly analogous to the Motif entry dialog. *Help Windows* are popped up when the user requests interactive help, and display an enlarged view of the screen area in question, together with an informative message. There is no analog to help windows in Motif. *Property Windows* are the most general pop-up, allowing the user to set application controls. Like Motif dialogs, OPEN LOOK Pop-up windows present buttons along the bottom, one of which is the default.

4.5 Button and Key Bindings

Both styles define a set of "standard" keyboard bindings for commonly-used user interface functions. Either style allows these default bindings to be revised to suit the end user or application programmer.

4.6 Context-Sensitive Help

Both styles endorse context-sensitive help. In OPEN LOOK, help is invoked by placing the pointer over the area of the screen for which help is desired. This may be a control, a graphics area or an application layout area. The HELP sequence is then entered, and help is displayed regarding the area being pointed at. Therefore, the "context" in which help is defined is determined by the area of the screen being indicated. OPEN LOOK defines a recommended presentation for help messages. A help window is popped up, containing a magnified view (complete with magnifying glass) of the the screen area, together with a helpful message.

In OSF/Motif, help is invoked in one of two ways. Applications are advised to supply a "Help" menu choice on the top menu bar for an application. Several recommended choices are defined for this menu. In the more general case, help is invoked by prssing the HELP button on the keyboard and invoking a widget. Motif does not define any particular presentation for help.

5.0 Comparison of X Toolkit Widget Sets

The OSF/Motif widget set was first released in 1989 under license from the Open Software Foundation. The information in this paper is taken primarily from the Version 1.0 release. Version 1.1, based on the MIT R4 intrinsics, has just been announced as this is written in September, 1990. Such information is available at this time has been used in a few places. Only minor changes are anticipated in the toolkit and API. The toolkit is available as a source code distribution from the Open Software Foundation, and is also distributed in binary format by a number of system vendors, including DEC, HP, IBM and SCO, who support it on their respective hardware and operating system platforms.

AT&T's OPEN LOOK GUI X Toolkit, an X Toolkit Intrinsics-based implementation of the OPEN LOOK look and feel, has been available since 1989. Information in this paper is taken primarily on the Version 2.0 release which was announced in early 1990. Version 4.0 is due out in late 1990; such information as is currently available on this release has been incorporated where relevant. There is no Version 3.0; the non-sequential release number reflects the fact that the V4.0 toolkit will be based on on the R4 intrinsics. The widget set is available in source code on the UNIX System V Release 4 distribution, and can also be had from AT&T in a binary distribution for AT&T architecture and Intel 80386 machines.

This section discusses specific similarities and differences between the two toolkit implementations.

5.1 Widget Sets

The widget sets supported by the two toolkits are shown in Table 1, which displays each widget next to its closest analog in the other toolkit. Specifics will be discussed in the following sections.

5.2 Units of Measurement

Widgets normally accept all dimensional data in integer pixel coordinates. Both toolkits endorse the use of a real-world coordinate system in order to mask differences in display resolution or pixel size, but they go about it rather differently.

Motif widgets can accept and report measurements in several alternate units, selected by the XmNunitType resource: 1/1000ths of an inch, 1/100th millimeters, decipoints and font units. Because XtSetArg cannot set a float value for a resource, these are all integer values. Widgets internally convert between the values they report, and the integer pixel coordinates used to communicate with the X server.

The OPEN LOOK widgets accept and report values in integer pixel coordinates. A set of convenience routines is provided to allow an application to convert pixel measurements to points or millimeters, as either integer or floating-point values. Thus conversion is done internally by the Motif widgets and externally by the OPEN LOOK toolkit. For the sake of portability, probably the right way to handle unit independence is to manage it within the application code.

5.3 Widget Tree Structure

The structure of the application widget tree is important if resources will be declared in class or user resource files. The fully-qualified name of a given widget resource is constructed from the names of all the widget's ancestors up to the application shell. If the widget tree constructed from either toolkit can be kept roughly homologous, resource file maintenance can be streamlined. Unfortunately, this is not always possible. Examples where homologous structure cannot be maintained include cascading menus, the OPEN LOOK Caption and the upper levels of application primary windows.

5.4 Shell Behavior and Instantiation

The two toolkits diverge on the issue of the proper treatment of shells. Shell widgets are used in both toolkits to parent menus, pop-up windows and other user-interface tools which demand a new window independent of any other. The windows associated with shell widgets are always children of the server RootWindow, and may be seen and controlled by window manager applications. In order to control window manager behavior, shells export resources which correspond to the window manager properties defined by Xlib. Shells are not managed like ordinary widgets, but are created as pop-up children with the Intrinsics routine XtCreatePopupShell. The Intrinsics define the routines XtPopup and XtPopdown as the mechanism for posting/unposting shells.

Because of the unique status of shells, the two toolkits treat them differently. Motif attempts to hide them by instantiating two varieties of "hidden" shell - the XmMenuShell and XmDialogShell. Hidden shells are instantiated by the Motif widget creation convenience routines. For example, when XmCreatePopupMenu is called, the Motif toolkit automatically creates a hidden shell, creates the XmRowColumn menu pane as a child of the hidden shell, and returns the widget ID of the XmRowColumn. This must be accomplished in a convenience routine, as XtCreateWidget provides no mechanism for the automatic creation of a parent. Motif shells may also be instantiated with XtCreatePopupShell.

Table 1 - OPEN LOOK and OSF/Motif Analogous Stylistic Elements and Supporting Widgets

Style Element	Motif widget	OPEN LOOK widget
Controls, Display and Data Entry		
Static Text or Image Label	XmLabel	StaticText Caption
Push Button	XmPushButton XmArrowButton XmDrawnButton	OblongButton
Toggle Button	XmToggleButton	CheckBox FlatCheckBox
"N-of-Many" Toggle Button	XmToggleButton (+XmRowColumn)	Nonexclusives (+ RectButton) FlatNonexclusives
"One-of-Many" Toggle Button	XmToggleButton (+XmRowColumn)	Exclusives (+ RectButton) FlatExclusives
Scale	XmScale	Slider
Gauge	no analog	Gauge (V4.0)
Scroll Bar	XmScrollBar	Scrollbar
Single-Line Text Entry	XmText	TextField
Multiple-Line Text Editing	XmText (+XmScrolledWindow)	TextEdit (V4.0)
Command Line	XmCommand	no analog
Scrolling List	XmList (+XmScrolledWindow)	ScrollingList
Menus		
Application top menu area (menu bar)	XmRowColumn	ControlArea
Menu Button triggering pulldown menu	XmCascadeButton	MenuButton
Pull-down menu	XmMenuShell (+XmRowColumn)	MenuShell
Pop-up menu	XmMenuShell (+XmRowColumn)	MenuShell
Option Menu	XmRowColumn	AbbrevMenuButton
"Standard" Pop-up windows		
Message Boxes	XmMessageBox	NoticeShell
File Selection Box	XmFileSelectionBox	no analog
Selection Box	XmSelectionBox	no analog
Prompt Box	XmSelectionBox	no analog
General pop-up window	XmDialogShell + manager	PopupWindowShell
Layout Control		
"Standard" primary window layout	XmMainWindow	FooterPanel + Form
Scrolling layout	XmScrolledWindow	ScrolledWindow
Paned window layout	XmPanedWindow	no analog
Simple fixed layout	XmBulletinBoard	BulletinBoard
Row/Column layout	XmRowColumn	ControlArea
Constraint-Driven Layout	XmForm	Form
Control Panel	XmRowColumn	ControlArea
Miscellaneous		
Client Graphics Area	XmDrawingArea	Stub
3D Ornamental Frame	XmFrame	no analog

Hidden shells pop themselves up when their child is managed. In the preceding example, if the child XmRowColumn is managed, its parent XmDialogShell pops up. When the child is unmanaged, the XmDialogShell pops itself down. The benefits of this approach are twofold. A single mechanism (XtManageChild/XtUnmanageChild) now suffices to post/unpost any widget, regardless of whether it is a pop-up widget or not. Also, the application need not explicitly instantiate the shell if the convenience routine is used.

OPEN LOOK shells are instantiated with XtCreatePopupChild, and are the first widget to be instantiated in a pop-up window or menu. Having this distinction, shells in the OPEN LOOK toolkit are given a major role in enforcing style. Shells automatically create their own manager children, which are used as layout areas inside the pop-up window. For example, the MenuShell creates a child Form and the Form's own ControlArea child. The manager children are used to parent any controls placed within the shell; the widget ID's of these managers are exported as resources by the shell itself. The advantage of this approach is that it follows the spirit of the MIT Intrinsics while enforcing the proper choice of manager to support the style appropriate to a particular pop-up tool.

These differences have several impacts on the programmer. First, the order in which widgets are instantiated will be different, depending on which toolkit is used. This is an inconvenience, but the differences here can easily be hidden by encapsulating menu or dialog box creation in an application library routine. Second, if the names of widgets are used in resource files, care must be taken to be sure that the proper widget is referenced. As an example, if a widget name is specified to the Motif routine XtCreatePopupMenu, the name is received by the XmRowColumn menu pane. If the analogous OPEN LOOK construct is created by calling XtCreatePopupShell to instantiate a MenuShell, the shell itself receives the name. In cases like this, the asterisk notation in resource files is helpful.

5.5 Controls

There is no room in this paper to present all the detailed comparisons made between each widget and its counterpart. Most differences revolve around the number and use of widget resources. Widgets were determined to be functionally equivalent if they:

- Support essentially the same user-interface abstraction
- Provide a similar set of manipulation resources and/or convenience routines.
- Provide a similar set of callbacks, from which an "adequate" common subset can be defined

The major findings in comparing widgets from the two toolkits are presented below.

5.5.1 Static Text or Image Labels

To present static textual information, the XmLabel and StaticText widgets serve roughly the same function and are equivalent in functionality. If a static image label is required, the XmLabel can display an application-supplied pixmap. To handle images, AT&T examples instantiate the undocumented Button superclass, which accepts an XImage. The fact that this widget remains undocumented is a bit troubling, yet it actually approaches a closer match with the XmLabel than does the StaticText widget.

For labeling controls, the OPEN LOOK toolkit provides the Caption widget. This is actually a manager widget which parents the child control and presents the text label next to it. The advantage of this widget is that it interacts with the ControlArea widget to enforce alignment of labels and controls in accordance with the OPEN LOOK style. There is no exact analog for this widget in the Motif toolkit; XmLabels in Motif are usually siblings of the controls which they annotate.

5.5.2 Push Buttons

The XmPushButton and OblongButton are roughly equivalent in action and resources. Motif also provides the XmArrowButton and XmDrawnButton visual variants of the XmPushButton, which support the same actions and resources.

5.5.3 Toggle Buttons

The Motif XmToggleButton and OPEN LOOK CheckBox are functionally equivalent. OPEN LOOK makes a semantic distinction between the CheckBox, which is a generalized toggle, and the RectButton (discussed below) which toggles choices within a group of related items.

5.5.4 Exclusive and Non-exclusive Groups of Toggle Buttons

The Motif XmToggleButton and OPEN LOOK RectButton are both toggle buttons that maintain a state, and support very similar resources and semantics. RectButtons are always parented by an Exclusives or Nonexclusives manager widget, which enforces the button layout peculiar to the OPEN LOOK style. Also, RectButtons support several appearance resources which are used to reflect changes in application state or the combined state of several related toggle buttons.

Exclusive behavior is enforced in Motif by making the XmToggleButtons children of an XmRowColumn.

5.5.5 Valuators

The Motif XmScale and OPEN LOOK Slider widgets are generally equivalent in action and resources. Version 4.0 of the OPEN LOOK toolkit implements the Gauge widget. There is no direct analog for this in the Motif toolkit; for portability the best approach would be to use the Motif XmScale widget as an output-only widget.

5.5.6 Scroll Bars

The Motif XmScrollBar and OPEN LOOK Scrollbar widgets are generally equivalent in action and resources.

5.5.7 Single-Line Text Entry

The Motif XmText and OPEN LOOK TextField widget are generally equivalent.

5.5.8 Multiple-Line Text Entry

The Motif XmText and OPEN LOOK TextEdit (V4.0) widget are generally equivalent in function and actions. The Motif widget does not provide scroll bars of its own; scrolling behavior is implemented by making the XmText widget a child of an XmScrolledWindow. Certain resources of the XmText are passed to its parent to control behavior, and the convenience routine XmCreateScrolledText is provided to automatically create both the XmText child and its XmScrolledWindow parent. A similar set of convenience routines is provided in both toolkits to set, get and edit the displayed text.

5.5.9 Command Line Editor

The Motif widget set provides a widget which implements the command-line entry area used in a primary window. There is no direct analog for this in the OPEN LOOK widget set.

5.5.10 Scrolling Lists

The Motif XmList and OPEN LOOK ScrolledList widgets implement a similar user interface function, but there are important differences in their implementation which must be addressed. Because these tools implement a fairly complex dialog with the user, numerous convenience routines are provided in both toolkits to position the list, add items, delete items and retrieve the selected item or items. The Motif widget supports convenience routines which are declared in the widget header files. In the OPEN LOOK implementation, the addresses of the analogous convenience routines are exported as resources by the ScrolledList widget.

The Motif widget does not provide scroll bars of its own; scrolling behavior is implemented by making the XmList widget a child of an XmScrolledWindow. Certain resources of the XmList are passed to its parent to control behavior, and the convenience routine XmCreateScrolledList is provided to automatically create both the XmList child and its XmScrolledWindow parent.

5.6 Menus

5.6.1 Menu Buttons and Pull-down Menus

The Motif XmCascadeButton and OPEN LOOK MenuButton are functionally equivalent, but present procedural and structural differences which must be addressed by the programmer. The OPEN LOOK widget automatically creates an associated MenuShell, Form and ControlArea. The widget ID of the ControlArea is exported as a resource of the MenuButton widget, and must be retrieved with XtGetValues so that it can be used to parent button children.

5.7 Application Layout

The greatest divergence between toolkits is found in the manager widgets which implement layout control. Consequently, these present the programmer with some of the greater challenges.

5.7.1 Simple Fixed Layout (Bulletin Board)

The Motif XmBulletinBoard and OPEN LOOK BulletinBoard widgets are functionally equivalent.

5.7.2 Constraint-Drive Layout (Form)

The Motif XmForm and OPEN LOOK Form perform much the same function as managers, enforcing relative position of their children with respect to one another. The relative positioning of children is defined by constraint resources set on the children themselves. Unfortunately, there is little similarity between the constraints defined by the two widgets. This is an area requiring customized treatment.

5.7.3 Row/Column Layout

Both toolkits provide a manager widget which orders its children in neat rows and/or columns. This is most often used to provide order in menus or pop-up dialog boxes. The properties of the Motif XmRowColumn and OPEN LOOK ControlArea widgets are similar, and they are both used to parent a set of choices when employed in menus. However, the OPEN LOOK widget interacts with any Caption children in order to enforce the alignment of controls and captions used in OPEN LOOK pop-up windows.

5.8 Application Drawing Area

Both toolkits provide a generalized widget which can be used for this purpose. The Motif XmDrawingArea and OPEN LOOK Stub widgets are actually very different, but may be treated as functionally equivalent if the application defines its own translations for the widgets.

5.9 Context-Sensitive Help

AT&T's toolkit supplies the help registration function OlRegisterHelp, which is used to associate an application-supplied help message with a particular window, widget or gadget. When help is invoked with the pointer indicating a registered object, the help facility opens the stylized OPEN LOOK help window. The proto-typical help window is supplied courtesy of the (undocumented) Help widget, and displays the help message together with an enlarged view of the screen area for which help was requested. The help facility may also be vectored to an application-supplied help routine; in effect, the application receives a callback when help is requested. In this case it is up to the application to provide an appropriate visual interface to help services.

Each widget in the OSF/Motif widget set supports a help callback list, invoked when the help sequence is activated with focus in that widget. A default help sequence is defined for some widgets; for others it is up to the programmer to define a translation which calls the routines in the help callback list. This means that different programmers may build different conventions for invoking help. Moreover, the exact format for displaying help is not defined by the OSF/Motif Style Guide. It is up to the programmer to determine the "proper" use of the help callback, and define an interface to an application-supplied help facility.

This is an area requiring customized treatment, since conformance to the OPEN LOOK help style is considered mandatory by the OPEN LOOK specification.

5.10 File Selection

Both styles define a "standard" file selection menu, and the choices on these File selection and management are treated similarly by the two toolkits.

The OSF/Motif toolkit provides a file selection widget that displays the files in a directory, allowing the user to traverse the directory tree and select a file. The selected filename is returned in a callback to the application. Motif applications should use the file selection widget to ensure stylistic conformity.

OPEN LOOK filenames may be retrieved in a command window; since no toolkit-specific widget is provided for this purpose, this is an area requiring customized treatment. Applicaitons running in the OPEN LOOK environment should also be prepared to interact with the OPEN LOOK File Manager application. Files needed by an application are selected in the File Manager window, and then dragged to the application window. Communication with the File Manager is accomplished through properties.

5.11 Namespace collisions

It would be nice to be able to build an application and link both toolkit libraries at compile time, such that a run-time choice of user interface style could be made. This is only possible if there is no fatal conflict between exported symbols defined by the two toolkits. A preliminary review of the namespaces reserved by the toolkits shows no overlap; the naming convention followed by each toolkit are shown in Table 2. Bold letters indicate literal characters that invariably appear in the symbol. Capitalization is important and follows the conventions shown.

Version 1.0 of Motif and Versions 2.x of the OPEN LOOK toolkit rely on customized implementations of the Intrinsics library (`libXt.a`). Without a great deal of effort, this effectively precludes the reliable use of these releases together. By the time this paper is published, the latest releases of both toolkits will be based on the standard R4 Intrinsics, and could be linked against the MIT libraries. This issue should be revisited at that time.

Table 2 - Naming Conventions

Symbol Type	OSF/Motif Convention	OPEN LOOK Convention
Resource name symbolic constant	**XmN**resourceName	**XtN**resourceName
Resource class symbolic constant	**XmC**ResourceClass	**XtC**ResourceClass
Toolkit-specific symbolic constant	**Xm**SYMBOLIC_CONSTANT	**OL_**SYMBOLIC_CONSTANT
Toolkit-specific data type	**Xm**Type	**Ol**Type
Widget class	**Xm**WidgetClass	WidgetClass
Class pointer used in XtCreateWidget	**xm**WidgetClass**WidgetClass**	widgetClass**WidgetClass**
Widget header file	<**Xm/**WidgetClass**.h**>	<WidgetClass**.h**>
Convenience routine	**Xm**RoutineName	**Ol**RoutineName
Widget-specific creation routine	**XmCreate**WidgetClass	n/a
Widget-specific convenience routine	**Xm**WidgetClassRoutineName	**Ol**WidgetClassRoutineName

6.0 Conclusions

The good correspondences between the OPEN LOOK and OSF/Motif styles and toolkits indicate the general feasibility of applications supporting both GUI's. This work has defined situations requiring special treatment and a number of areas for further study. Some general principles for building GUI-independent code are derived below.

- Do design from the start to support both styles, and choose user interface constructs accordingly.
- Do insist on the standard MIT Intrinsics, as this will make it possible to add user-written widgets where necessary, and possibly will make run-time choice of GUI a reality.
- Do use toolkit-specific convenience routines where helpful, particularly to manipulate complex widgets, such as the text and list widgets.
- Do encapsulate toolkit-specific code inside application library routines. Only build the library of tools that you need to implement your user interface.
- Don't use "exotic" resources and callbacks unless absolutely necessary.
- Do adopt the style definition of the more tightly-defined style, as long as it does not break the less-tightly defined style.

- Don't fight the style implemented by the widget set. Wherever possible, let the widget set itself enforce/support the style for the application. Not only is this more in keeping with the philosophy of "standard" styles, it also allows the programmer to avoid a lot of detailed programming.
- Don't fight the window manager. Window managers implement a portion of the look and feel inherent in a GUI specification. Unless there is an overwhelming reason to override this behavior , it is recommended that the default behavior of these window managers remain unchallenged. This means, specifically, that application developers should give up notions about the "proper" placement of pop-up tools, "correct" sizes for application windows, and "right" stacking order. The ICCCM is required reading.

7.0 Items requiring additional study

A number of issues remain to be resolved. First, new releases of both toolkits will be available by the end of 1990; these must be evaluated. In particular, the OPEN LOOK toolkit will implement 3D appearance in this release, and new resources are expected to control this behavior. The entire issue of internationalization has not been explored in any detail. Procedural issues surrounding keyboard traversal, cursor handling and interclient communication have yet to be addressed. Detailed consideration of pop-up interface tools is also not yet complete at the time of publication.

8.0 Acknowledgements

I would like to thank Lindsey Robinson, Chuck Price and the ISV Program at Digital Equipment Corporation for the time and space to work on this, as well as the loan of a DECstation and significant information on Motif and DECwindows/XUI. Thanks also to Marcel Meth (now at Lotus Development Corp.) for his long and thoughtful hours discussing and comparing toolkits, and to Steve Humphrey at AT&T, for generously providing information on the OPEN LOOK toolkit.

This paper was entirely composed and typeset on a DECstation 3100 using DECwrite, an X Toolkit-based WYSIWYG editor and document processing system produced by Digital Equipment Corporation. Figures and illustrations were also developed in DECwrite. Screen images were dumped with the MIT **xwd** utility and then converted to 256-greyscale PostScript images by a version of **xwud** which was modified for the purpose. Drafts were rendered in PostScript and printed on a Digital Equipment LPS40 network laser printer.

9.0 References

The OSF/Motif manuals provide detailed information on the Motif style and programming interface, and are published as a five-volume set by Prentice-Hall:

1. OSF/Motif Style Guide, Revision 1.0
   Prentice Hall, 1990 ISBN 0-13-640491-X
2. OSF/Motif User's Guide, Revision 1.0
   Prentice Hall, 1990 ISBN 0-13-640509-6
3. OSF/Motif Programmer's Guide, Revision 1.0
   Prentice Hall, 1990 ISBN 0-13-640525-8
4. OSF/Motif Programming Reference, Revision 1.0
   Prentice Hall, 1990 ISBN 0-13-640517-7
5. OSF Application Environment Specification User Environment Volume, Revision A
   Prentice Hall, 1990 ISBN 0-13-640483-9

The source books describing OPEN LOOK style are the Style Guide and Functional Specification produced by Sun Microsystems, listed below:

6. OPEN LOOK Graphical User Interface Style Guidelines
   Addison-Wesley, 1990, ISBN 0-201-52364-7

7. OPEN LOOK Graphical User Interface Functional Specification
   Addison-Wesley, 1990 ISBN 0-201-52365-5

Programming documentation on the AT&T OPEN LOOK toolkit is included in the UNIX System V Release 4 manual set. Relevant volumes:

8. UNIX System V, Release 4; OPEN LOOK Graphical User Interface User's Guide
   Prentice Hall, 1990 ISBN 0-13-931916-6

9. UNIX System V, Release 4; Programmer's Guide: OPEN LOOK Graphical User Interface
   Prentice Hall, 1990 ISBN 0-13-931908-5

10. UNIX System V, Release 4; OPEN LOOK Graphical User Interface Programmer's Reference Manual
    Prentice Hall, 1990 ISBN 0-13-931924-7

AT&T has also established a trademark certification effort. Elements of the OPEN LOOK style required for certification are described in the following volume, available from AT&T:

11. AT&T OPEN LOOK Graphical User Interface Trademark Guide
    American Telephone and Telegraph, 1989

10.0 Trademarks

UNIX is a registered trademark of AT&T

OPEN LOOK is a trademark of AT&T

DECstation, DECwrite and DECwindows are trademarks of Digital Equipment Corporation

OSF/Motif is a trademark of the Open Software Foundation

"X" and X Window System are trademarks of the Massachusetts Institute of Technology

PostScript is a trademark of Adobe Systems, Inc.

Presentation Manager is a trademark of Microsoft, Inc.

The Use of Erasable Optical Disk Technology for Data Archival in a UNIX Internet Environment

Mark A. Clark
LTV Aircraft Product Group
P.O. Box 655907 M/S 31-06
Dallas, TX 75265-5907
(214) 266-5612

The Use of Erasable Optical Disk Technology for Data Archival in a Unix Internet Environment

Mark A. Clark
Senior Process Control Engineer

Nondestructive Test Lab
LTV Aircraft Products Group
(214) 266-5612

Recent developments in storage media have created new possibilities for the critical task of archiving data files in computerized data acquisition environments. In particular, the recent commercial availability of erasable optical media allows for the development of archiving systems which allow for greater data security, longer storage life and much faster retrieval of archived files than the traditional magnetic tape archive.

This paper describes the implementation of a data archiving system with over 2 gigabytes of mass storage capacity using both magnetic and erasable optical disks on a DEC Microvax II computer system under the Ultrix operating system. This machine serves as the central host for the LTV Quality Network, which includes eight Unix hosted, multi-axis robotic ultrasonic test systems used for the non-destructive testing of aircraft components. These ultrasonic test systems generate an average of 20 to 30 megabytes of inspection data per day. Government specifications require that the data used in evaluating the acceptance by Quality Assurance of these aircraft parts be retained for at least 5 years. In the past, data archival was performed on the local UT machine, with the data being stored on magnetic tape cartridges. This practice required a significant amount of CPU resources on each ultrasonic test system, which effectively precluded the use of the machine for its intended purpose during the time required for the tape backup. The manual tape backup system was replaced in 1989 by a remote backup system* which copies ultrasonic data files across the Internet to the Microvax II system in the NDT Lab, where it was centrally archived to magnetic tape cartridge on that system. The project described in this paper provides the next step in the evolution of this function.

* as described in: Clark, Mark A., "Remote Backups of Internet Hosts Through FTP," <u>1990 Uniforum Conference Proceedings</u>

Design Goals

As this project was begun, the disk space available for on-line storage of data files (that is, data files available for immediate download to the production ultrasonic systems) was limited to the space on the 'h' file system of our RA81 disk drive on the MicroVax. (about 320 Megabytes) This file system was also needed for user files and software applications software on the Vax. In practice, we were able to store seven to ten days worth of on-line data on this file system. Thus, one of the chief design goals for the project was to increase the on-line storage area dramatically.

Another goal was to provide quick access by the ultrasonic machine operators to archived data files. (those which are no longer on-line) The old method for this was that the UT operator would call the NDT Lab on the phone and tell us that he needed a certain file. We would then have to find out what tape cartridge that file was on, mount the tape in the tape drive and then copy the file from tape to the on-line storage directory from which the UT operator could then download it to his local system. Reduction of the amount of system and manpower resources required for archiving and retrieving data was, therefore, a prime consideration.

The third major design goal of the archival system was to increase the reliability of the archived data. Erasable optical media is more stable than either magnetic disk or magnetic tape media and is less susceptible to environmental factors such as heat and magnetic fields.

Design Choices

The selection of the Maxtor Tahiti disk drive as our erasable optical disk drive was not difficult. At the time we placed the order for the hardware for this system, it was the only erasable optical disk drive that was actually on-the-shelf ready to be sold. (although Fujitsu and Sony were close behind) The Tahiti drive with the 1.2 gigabyte platter will allow the creation of a file system on the MicroVax of 424 megabytes per side.

Because of our need to maximize the amount of on-line storage of data files on the system, we selected the Fujitsu 2263S SCSI magnetic disk drive which will allow the creation of a file system of 624 megabytes on our Vax under Ultrix.

To ease the chore of system configuration, we wanted to purchase a disk controller board that would control both the magnetic disk drives for the on-line storage area and the erasable optical drives for the archival area, while using an available device driver. The U.S. Design Q-STOR/QT model 1108 SCSI Host Adapter for the Q-Bus was selected because of its success in controlling Maxtor Tahiti drives on MicroVaxen under VMS. We were the first known site to attempt using this hardware under Ultrix on a MicroVax, which led to a little gotcha discussed later. This board appears to the operating system as if it is a DEC KDA50-Q disk controller board which allows the use of standard RA series device drivers. Each 1108 board will control four connected SCSI devices.

Because of the importance of maintaining operation even in the event of the failure of a component of the system, we elected to use a "no single point of failure" approach, that is, we bought two of everything. Two controller boards, two Fujitsu drives and two Maxtor Tahiti drives. This way, the failure of any single piece of hardware in the system will allow us to continue operation with the remaining hardware until that component is repaired.

Hardware Configuration and Installation

Installation of the controller boards and disk drives in the MicroVax was fairly straightforward. The CSR (control/status register) address of the 1108 boards were set to the addresses recommended by DEC for the second and third KDA50 controllers on a MicroVax II. The only other hardware settings required were the SCSI ID number settings on the disk drives.

Operating System Modifications

Kernel configuration and compilation on the MicroVax was accomplished through the /etc/doconfig script provided with the system by DEC. This script basically automates the editing of the configuration files and then makes the kernel in the standard way. Most recent Unix releases provide equivalent friendly reconfiguration programs for modifying the hardware setup of the system. Shown below is an excerpt from the configuration file of the MicroVax showing the csr address and driver information for the two disk controllers and the disks created for them.

```
.
.
.
adapter       uba0     at  nexus ?
controller    uda1     at  uba0
controller    uq1      at  uda1      csr 0160334    vector uqintr
disk          ra4      at  uq1       drive 0
disk          ra5      at  uq1       drive 1
disk          ra6      at  uq1       drive 2
disk          ra7      at  uq1       drive 3
controller    uda2     at  uba0
controller    uq2      at  uda2      csr 0160340    vector uqintr
disk          ra8      at  uq2       drive 0
disk          ra9      at  uq2       drive 1
disk          ra10     at  uq2       drive 2
disk          ra11     at  uq2       drive 3
.
.
.
```

Note that the /etc/doconfig program creates four disks for each controller because that is the maximum number of drives that can be slaved from a single controller. Below is a portion of the message that is seen when the MicroVax is booted. Note that, here, only the disks detected at boot time are shown. The two spare disk drivers per controller are not made available for use by the system unless they are physically present on the system at boot time.

```
.
.
.
real mem  = 16769024
avail mem = 13763584
using 210 buffers containing 1676288 bytes of memory
MicroVAX-II with an FPU
Q22 bus
klesiu0 at uba0
tmscp1 at klesiu0 csr 174500 vec 774, ipl 17
tms0 at tmscp1 slave 0
uda0 at uba0
uq0 at uda0 csr 172150 vec 770, ipl 17
ra3 at uq0 slave 3
uda1 at uba0
uq1 at uda1 csr 160334 vec 764, ipl 17
ra4 at uq1 slave 0
ra5 at uq1 slave 1
uda2 at uba0
uq2 at uda2 csr 160340 vec 760, ipl 17
ra8 at uq2 slave 0
ra9 at uq2 slave 1
.
.
.
```

File System Creation

In creating the file systems for the magnetic and optical disk drives, we felt that, to simplify application development, it was highly preferable to have the entire disk be a single large file system. The procedure for doing this is to use /etc/newfs to create an 'a' file system for each of the disks. Then the /etc/chpt program (which appears to be Vax specific, but most other systems probably have an equivalent command) is used to change the partition size of the 'a' file system to the maximum block length for the disk. Finally, the /etc/newfs program is run again to create a new 'a' file system of the desired size. Note that, if you wish to create a single maximized partition for a disk drive, the partition must be the 'a' file system. This is because default file system sizes are determined from the file /etc/disktab. The /etc/chpt program stores the non-default partition table on the 'a' file system, which must, of course, exist in order for this to work. The output of these commands as it appears when run on one of the Tahiti drives is shown below:

```
ndtvax.mark # newfs -n rra9a ra80
Warning: 186 sector(s) in last cylinder unallocated
/dev/rra9a:   15872 sectors in 37 cylinders of 14 tracks, 31 sectors
    8.1Mb in 3 cyl groups (16 c/g, 3.56Mb/g, 1216 i/g)
super-block backups (for fsck -b#) at:
 32, 7008, 13984,
```

Note that the first time /etc/newfs is invoked for the 'a' file system, it shows the file system size as 8.1 megabytes. This is because this is the size listed for the 'a' partition in /etc/disktab. The /etc/chpt command is used both to determine the maximum number of blocks available on the disk and to alter the partition table to set the 'a' partition to the total disk size. The first step is to invoke /etc/chpt with the -q option. The top block of the last available file system on the disk ('h' in this case) is the maximum blocks available.

```
ndtvax.mark # chpt -q /dev/rra9a
/dev/rra9a
Current partition table:
partition     bottom         top        size     overlap
    a              0       15883       15884     c
    b          15884       49323       33440     c
    c              0      904990      904991     a,b,d,e,f,g,h
    d         131404      166663       35260     c,h
    e         166664      201923       35260     c,h
    f         201924      904990      703067     c,h
    g          49324      131403       82080     c
    h         131404      904990      773587     c,d,e,f
```

Next, /etc/chpt, given the starting and ending block numbers of the file system, changes the partition table on the 'a' file system itself. The output of /etc/chpt shows the new partition table of the disk.

```
                         ---- print new partition table
                        |   ---- partition (file system) a
                        |  |   ---- starting block number
                        |  |  |   ---- ending block number
                        |  |  |  |       ---- device special file
                        |  |  |  |      |
                        |  |  |  |      |
ndtvax.mark # chpt -v -pa 0 904990 /dev/rra9a
/dev/rra9a
New partition table:
partition     bottom         top        size     overlap
    a              0      904989      904990     b,c,d,e,f,g,h
    b          15884       49323       33440     a,c
    c              0      904990      904991     a,b,d,e,f,g,h
    d         131404      166663       35260     a,c,h
    e         166664      201923       35260     a,c,h
    f         201924      904990      703067     a,c,h
    g          49324      131403       82080     a,c
    h         131404      904990      773587     a,c,d,e,f
```

When /etc/newfs is run again for the 'a' file system, the partition table on the new file system overrides the default partition table in /etc/disktab.

```
ndtvax.mark # newfs -n rra9a ra80
Warning: partition table overriding /etc/disktab
Warning: 348 sector(s) in last cylinder unallocated
/dev/rra9a:  904976 sectors in 2086 cylinders of 14 tracks, 31 sectors
    463.3Mb in 131 cyl groups (16 c/g, 3.56Mb/g, 1600 i/g)
super-block backups (for fsck -b#) at:
32, 7008, 13984, 20960, 27936, 34912, 41888, 48864, 55840, 62816,
69792, 76768, 83744, 90720, 97696, 104672, 111136, 118112, 125088,
132064, 139040, 146016, 152992, 159968, 166944, 173920, 180896, 187872,
194848, 201824, 208800, 215776, 222240, 229216, 236192, 243168, 250144,
257120, 264096, 271072, 278048, 285024, 292000, 298976, 305952, 312928,
319904, 326880, 333344, 340320, 347296, 354272, 361248, 368224, 375200,
382176, 389152, 396128, 403104, 410080, 417056, 424032, 431008, 437984,
444448, 451424, 458400, 465376, 472352, 479328, 486304, 493280, 500256,
507232, 514208, 521184, 528160, 535136, 542112, 549088, 555552, 562528,
569504, 576480, 583456, 590432, 597408, 604384, 611360, 618336, 625312,
632288, 639264, 646240, 653216, 660192, 666656, 673632, 680608, 687584,
694560, 701536, 708512, 715488, 722464, 729440, 736416, 743392, 750368,
757344, 764320, 771296, 777760, 784736, 791712, 798688, 805664, 812640,
819616, 826592, 833568, 840544, 847520, 854496, 861472, 868448, 875424,
882400, 888864, 895840, 902816,
```

Mounting the file system and running /bin/df shows the space available on the new file system.

```
ndtvax.mark # mount /dev/ra9a /opt2
ndtvax.mark # df
Filesyste Total   kbytes  kbytes   %
node      kbytes  used    free     used  Mounted on
/dev/ra3a   7423    6104     577   91%   /
/dev/ra3g  38847   31947    3016   91%   /usr
/dev/ra3h 361590  219000  106431   67%   /usr/users
/dev/ra4a 624870  475170   87213   84%   /mag1
/dev/ra5a 624870  487730   74653   87%   /mag2
/dev/ra8a 424173  182594  199162   48%   /opt1
/dev/ra9a 424173       9  381747    0%   /opt2
```

In the case of the erasable optical drives, the above file system creation procedure must be followed every time new media is mounted in the drive. This is a pretty common occurrence in our installation, so the following script was written to automate the function:

```
*************************************************************************
# program: optinit - initialize erasable optical media
#
# author:  Mark A. Clark
#
clear
echo '     Optical Disk Initialization Program  (rev. 1.0, 8/17/90)   (mac)'
echo ' '
echo 'This program allows the superuser (only) to initialize the'
echo 'erasable optical disk media used in the archiving of ultrasonic'
echo 'data files on this system. It should be used with extreme caution'
echo 'as initializing media which contains data will result in the loss'
echo 'of that data.'
echo ' '
echo ' '
case $1 in
1|opt1|/opt1|/opt1/|ra8a|rra8a|/dev/ra8a|/dev/rra8a)
    drive=rra8a
    ;;
2|opt2|/opt2|/opt2/|ra9a|rra9a|/dev/ra9a|/dev/rra9a)
    drive=rra9a
    ;;
*)
    echo 'Device number may be 1 for opt1 or 2 for opt2. '
    echo ' '
    echo -n 'Enter Device number or [Enter] to cancel:  '
    read inword
    case $inword in
    1)
         drive=rra8a
         ;;
    2)
         drive=rra9a
         ;;
    *)
         echo 'optinit: operation cancelled.'
         sleep 1
         exit 1
         ;;
    esac
    ;;
esac
echo ' '
beep
echo '****  WARNING   WARNING   WARNING   WARNING   WARNING  ****'
echo -n 'This process will erase all contents of target media in /dev/'
echo $drive
echo -n 'Are you absolutely sure that you wish to proceed?  (y/N)  '
read reply
echo ' '
```

```
case $reply in
y|Y)
    newfs -n $drive ra80
    chpt -v -pa 0 904990 /dev/$drive
    newfs -n $drive ra80
    fsck /dev/$drive
    echo ' '
    echo -n 'Operation Complete, Press [Enter] to continue ... '
    read reply
    ;;
*)
    echo 'optinit: operation cancelled.'
    sleep 1
    ;;
esac
****************************************************************************
```

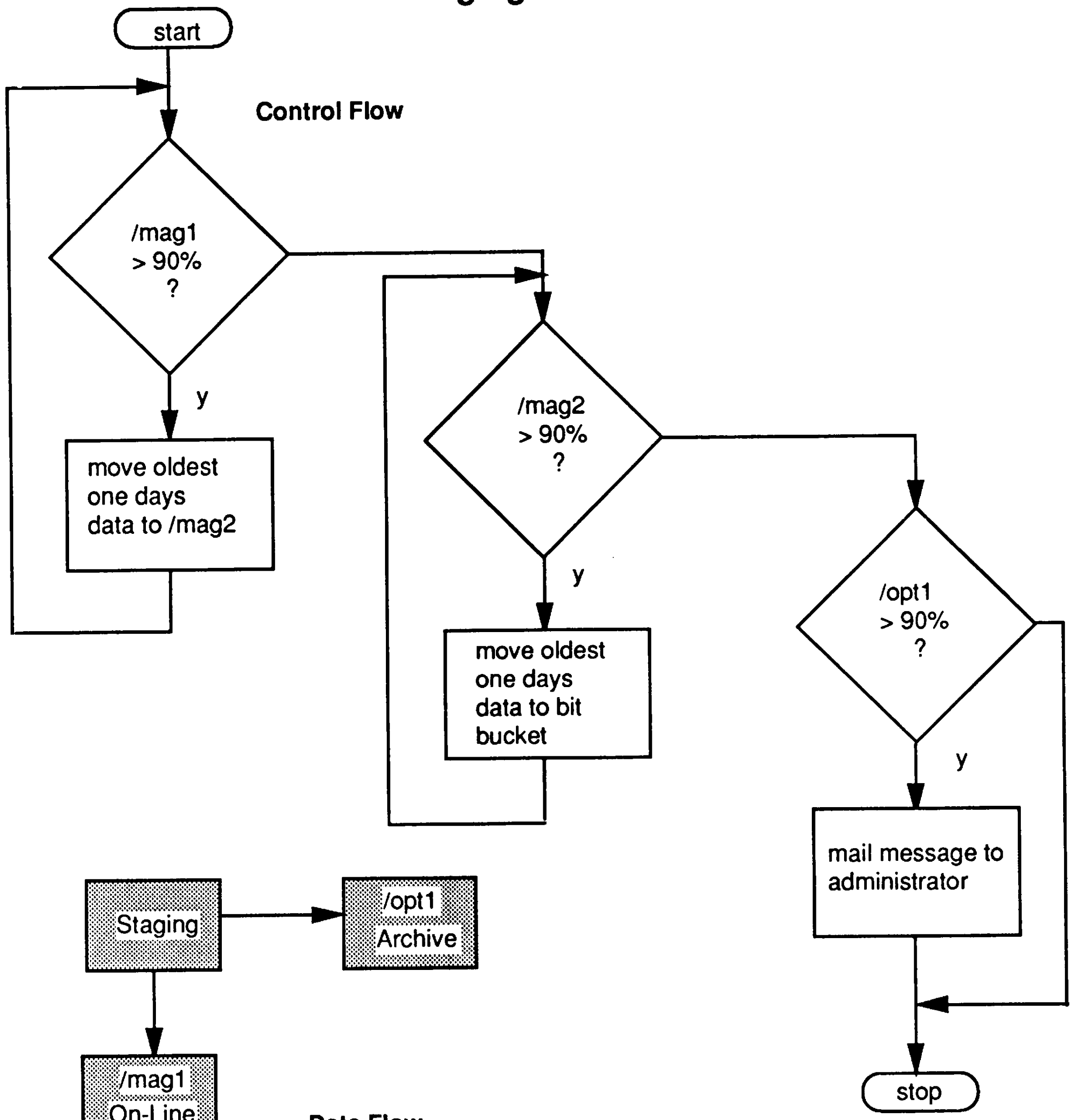
Archival System Software

The rbackup program, detailed in a previously published paper, was modified for this project to copy the data files to a staging directory. From there, the files are first copied to the /opt1 directory and then moved to the /mag1 directory, leaving the staging directory empty. This insures that all data entering the on-line storage area is mirrored to the archive area. Also modified for this system were the programs which run on the UT systems and allow the UT operators to download data files from the MicroVax. An additional program allowing the operators to request data files from archive via Internet mail was written and added to their menu interface.

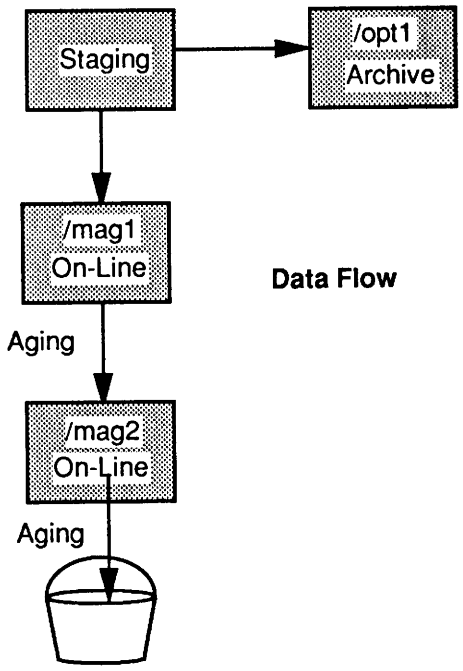
File Aging

The 'aging' program automates the aging of ultrasonic data in on-line storage on the MicroVax. First, the size of the /mag1 file system is tested. If this size is above the percentage limit set in the fssize.awk script (currently 90 percent) then the oldest one days worth of data files are moved over to the /mag2 file system. This check is then repeated until the file system is below the limit. This process is then repeated for /mag2, but, in this case, the oldest files are removed from the disk. The /opt1 file system is checked against the same size limit so that a warning can be mailed when the media needs to be changed on the Tahiti. Data and control flow charts of this process are shown below:

Aging Process



start
Control Flow
/mag1 > 90% ?
y
move oldest one days data to /mag2
/mag2 > 90% ?
y
move oldest one days data to bit bucket
/opt1 > 90% ?
y
mail message to administrator
stop



Listed below are the scripts which perform these functions:

```
****************************************************************************
#   program: aging
#
#   author:        Mark A. Clark
#
#   purpose:  provide file aging and transfer for archive system
#
#   This Bourne shell script is normally started by cron but can be
#  run interactively by root.
#
#   parameters: called by cron (normally) or interactively (by root)
#
#
HOME='/usr/users/local/aging'

# check /magl file system to see if it is full

while ($HOME/fssize /magl)
do

# move oldest one day worth of files to the next file system, then check
again

    $HOME/move /magl /mag2
done

# check /mag2 file system to see if it is full

while ($HOME/fssize /mag2)
do

# move oldest one day worth of files to trash and check again

    $HOME/move /mag2 /user/BUCKET
done

# check /optl file system to see if it is full
if $HOME/fssize /optl
then

# send mail telling me to change media

    /usr/ucb/mail mark < $HOME/alertme

fi

# clean up after yourself

rm -f $HOME/*.tmp
rm -f /user/BUCKET

****************************************************************************
```

The 'aging' script, listed above, is the main driver for file aging on the Vax. In order to determine whether the file system contents is above the defined limit, the fssize script is called.

```
***************************************************************************
#   module: fssize
#
#   author:        Mark A. Clark
#
#   Called by the 'aging' script. Returns 0 if the file system
#   passed in as $1 is full, otherwise returns 1.
#
#       (note: the term 'full' in this case means larger than the
#   amount allowed by the 'limit' variable in the fssize.awk
#   program.)
#
# set up default directory

HOME='/usr/users/local/aging'

df $1 | grep /dev/ | awk -f $HOME/fssize.awk > $HOME/flog.tmp
if /usr/bin/fgrep -s 'file system full' $HOME/flog.tmp
then
    exit 0
else
    exit 1
fi
***************************************************************************
```

The fssize script calls the fssize.awk program.

```
***************************************************************************
#   module: fssize.awk
#
#   author:   Mark A. Clark
#
BEGIN {

# note:
# the 'limit' variable can be changed here as circumstances dictate.
# it is, of course, to our advantage to keep as much data on-line as
# possible.
#
# known bugs: If the file system gets above 100% this doesn't work.
#             Awk fails to grok that 100 is greater than 90 for some reason.

limit = 90
}
$5 > limit {print "file system full" ; }

***************************************************************************
```

The 'move' script, which calls the awk program 'move.awk,' moves the oldest one days worth of data files from the directory passed in as $1 to the directory passed in as $2.

```
*****************************************************************************
#   module: move
#
#   author:        Mark A. Clark
#
#   purpose: move one days worth of data files from directory passed
#         in as $1 to directory passed in as $2
#
# set up default directory

HOME='/usr/users/local/aging'

# put parameters where awk can get at 'em

/bin/echo $1 $2 > $HOME/directories

# generate directory listing in reverse time stamp order, oldest first

/bin/ls -ltr  $1 > $HOME/files.tmp

/bin/awk -f $HOME/move.awk $HOME/directories $HOME/files.tmp >
$HOME/move.cmd

/bin/sh $HOME/move.cmd

*****************************************************************************
```

```
*****************************************************************************
#   module: move.awk
#
#   author:   Mark A. Clark
#
#
#   input:    piped directory listing of source directory
#
#   output:   Unix 'mv' command to stdout
#
#
#
BEGIN {
# initialize log file name
LOGFILE = "/usr/users/local/aging/flist" }
NR == 1 { FROM  = $1; TO = $2 }
NR == 3 { olddays = $6 }
# for each subsequent record, do comparisons on directory listing
NR > 2 && olddays == $6  { print "mv "FROM"/"$8" "TO  }

*****************************************************************************
```

Results

The completed system has generally met the original design goals that were set for it. The 1248 megabytes of on-line storage area on the Fujitsus translates to approximately 3 months worth of ultrasonic data files produced by the production machines. Each side of an archive platter on the Tahitis will hold between three and four weeks of data. If the need arose, four additional SCSI devices are available and already configured into the kernel. This would allow us to expand by adding Fujitsu, Tahiti or most any other SCSI type drives at will.

The 1108/Tahiti combination is reported to work very well under VMS on a MicroVax. We do experience a problem under Ultrix which appears to be related to the fact that VMS does not have mountable file systems as such, the way Unix has. The optical platter must be inserted in the Tahiti drive before the system is powered up, otherwise the 1108 board will not recognize that the Tahiti exists. Once the system is up and running, it must be powered down before the media can be released again. This means that, instead of merely unmounting the file system and ejecting the platter, we have to shut the Vax down and cycle power on the Tahiti before the media will eject. This has not, so far, been a major inconvenience because our system doesn't require too frequent media changes. In some applications it would be a real pain. U.S. Design is aware of this problem, but, as of this writing, we have received no time estimate from them on when a solution will be available.

Acknowledgments

In addition to myself, the project described in this paper is the result of the efforts of my friend and co-worker, Fred M. Burns, whose help and expertise were, as always, invaluable to the completion of the archiving system.

CD-ROM and UNIX: Making CD-ROMs Usable Under the Multi-user UNIX System Environment

Thomas K. Wong
Sun Microsystems, Inc.
M/S 5-44, 2550 Garcia Ave.
Mountain View, CA 94043
(415) 336-6750
twong@eng.sun.com

CD-ROM and UNIX: Making CD-ROMs Usable Under the Multi-user UNIX System Environment

Thomas K. Wong

Sun Microsystems, Inc.
2550 Garcia Ave
Mountain View, CA 94043
Email: twong@eng.sun.com

1. INTRODUCTION

CD-ROM technology (Compact Disc - Read Only Memory) is increasingly significant for inexpensive distribution of software and large volumes of data. Production costs of CD-ROMs are under $2 in quantity and each CD-ROM can store up to 650 Megabytes of data.

One of the two major problems [1] that may slow down the acceptance of using CD-ROMs in the UNIX system environment is the slow access time and the low data transfer rate of the CD-ROM devices. While the poor performance of CD-ROM devices in the single user environment such as PCs may be acceptable, it may at times become intolerable under the multi-tasking and multi-user UNIX system environment.

This paper describes the design and a prototype implementation of an ISO 9660 CD-ROM cache file system for the UNIX system environment to overcome the poor access performance of the CD-ROM device under a heavy use situation. The CD-ROM cache file system reduces the impact of the slow random access time and the low data transfer rate of the CD-ROM devices by caching the most recently accessed data and directory information on a local file system.

Under the CD-ROM cache file system, CD-ROMs can be shared by many users without significant degradation in access performance. Another pleasant side-effect of the CD-ROM cache file system is that a CD-ROM disk mounted as the CD-ROM cache file system appears to be "writable", and "corrections" to the CD-ROM disk can be made easily.

2. PERFORMANCE PROBLEM OF THE CD-ROM FILE SYSTEM

The existing generation of CD-ROM devices has a very slow average access time and very low data transfer rate. For example, the average access time of a CD-ROM drive is about half a second, and the maximum data transfer rate is about 150K bytes per second.

However, when a CD-ROM drive is lightly accessed, the CD-ROM file system performance is found to be acceptable. This is because most CD-ROM access patterns follow the principle of locality of reference, that is, the next access to the CD-ROM drive will be very close to the current location. As a result, the average CD-ROM access time under a light use situation is in the 20ms to 30ms range instead of the expected half a second.

To illustrate the effect of the principle of locality of reference, consider the following tests: "ls -l" and "wc *"[2] on the /manual_pages/man1 directory in the SunCD Demo Disc-1.0. The directory contains 467 "man" files, with of a total 71536 lines, 273306 words and 1552333 bytes.

[1] The other problem is that the ISO 9660 CD-ROM Standard does not fully support the UNIX file system semantics. see Reference[x].

[2] *wc* is a UNIX command to display a count of lines, words, and characters of a file

Sun SparcStation 1 with UNIX file system on a SCSI disk						
command	user time	system time	real time	cpu util	# disk io	# page fault
time ls -l	0.5s	1.0s	0:06	22%	18	5
time ws *	4.0s	3.5s	0:16	45%	546	544

Sun SparcStation 1 with CD-ROM file system						
command	user time	system time	real time	cpu util	# disk io	# page fault
time ls -l	0.4s	0.6s	0:05	19%	2	2
time ws *	4.0s	2.0s	0:16	36%	468	468

The result shows there is no noticeable difference in execution time of the above tests between a CD-ROM drive and a SCSI drive. This is also true when running most programs directly from the CD-ROM.

If the above tests are repeated when there is another process accessing the CD-ROM drive at the same time, the performance changes drastically.

Sun SparcStation 1 with CD-ROM file system under heavy use						
command	user time	system time	real time	cpu util	# disk io	# page fault
time ls -l	0.4s	0.7s	0:11	10%	1	1
time ws *	4.0s	1.5s	9:43	0%	468	468

This is because when a CD-ROM drive is heavily accessed, the principle of locality of reference no longer applies. The slow access time now becomes the bottleneck. As a result, the command "wc *" takes close to 10 minutes to complete instead of the usual 16 second if the CD-ROM drive is not under heavy use.

Under the multi-user, multi-tasking UNIX environment, it is not possible to know in advance the access load of a CD-ROM device. Therefore, the response time of the CD-ROM file system under such environment may either be acceptable or intolerable. This unpredictability of response time discourages the sharing of CD-ROM devices, especially the exporting of CD-ROM file systems under NFS. Unless this performance problem can be overcome, CD-ROM devices will be treated just like a tape device, and will be used mainly as a distribution medium.

3. PROPOSED SOLUTION TO OVERCOME THE PERFORMANCE PROBLEM

There are many solutions to overcome this performance problem. The simpliest solution is to copy the contents of a CD-ROM disk to a local disk, using tar(1) or cpio(1). If it is known in advance that the contents of a CD-ROM will be accessed heavily, this may be the only viable solution. The only problem is to find the disk storage space to hold the data copied from a CD-ROM disk.

Because of the 600M byte storage capacity, there is a tendency shown by the CD-ROM disk publishers to pack the CD-ROM disk with more information. It will be getting more and more difficult to find the local disk space to store the contents of the CD-ROM disk, particularly when only some data may be needed to be on-line all the time. A general solution or policy is therefore needed to use the disk space more effectively.

Another solution is to have multiple CD-ROM drives all use the same copy of the CD-ROM disk. The purpose is to distribute evenly the CD-ROM accesses to each CD-ROM drive. This approach is similar in concept with the replicated file system. The biggest drawback is the extra expense of acquiring multiple CD-ROM drives and multiple copies of the same CD-ROM disk.

Another approach is to use a large Random Access Memory (RAM) buffer cache that is shared by all of the CD-ROM drives in a system. CD-ROM data are first copied to the RAM buffer cache. Subsequent accesses to the same piece of data will be retrieved from the RAM buffer cache, thus reducing accesses to the CD-ROM drive. The major drawback, besides the added cost expense of the RAM buffer, is that the data in the RAM buffer cache does not get preserved between each reboot of the system.

All of the above approaches lead to a very simple and low cost solution, namely, to use the local disk space as a cache to store the most recently accessed data retrieved from the CD-ROM drive. The first access to a CD-ROM will automatically store the data retrived from the CD-ROM on a local disk. Subsequent accesses to the same piece of data will be retrieved from the local disk instead of from the CD-ROM drive. The benefits of this approach are:

(1) Low Cost
No additional hardware is required - all that is needed is a local disk.

(2) Scalabilty
The local cache can be a small inexpensive SCSI disk or an expensive IPI disk. This solution is therefore good for both the low cost desk top systems as well as expensive large servers.

(3) Data Persistency
Data in the local cache persist across a system reboot

(4) Allow update
Since CD-ROM data is first retrieved from the local cache, updating the local cache has the same effect as updating the CD-ROM disk. As a result, the CD-ROM disk appears updatable.

4. DESIGN GOAL OF THE CD-ROM CACHE FILE SYSTEM.

The purpose of this paper is very specific - to solve the poor access performance of CD-ROM file systems under heavy use . The solution proposed is to use the cache file system concept. However, the concept of a cache file system is very generic, and can be applied equally well to, say, using a local cache to cache a NFS file system to reduce the network traffic.

The fact that CD-ROMs are read-only makes implementing a CD-ROM cache file system relatively simple because we don't have to worry about the cache coherence problem. In view of the objective of this paper, the prototype CD-ROM cache file system will work only with the ISO 9660 Format CD-ROM disks, and the *4.3 BSD ufs* as the local file system. Other design goals of the CD-ROM cache file system are:

(1) To maintain UNIX file system semantics.

(2) To overcome the poor access performance of CD-ROM under heavy load.

(3) To support updates to the CD-ROM cache to make the CD-ROM disk mounted as a CD-ROM cache file system appear writable.

(4) Must be flexible to allow many CD-ROM caches stored on a local file system. The number of CD-ROM caches that can be stored on a local file system should only be limited by the available disk space.

(5) Should allow selectively pre-caching the contents of a CD-ROM disk to a CD-ROM cache.

5. THE CD-ROM CACHE FILE SYSTEM DESIGN AND IMPLEMENTATION

5.1. The CD-ROM Cache File System

The CD-ROM cache file system *(CCFS)* consists of two components:

(1) the ISO 9660 CD-ROM file system.

(2) a directory (CD-ROM Cache) in a local *4.3BSD ufs* file system that is used to store the most recently accessed data retrieved from the ISO 9660 CD-ROM file system.

There is one CD-ROM cache for each mounted ISO 9660 CD-ROM file system. Since a CD-ROM cache is a directory, a local ufs file system can be used as CD-ROM caches for many CD-ROM disks.

CCFS is not a user visible file system type. It is activated automatically as part of the *mount* operation if the mount point (directory) of a ISO 9660 CD-ROM file system is a matching CD-ROM cache (see section 5.2).

5.2. The CD-ROM Cache

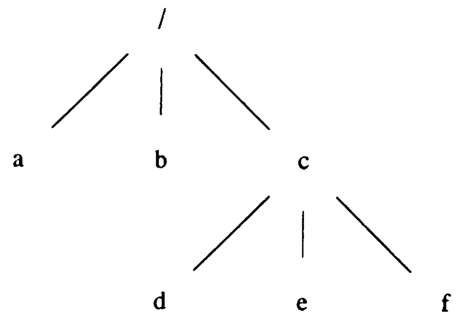
The CD-ROM cache is a directory in a local ufs file system. Each CD-ROM cache contains a ".pvd" file that identifies the CD-ROM cache. The ".pvd" file contains a copy of the primary volume descriptor of the ISO 9600 CD-ROM disk.

When a CD-ROM cache is the mount point of a ISO 9660 CD-ROM file system, the contents of the ".pvd" file is then compared with the Primary Volume Descriptor of the ISO 9660 Format Disk. If they match, CCFS is activated automatically.

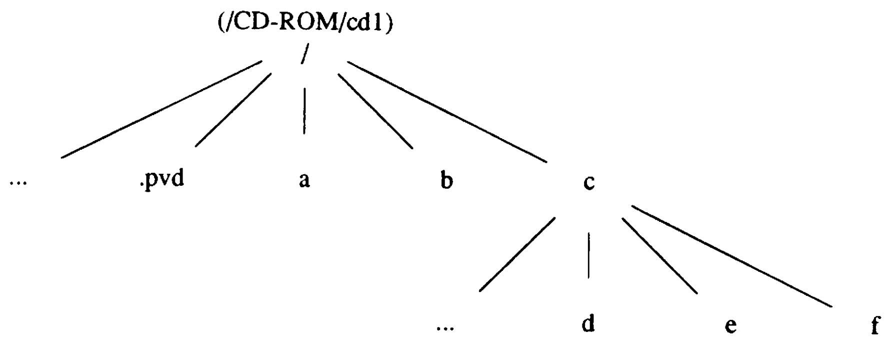
The CD-ROM cache also contains a sub-directory that is either a duplicate of the root directory of the CD-ROM disk, or a complete duplicate of the whole directory tree (including all files and directories) on the CD-ROM disk.

Every directory in a CD-ROM cache that is replicated from the CD-ROM disk also contains one additional new file, named "...". The "..." file is a copy of the ISO 9960 CD-ROM directory file for this directory. Files or directories that have never been referenced are stored as hard-links to the "..." file, that is, the contents of these files or directories are the same as the content of the "..." file.

For example, if a CD-ROM disk contains the following directory structure:



Then the contents of a CD-ROM cache under the path /CD-ROM/cd1 of a UFS file system is:



where file "a" and "b" are hard-linked to file "..." in directory /CD-ROM/cd1, if they have not yet been referenced. Otherwise, they will contain the data copied from the CD-ROM disk. Similarly, file "d", "e", and "f" are hard-linked to the file "..." in directory /CD-ROM/cd1/c, only if they have not been referenced before.

Under this scheme, every file on the CD-ROM disk will be in the CD-ROM cache. If a file cannot be found in a CD-ROM cache, it will not be found on the CD-ROM disk either. If a file or a directory has never been referenced, it will be stored as a hard-link to the file "..." in the same directory. The hard-link of each unreferenced file in the CD-ROM cache will be replaced automatically by CCFS with a corresponding file or directory from the CD-ROM disk the first time it is referenced. Therefore, there is no storage space overhead under this scheme for any unreferenced files in the CD-ROM cache.

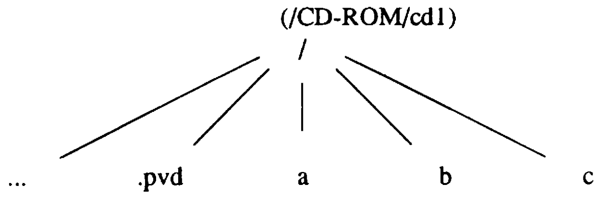
5.3. Creating a CD-ROM Cache

The *initcache* command is provided to completely replicate the directory tree of the ISO 9660 CD-ROM disk, with each data file hard-link to the "..." file in the same directory.

For a CD-ROM disk with 132 directories and 6870 files, the *initcache* command takes approximately 8 minutes to complete. Since many CD-ROM disks have even more files and directories, it is not uncommon for the *initcache* command to take an hour to finish. The reason that the *initcache* command takes so long is because creating a new directory and doing a hard link are done synchronously under ufs.

A quicker way to create a CD-ROM cache is to used the "makecache" option in the mount command, which creates a CD-ROM cache with only the root directory level of the CD-ROM disk getting replicated.

For example, using the same CD-ROM disk as described in section 5.1, the "makecache" option of the mount command will create:



where file "a", "b", and "c" are all hard-linked to file "...". The directory "c" will be created with another level of information filled in only if it is referenced.

5.4. Accessing the CD-ROM Cache

The CD-ROM cache can be accessed in two ways: under the UFS file system or under the CD-ROM cache file system.

Since the CD-ROM cache is a regular subtree under UFS, normal ufs operations also apply to the CD-ROM cache. Files can be accessed, created, modified, or deleted. If files are created, they will have a user-id and group-id of the person who creates the file.

However, some of the operations on the files in the CD-ROM cache may not make sense because of these anomalies:

(a) Files that are hard-linked to the "..." file are meanless under UFS. This is because they have never been referenced. Only the CD-ROM cache file system knows how to retrieve data from these files correctly.

(b) Some files may have holes in them. This is because some parts of the files have never been referenced before and have not been filled up with the data from the CD-ROM disk.

Accessing the CD-ROM cache under CCFS will eliminate the above anomalies. Files that are hard-linked, when referenced, will be converted automatically to a regular file or a directory, depending on the information stored in the file "...". Data are cached automatically on demand on a per page basis. Holes in a file automatically get filled in one page at a time with the contents from the corresponding files on the CD-ROM disk.

A newly created file written into a CD-ROM cache will appear as if it exists originally on the CD-ROM disk. Each page of a file that is modified in the CD-ROM cache will appear overwriting the data page of the corresponding file on the CD-RÓM disk.

Cached files may be deleted in the CD-ROM cache. In this way, the matching files on the CD-ROM disk will be whiteout. A "whiteout" file can be made visible again (unwhiteout) by simply using a hard-link (ln(1V)) to link the same name to the file "..." in the same directory.

If enough data from the CD-ROM drive has been cached into the CD-ROM cache, the CD-ROM cache can be accessed directly under UFS, and the CD-ROM drive can be freed up for other purposes.

5.5. Cache Space Management

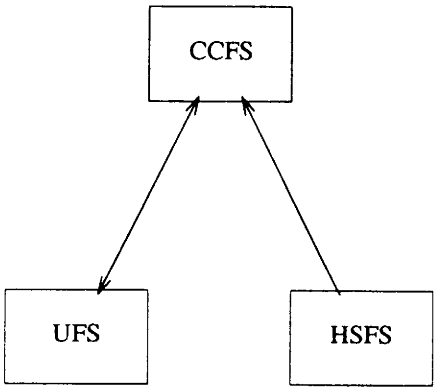
The user can specify a space limit in the */etc/fstab* on the amount of disk space that a CD-ROM cache can consume. Upon reaching the limit, the data file caching capability of the CD-ROM cache file system is disabled. The CD-ROM cache file system then functions exactly the same as the ISO 9660 CD-ROM file system.

Another utility is provided to scan thru a CD-ROM cache to remove files that have not been referenced recently (user defined) to reclaim the disk space from the CD-ROM Cache.

5.6. CCFS Implementation

The prototype implementation of the CD-ROM cache file system was done on Sun's implementation of UNIX, SunOS.[3] SunOS supports multiple file systems simultaneously through the use of a virtual file system *(vfs)* abstraction. By using this mechanism, the use of the CD-ROM cache file System is transparent to the application using the file system. The *vfs* mechanism in UNIX System V Release 4 is similar to the one currently in SunOS, and other implementations of UNIX.

Even though the cached CD-ROM file system (CCFS) is not visible in the user level, it does exist as a separate *vfs* object with its own *vfs* and *vnode* operations. The relationships among CCFS, HSFS, and UFS are depicted as follows, with arrow denotes the flow of data.



The CCFS implementation follows the same implementation technique used by other virtual file systems, and is surprisingly simple. Most of the time, CCFS behaves like a file system switch by redirecting most *vfs* or *vnode* operations to either the UFS or HSFS layer for processing. There are two CCFS *vnode* operations that are quite different from other virtual file systems. They are the *lookup* operation and the *getapage* operation.

5.7. CCFS vnode lookup operation

The purpose of the *lookup* operation is to associate the name of a file with a *vnode*, the internal representation of a file in the SunOS kernel.

In addition, the CCFS *lookup* operation is also responsible for converting any unreferenced file in the CD-ROM cache into a file with exactly the same file attributes as the corresponding file on the CD-ROM disk.

Once invoked by the *vfs* layer, the CCFS *lookup* operation simply redirects the *lookup* operation to the UFS layer to search the CD-ROM cache. If the UFS *lookup* operation returns failure, the CCFS *lookup* operation will also return failure. This is because of the CD-ROM cache design - if a file is not found in the CD-ROM cache, it will definitely not exist on the CD-ROM disk.

If the UFS *lookup* operation succeeds and returns a *vnode*, the returned *vnode* is then checked to see if the file is an unreferenced CD-ROM cache file. Under the CCFS design, a CD-ROM cache file is unreferenced if it is a hard-link to the "..." file. Under UFS, if a file is a hard-link to another file, the link

[3] SunOS is a trademark of Sun Microsystems, Inc.

count field in the *inode* area embedded in the *vnode* will be greater than one. Most of files in a UFS file system have link count equal to 1. This hint is therefore very important in reducing the number of files needed to be checked as unreferenced files, because only regular data files with link-count greater than 1 need to be checked.

To check whether a file is unreferenced, the UFS layer is called to lookup the "..." file (this should be fast because the "..." file should have already been in the directory name cache.) The *vnode* returned from this UFS *lookup* is then compared with the *vnode* returned from the previous *lookup*. If both *vnodes* are the same, the file is an unreferenced CD-ROM cache file.

To convert the unreferenced CD-ROM cache file into a file with the same file attributes as the corresponding file on the CD-ROM disk, the "..." file is searched for the unreferenced file name, by using the *lookup* operation in the HSFS layer. The file attributes (including the length of the file) for the unreferenced file is then used to create the cache file in the CD-ROM cache.

For those files that are created as part of the *lookup* operation, CCFS stores additional information in the UFS *inode* area. They are:

(a) The *ic_flags* field is set to indicate that this *inode* belongs to a CD-ROM cache file and has a corresponding file on the CD-ROM disk. If this file has been modified, this field is also used to indicate that the space occupied by this file should not be reclaimed by CCFS.

(b) The *ic_gen* field is used to store the *i_number* of the "..." file in the same directory. UFS will increment this field when this *inode* is freed and reused by another file. Thus, CCFS will not be confused by deleting a file and recreating another file with the same name because the *ic_gen* number will not have a correct *i_number* for the "..." file.

(c) The *ic_spare[0]* is used to stored the offset in the "..." file that contains the directory record describing the corresponding file on the CD-ROM disk. This field is also used to identify the corresponding file on the CD-ROM that is cached by this file.

5.8. CCFS vnode getapage operation

The *getapage* operation is the page fault handler for the CD-ROM cache file system. The *getapage* operation must handle the page faults from three kinds of file:

(1) HSFS files

These files originate from the CD-ROM disk but are not put into the CD-ROM cache by CCFS. This is possible if CCFS has stopped caching data in the CD-ROM cache because the CD-ROM cache has reached the assigned storage space limit.

(2) UFS files

These are files that are newly created by users in the CD-ROM cache. They are not duplicates of any files on the CD-ROM disc.

(3) CD-ROM cache files

These files are duplicates of files on the CD-ROM disk.

The page fault handling for HSFS and UFS files is very straight forward. The UFS *getapage* or the HSFS *getapage* operation is called to retrieve the data page. Under SunOS, each page is identified by a *vnode* and an offset within the *vnode*. CCFS simply switches the page identity with the CCFS *vnode*.

The page fault handling for CD-ROM cache files is slightly more complicated. All unreferenced data pages of a CD-ROM cache file are represented as holes. When referenced, these holes in a CD-ROM cache file must be filled with the data pages of the corresponding file from the CD-ROM disk.

To handle a page fault that maps to a hole in a CD-ROM cache file, the CCFS page fault handler first pages in the data page from the CD-ROM disk using the HSFS *getapage* operation. Ideally, the data page should be switched with a new identity with the CCFS vnode, and the page marked as dirty. A dirty page will eventually get flushed out to the disk by the pageout demon of the operating system.

Unfortunately, this strategy does not quite work. If the system crashes after the disk storage space for the page is allocated but before the data page is written to the disk, CCFS will mistakenly think the data on the disk is valid. In order to avoid this confusion problem, the data page is first written to the disk synchronously, then followed by allocating the disk space. These two steps substantially increase the overhead of the caching data from the CD-ROM disk into the CD-ROM cache.

6. CCFS Performance

The performance tests in section 2 are repeated on a CD-ROM cache created using the *initcache* command:

Sun SparcStation 1 with CD-ROM cache file system						
command	user time	system time	real time	cpu util	# disk io	# page fault
time ls -l	0.5s	0.9s	0:06	25%	3	3
time ws *	3.8s	7.6s	1:27	13%	1486	467
time ws * (again)	4.0s	3.2s	0:15	45%	321	314

The performance result of "ls -l" is similar in all three file systems. However, the CD-ROM cache file system takes 1 minutes 27 second in the first run to complete the "wc *" command versus the 16 second for ufs file system and the cdrom file system under light use. Even though this may sound discouraging, it is anticipated that with the added concurrency provided by a multi-threaded OS kernel, the performance should be greatly improved and should be comparable with the performance of the ufs file system and the cdrom file system under light use. On the other hand, once a file is cached in the CD-ROM cache, the next execution of "wc *" command takes about the same time as expected from a local disk.

7. CONCLUSION

This paper describes the design and a prototype implementation of an ISO 9660 CD-ROM cache file system that allows a CD-ROM disk to be shared by many users without significant degradation in access performance. In addition, corrections can also be made to a CD-ROM disk mounted under the CD-ROM cache file system. While the overhead is still high the first time the data is cached into the CD-ROM cache, it is expected that the overhead will be substantially reduced under a multi-threaded OS kernel.

8. ACKNOWLEDGEMENT

The author would like to thank Mark Smith for his review of this paper.

9. REFERENCES

[1] ISO 9660 - Information processing - Volume and file structure of CD-ROM for information interchange

[2] Wong, T.K., "CD-ROMs and UNIX Systems: The implementation of a CD-ROM Disc Format and File System For SunOS Systems", Uniforum 1990 Proceeding, pp 229-237.

[3] Klieman, S. R., :Vnodes: An Architecture for Multiple File System Types in Sun UNIX", USENIX Conference Proceedings, Atlanta, Georgia, (Summer 1986)

[4] McKusick, M.K., Joy, W. Fabry R., " A Fast File System for UNIX", ACM TOCS, 2,3, August 1984, pp 181-197.

[5] Kozlow, J.D., "CD-ROM File System Performance Results", Internal Sun Document

Rethinking The Information Security Paradigm For Workgroup Computing

Christopher J. Riddick
Simpact Associates, Inc.
12007 Sunrise Valley Dr.
Reston, VA 22091
(703) 758-0190 x2156
uunet!nss1!cjr

Rethinking The Information Security Paradigm For Workgroup Computing [*]

Christopher J. Riddick
Program Manager
Simpact Associates, Inc.

Abstract

Workgroup computing requires information security that accommodates workgroup members' needs to access restricted data on a task-specific basis. This requires rethinking the existing information security paradigm which presents obstacles to workgroup members efficiently accessing such data in *need-to-know* cases. This paper looks at the challenges of implementing information security for workgroups. It offers a workflow model, based on current developmental work in a UNIX[1] environment, of how security can be accomplished.

The Workgroup

For the purposes of this paper, a **workgroup** shall be defined as any collection of users performing functions with a single objective as the end result. For example, the staff of an engineering project could be considered a workgroup with the end result of their efforts the successful completion of a specific project. As another example, temporary associations of people formed when a purchase order is filled out can be considered a workgroup. The requestor completes a purchase request and forwards the form to a manager for approval. Management verifies the need and signs the form. It is then passed to purchasing who first assigns a formal purchase order number and verifies that funds are actually available for the purchase. The purchase order is sent to a buyer who locates the best price and makes the purchase. When the purchased item arrives at shipping and receiving, it is logged as received and sent to the requestor specified on the purchase order. The invoice is sent to accounts payable for payment to the vendor.

There are several goals of **workgroup computing**. One goal is to facilitate the performance of the tasks within a workgroup. This may be through the automation of some or all functions, and it may also be through the management of workgroup information. Another objective of workgroup computing is to make information readily available to the members of the workgroup. As information is required for a workgroup task, the system must be able to provide the information in a timely manner.

[*] Copyright 1990, Simpact Associates, Inc.

A name frequently used for workgroup computing is cooperative computing. This implies that members of the workgroup are cooperating on tasks to reach a common goal. It is important that workgroup members do not duplicate effort or waste time when operating on the workgroup information. Workgroup computing should help to make it possible to perform workgroup tasks in an efficient manner.

Review of Information Security

Mayerfeld and Troy identified three states for information in conjunction with their model for risk management: *storage* (both operative and stored data), *transfer* (data in transit), and *transformation* (active programs) [1]. These states successfully characterize the forms information may take in an information system.

Security is defined to be the procedures and mechanisms used to satisfy the three security criteria for information in an information system: *confidentiality*, *integrity*, and *availability*. Confidentiality is the degree to which information is protected from harmful disclosure. Integrity is the protection of information from unauthorized modification. Availability is the guarantee that the information is available when needed.

A *security violation* is any event that causes a harmful disclosure, an unauthorized modification, or a loss of availability of the information protected by the system.

The Concept of Workflow

A definition of workgroup has been presented which specifies functions performed by a collection of users. This is not simply a list of functions to be performed by the workgroup. Rather, a workgroup may have associated with it a set of **workflows** which define not only the tasks to be performed, but the order in which they are to be performed, the subjects who will perform them, and the data on which they are to be performed.

A workflow is a sequence of activities performed in a predefined order. Each activity consists of a function performed upon a set of objects under the direction of a single subject. A function takes the information system from an initial state to a completion state such that arrival at the completion state automatically invokes the next activity in the workflow. A workflow is complete when all activities have been completed and the system has arrived at a final state.

Functions may be performed on a monolithic host, or they may be performed in a distributed processing system on client or server nodes. The key principle is the adherence to a well-formed transformation of specified objects from one state to another.

To specify a workflow, it is necessary to identify each state through which the information system will pass during the workflow. For each state transition, a function to accomplish the transition must be identified. A responsible subject associated with the function is also required. The subject may actually perform the function, or it may authorize the system to perform the function on its behalf.

Every workgroup has a set of workflows associated with it. The workflows are specified by profiles created by the system administrator. The **workflow manager** is an automated function residing on a workgroup server that mediates access to, and invocation of, the workflows. Subjects are restricted to the set of workflows which they can invoke, and the workflow manager ensures that 2 subjects are identified, authenticated, and authorized before permitting them to invoke workflows.

When a subject is required to perform a specific function during a workflow, the system grants the subject the necessary permissions to complete the function. No action on the part of the system administrator or the subject is necessary to gain the needed access rights to data objects required for a given function.

The workflow manager must be trusted to enforce the workflow profiles stored in its database by the workflow administrator. Also implied is the authentication of subjects to the workflow manager and to the file servers containing the objects. Objects are stored on file servers without security labels. The authentication does not determine the access to the object. The workflow profile defines the access rights of subjects to the objects based upon the current state of the workflow.

Access rights are dynamic. As the workflow progresses (as the workflow manager changes state), access rights may change in accordance with the profiles.

The Security Problem

A workgroup can be a static or a dynamic grouping of people participating in a common task. Workgroups transcend normal organizational boundaries. Workgroups can involve people from different administrative departments within a company. This variety in composition of a workgroup is a key benefit of workgroup computing. Unfortunately, it is also a definite problem for information security.

Traditional security models presume that access control using a lattice model meets the needs of a computer system. The lattice model places users (subjects) down one side of a matrix and data (objects) across the top of the matrix. The elements of the matrix are filled in with attributes defining access rights of a subject to a given object. The Bell-LaPadula *-property enforces these access controls by prohibiting a subject from writing to an object of a lower security classification than the subject itself [2]. Most security controls in computer systems and networks use some variation of the *-property in enforcing the system's security policy.

It is the premise of this paper that the Bell-LaPadula model is inadequate to meet the information security requirements of workgroup computing. Specifically, that model of security is based upon rigid security levels, and the computing system is structured to allow only authorized subjects access to the information. On the surface, this appears to be a sound approach. Upon reflection on the concept of workgroup computing, the model falls short of two important goals of the workgroup: making required information readily available to those who need it, and eliminating duplication of effort.

The security challenge is to protect information while allowing workgroup members efficient access on a *need-to-know* basis when required by a task. The result is a need for an information security approach that is dynamic to accommodate each workgroup member's changing need for information.

The concept of personnel clearances is not relevant in this case. In essence, all people are cleared for all information, but access is based upon need-to-know. Information is not labelled. No classification or categories are associated with the information.

An example shows how a traditional information security approach is inadequate for a workgroup. A personnel manager stores sensitive resumes on a file server labelled for restricted access by the personnel department only. A sales manager needs to hire a new salesperson and wishes to review resumes. The sales manager does not have access rights to the sensitive resume database. Under the current security model, either the sales manager must be given access rights to the resumes, or the selected resumes must be manually copied to a location to which the sales manager has access. This manual copy must be accomplished by a trusted third party to ensure that only the required resumes were copied to the new location, resulting in wasted effort.

The sales manager has only a task-specific need to see resumes, not an on-going reason to access resumes or other information (i.e., salaries, reviews) maintained on the personnel file server. The personnel manager does not want to grant permanent or unrestricted access to the resumes. Workgroup information security needs to accommodate the sales manager's task-specific requirement for direct access to the resumes while still protecting the confidentiality and integrity of the personnel database.

The Clark-Wilson Integrity Model

Clark and Wilson introduced the notion that Bell-LaPadula was inadequate to meet the objectives of commercial information security in [3]. Clark and Wilson present the concept of ***well-formed transactions*** and they state that ***separation of duty*** must be based on the control of subjects' access to these transactions. They introduce **Constrained Data Items (CDIs)**, which are the data items within the system to which the integrity model must be applied. **Integrity Verification Procedures** (IVPs) confirm that all of the CDIs in the system conform to the integrity specification at the time the IVP is executed. **Transformation Procedures (TPs)**, corresponding to the principles of the well-formed transaction, change the set of CDIs from one valid state to another.

To maintain the integrity of the CDIs, the system must ensure that only a TP can manipulate the CDIs.

Not all data is constrained data. There may also be data not covered by the integrity policy which are subject only to discretionary controls. These data are called **Unconstrained Data Items (UDIs)**. UDIs are important because they represent the way new information is entered into the system. Certain TPs may take UDIs as input values, and may modify or create CDIs based on this information. For example, information typed by a user at the keyboard is a UDI; it may have been entered or modified arbitrarily.

Clark and Wilson go on to define a set of rules to enforce this security policy. These rules are specified below:

C1: (Certification) All IVPs must properly ensure that all CDIs are in a valid state at the time the IVP is run.

C2: All TPs must be certified to be valid.

That is, they must take a CDI to a valid final state, given that it is in a valid state to begin with. For each TP, and each set of CDIs that it may manipulate, the security officer must specify a "relation", which defines that execution. A relation of the form: (TPi, (CDIa, CDIb, CDIc, ...)), where the list of CDIs defines a particular set of arguments for which the TP has been certified.

E1: (Enforcement) The system must maintain the list of relations specified in rule C2, and must ensure that the only manipulation of any CDI is by a TP, where the TP is operating on the CDI as specified in some relation.

E2: The system must maintain a list of relations of the form: (UserID, TPi, (CDIa, CDIb, CDIc, ...)), which relates a user, a TP, and the data objects that TP may reference on behalf of that user. It must ensure that only executions described in one of the relations are performed.

C3: The list of relations in E2 must be certified to meet the separation of duty requirements.

E3: The system must authenticate the identity of each user attempting to execute a TP.

C4: All TPs must be certified to write to an append-only CDI (the log) all information necessary to permit the nature of the operation to be reconstructed.

C5: Any TP that takes a UDI as an input value must be certified to perform only valid transformations, or else no transformations, for any possible value of the UDI. The transformation should take the input from a UDI to a CDI, or the UDI is rejected. Typically, this is an edit program.

E4: Only the agent permitted to certify entities may change the list of such entities associated with the other entities: specifically, the associated entity associated with a TP. An agent that can certify an entity may not have any execute rights with respect to that entity.

A Workflow Security Model

The previous definition of a workflow maps conveniently onto the Clark-Wilson Integrity Model. The objects of the workflow are equivalent to the CDIs. The workflow functions are the TPs. The workflow profiles are the relations specified by Clark-Wilson.

With this mapping of the integrity model over our workflow definition, we are able to use the rules defined by Clark-Wilson to determine the security mechanisms required for our workgroup to ensure the proper enforcement of workflow security according to the Clark-Wilson model.

Clark-Wilson specifies the requirement for **identification** and **authentication** of the subject before permitting the execution of a TP (rule E3). Rules C2 and E1 identifies the requirement to maintain a database of relations between TPs and CDIs. Rule E2 requires a database relating specific subjects to each of the relations between TPs and CDIs. We will call this the **authorization** database. Each relation in the database will be called a **workflow**. Finally, rule C4 requires an **audit trail** to which all TPs must append workflow reconstruction information.

An Application of the Workflow Security Model

A prototype implementation of the Workflow Security Model is under development by the author and his colleagues. The services provided to the workgroup consist of a trusted authentication server which also maintains the workflow database. The state of a given workflow is maintained by a workflow manager executing as a certified process in a secure server. Our prototype authentication server and workflow manager runs on an Intel 386-based PC executing SCO Unix V3.2 with the C2 security features activated.

Rule E3 authentication is provided via the Kerberos[2] authentication system [4]. The tickets used in the Kerberos protocol also serve as the vehicle for assuring that a TP will only be able to operate on a CDI within the constraints imposed by the authorization rule E2. File servers will not grant access to a CDI unless a ticket indicating a valid TP is presented according to rule E1.

Since all TPs must first receive a service ticket from the authentication system, it is possible to maintain a log of all transactions on the CDIs. Thus, the certification rule C4 is enforced.

The client-server processing model enforces the C2 rule in which a transaction takes a CDI to a final state before the next transaction may commence. Data and a function request is presented by the client to the server. The server only returns a result upon arrival at a final state of processing for the requested function.

Rules C1, C3, and C5 are enforced through offline audit tools and verification techniques. Rule E4 is enforced via a combination of authentication of the certification agent and an authorization database maintained by the authentication server. The certification agent is not authorized to perform TPs on CDIs (i.e., invoke a workflow). However, it can be assumed that the process of certification is, itself, a workflow which falls under the jurisdiction of the integrity rules and enforcement procedures. Therefore, the certification agent is only authorized to perform the certification process defined under the certification workflow.

The UNIX operating system was selected as the base platform for implementing the workflow manager and security services because of the existence of the de facto standard Kerberos authentication system and the ability of UNIX to handle multiple processes and communications protocols. Although a workgroup computing system could be implemented on a single host machine, the use of a network of clients and servers provides greater flexibility in the configuration of workgroups.

Since all transactions performed on the network are required to be well-formed and under the control of the workflow manager, the principal network security concern is the provision of a reliable, secure path between the client and the server. The authentication protocol used in our implementation provides a session key which enables the encryption of all packets between client and server. The UNIX-based workflow manager and authentication server can also enable the use of non-UNIX clients and servers such as MS-

DOS or OS/2 workstations.

Conclusions

Workgroup computing requires security that can be enforced on a task-specific basis which ensures the integrity and confidentiality of the workgroup data without imposing rigid access controls. Commercial applications of workgroup computing are driven primarily by the need to provide members of a workgroup access to the data they require to perform their function without the intervention of security administrators or the violation of access controls.

A model for **workflow** security was derived from the Clark-Wilson commercial integrity model and was applied to the workgroup environment using existing client-server applications and a de facto standard authentication system.

References

[1] Mayerfeld, H.N.; Troy, E.F. "Knowledge-based modelling of System Usage For Risk Management", *Proceedings of the 11th National Computer Security Conference*, Oct 1988.

[2] Bell, D.E. and LaPadula, L.J., *"Secure Computer Systems: Mathematical Foundations and Model"*, M74-244, Mitre Corp., Bedford, MA. 1973 (NTIS AD-771543).

[3] Clark, D. and Wilson, D., *"A Comparison of Commercial and Military Security Policies"*, IEEE Symposium on Security and Privacy, Oakland, CA, 1987, pp 184-194.

[4] Miller, S.P., Neuman, B.C., Schiller, J.I., and Saltzer, J.H., *"Section E.2.1, Kerberos Authentication and Authorization System"*, Project Athena Technical Plan, Massachusetts Institute of Technology, 1988.

1. *UNIX is a trademark of AT&T Bell Laboratories.*

2. *Kerberos is a trademark of the Massachusetts Institute of Technology.*

Why Isn't My Data Portable?

Michael J. Andrew
Digital Equipment Corporation
2465 Mission College Blvd.
Santa Clara, CA 95054
(408) 496-3481
mick@budgie.enet.dec.com,
uunet!decwrl!budgie.enet.dec.com!mick

Why Isn't My Data Portable?

Michael J. Andrew

ULTRIX Resource Center
Digital Equipment Corporation
Santa Clara, CA 95054
mick@budgie.enet.dec.com

ABSTRACT

This paper discusses the problems in porting applications between the various computing platforms available today. In particular, rather than discussing code portability, which is becoming reasonably well understood, this paper concentrates on the specific problems of data portability. Most emphasis will be placed on porting between various flavors of the UNIX operating system, but consideration will be given to VMS, MS-DOS and MVS. Although technical in nature, the paper has been written with the intent to be read and understood by programmers and non-programmers alike, and to explain why the seemingly simple idea of portable data and files is, in reality, not a simple matter at all.

1 Introduction

The problems of porting application code between various flavors of the UNIX operating system, and between different operating systems are becoming well understood. This is mainly because the programmers of the world have been required to move applications from one platform to another due to commercial market pressures. A prime example of this today is the migration of many PC applications to the UNIX market. An experienced programmer who has partaken of porting efforts usually has attained an "eye" for code segments which may cause problems when moved from one platform to another. It is certainly a fact that the creation of portable code is now seen as one of the major requirements of most new product specifications, even if that product is initially targeted for a single operating system and specific underlying hardware. Nevertheless, it is observed that issues of data file portability are not always subject to the same scrutiny as the issues of code portability. Now that networks are becoming commonplace, and shared files across heterogeneous environments are a reality, the difficulties in accessing those shared files are becoming more apparent, and strategies to deal with the problem must be found. It is both those difficulties, and some of the available strategies for overcoming them, which are presented below.

There are four broad approaches to the data portability problem, each with associated advantages and drawbacks;

- Ignore the problem
  This has the advantage of being simple and cheap, at least in terms of programming resources. However, it probably restricts the application to a particular platform.

- Write a data conversion utility
  The feasibility of writing a converter is proportional to the complexity of the data being converted. This is a useful method when data files do not need to be shared concurrently, or which are subject to a one time relocation or copy from one platform to another.
  Disadvantages are that a new converter must be written for each data file, or existing converters changed when the format of the contents of the file is modified. Also, one converter needs to be written for each one-way conversion between two platforms, which usually means a minimum of two programs for each file to be converted, in order to facilitate two-way transport of files.

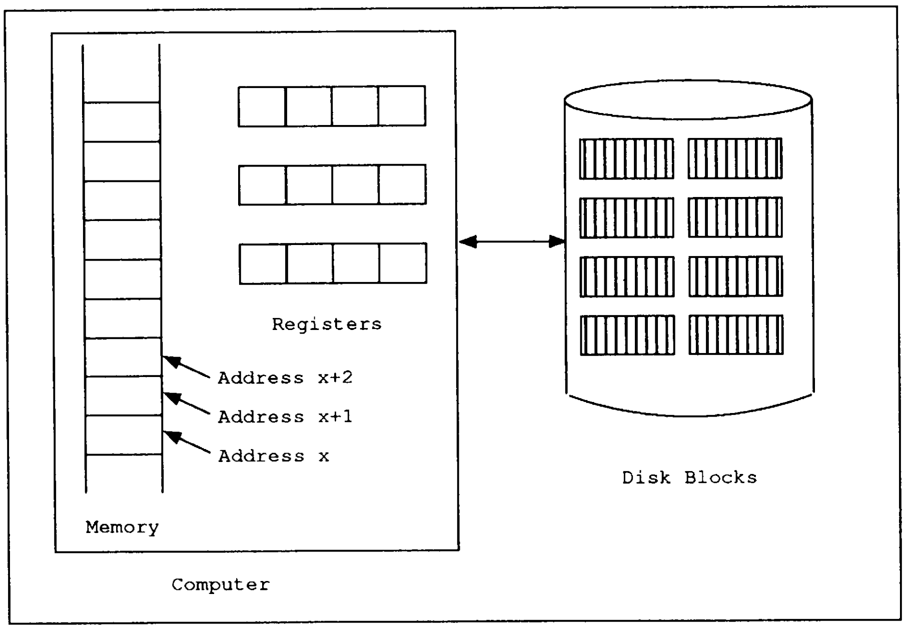
- Convert data "on-the-fly" at execution time.
  Increase the intelligence of the application to manipulate the content of the same data file no matter what the underlying hardware platform and operating system. This has the advantage of making the application data files portable over a wide range of platforms. The problem with this approach is that it is hard to implement in the first instance, and even harder to reverse engineer into existing applications. Furthermore, all future enhancements to the application must take into account the data portability considerations. Also, the costs in CPU cycles for on the fly conversion is paid every time data is read or written.

- Allow an external agent to take care of the data for you.
  This external agent is usually in the form of a database manager. This has the advantage of freeing the application from worrying about the data portability issues. Data is fed to the DBMS and is received in the format which is natural for the execution platform. Associated disadvantages of this approach are the learning curve for dealing with the DBMS, performance issues, cost, availability on your required platforms and the addition of an external agent into the confines of your application environment.

2 Disks and Files

2.1 Hardware basics

A hard disk is the usual repository for the data files we will consider. To the user or programmer, the data on disk is stored in *files*. Each file has one or more names by which it may be referred. It is up to the operating system to keep track of filenames, and the location of files on the disk. We shall not concern ourselves with the mechanics of this process. The format of the data as it physically resides on the disk is usually in equal sized chunks called *blocks*. Each block contains a fixed number of data items, the 1's and 0's which represent the data, called *bits*. Any transfer to and from the disk to the host computer is done by passing data blocks. There is also other data present on the surface of the hard disk, to perform tasks such as error-correction and rotational timing, but we shall not concern ourselves with these; we shall concentrate entirely upon the data which is visible from inside user applications. To speed throughput of the computer when dealing with data, the bits are grouped together and passed around the machine in parallel, and operated upon simultaneously. This grouping normally takes place in ordered sets of 8, 16 or 32 bits. The 8 bit quantities are called *bytes*. Historically, machines have been built with some number of bits in a byte other than 8, but they are becoming rare, and we shall not examine them explicitly. Because a byte is the smallest manipulable group of bits, the data blocks on disk always consist of a whole number of bytes. Data read from disk is stored in the computers *memory*. These bytes in memory are referred to by their *address*. For our purposes, this is a number which starts at zero and increases by one for each succeeding byte of memory. Thus a byte is the smallest *addressable* unit of storage. When the program wishes to operate on data in memory, the data value is usually copied from a specified memory address into special locations within the CPU, called *registers*. It is the size of the data path from memory to the registers which defines the machines *wordsize*. This is usually an even number of bytes Most of today's computers have a wordsize of 32 bits, or 4 bytes. Such machines are called *32-bit machines*. The wordsize of the machine is normally the determining factor of the size of integers which the machine can represent and efficiently manipulate. The figure below shows a simplistic representation of this arrangement. Each small rectangle represents one byte of storage.



2.2 Files and Records

So far we have described the physical aspects of the machine. Blocks on disk are all very well, but manipulating these would be very cumbersome for a user or a programmer. What we need is a logical structure imposed on those blocks. This structure is provided as a service of the operating system, and is called a *filesystem*. The data blocks are grouped together into separate objects which can be given names. Each group of blocks is called a *file*, and its name is called a *filename*. Now the user or programmer need not be concerned about individual blocks on disk, but is able to create unique files of information which can be accessed by name, both at the user lever or from within an application.

2.2.1 Filenames

It is the operating system which has complete control over the format of the names which are valid. It is also the operating system which defines the layout of the filesystem, be it a flat namespace (such as on MVS) or based on a hierarchy of directories (MS-DOS, VMS, UNIX). In the latter case, the way in which the hierarchy is represented in a filename is different. For instance

`/system/dir1/subdir/myfile.c`	in UNIX,
`C:\system\dir1\subdir\myfile.c`	in MS-DOS
`system:[dir1.subdir]myfile.c`	in VMS

are all equivalent names for "myfile.c" within similar directory structures. The case of the characters is significant in some operating systems, but not others. The following C language statements may or may not refer to the same file, depending on the underlying operating system or the implementation of the C library.

```
open("myfile.c", ...);
open("MYFILE.C", ...);
open("Myfile.c", ...);
```

If the language itself is also case-insensitive, then the problem is compounded.

The question of the meaning of special characters can have intriguing effects on the meaning of filenames from one platform to another. The file creation routine below,

```
        creat("C:\system\dir1\subdir\myfile.c", ...);
```

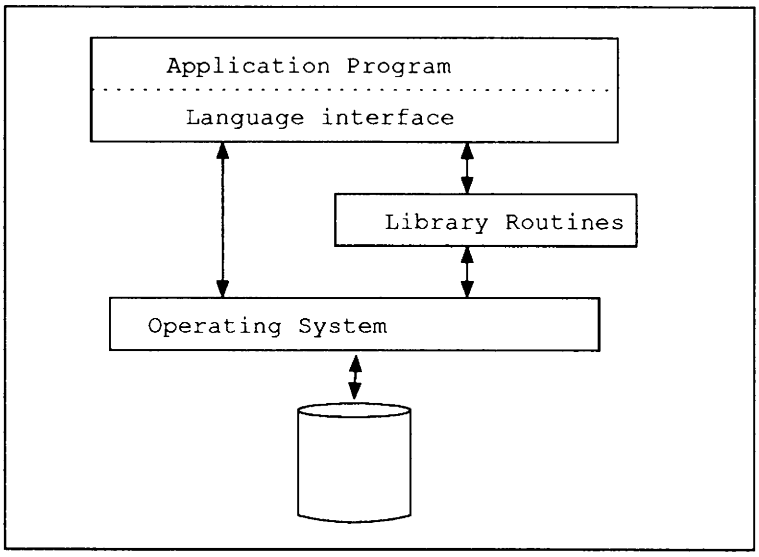
will create a file `myfile.c` deep in the directory hierarchy under MS-DOS, but will create a file called `C:systemdir1subdirmyfile.c` in the current directory under most UNIX implementations.

Problems are also caused by the different characters which are allowed in the various namespaces. UNIX will allow almost any character except '/' to be present in a filename. In VMS and MS-DOS only a handful of special characters are allowed, and the period character has a special significance in dividing up the filename. The length of the filename is subject to varying restrictions on different operating systems. This is particularly troublesome when porting between UNIX variants. In UNIX System V Release 2 and 3, filenames and directory names are restricted to 14 characters; excess characters are silently truncated.

2.2.2 Records

Just as the data blocks on disk are grouped together into files for the convenience of the user, the data blocks which comprise each file are not usually accessed as whole blocks. Instead, the data within them is accessed in many smaller groups of bytes called *records*. Each record is an ordered set of (usually related) data items, such as characters, integers, floating point number, etc. These items are called *fields*. In a single file there may be all the same kind (i.e. format) of record, or many different kinds; or there may be no records at all, just a long continuous stream of (literally) byte sized information. It is (usually) the programming language which defines and provides the link between records, files and the operating system.

Consider the following pictorial representation.



Note the explicit inclusion of a layer which is normally implied by default, namely the *programming language interface*. It might seem obvious, but it is worth emphasizing that the only interface the application programmer has to the outside world is that provided by the actual semantics of the language. This is a very important observation. Most languages define the I/O interface explicitly, as part of the language. For example

COBOL	`READ file`
Pascal	`readln()`

FORTRAN	`read()`
	`format()`

It is interesting that the C language does not define any I/O statements. The only interface C provides is that of the function call. It is up to the language implementors to provide callable functions which will access data files. (There is a well-defined *defacto* standard interface which is known as "standard I/O", and almost all C implementations come with a library which supports it.)

Different languages provide for different kinds of files. Most of these originated with the COBOL language, developed for solving business problems. The common kinds of files are described below.

2.2.3 File Types

A *byte-stream* file is one which has no underlying structure imposed on it by the language. It is simply an ordered set of bytes of any length. Once such a file is opened, any number of bytes may be read from it. After each read, the current file position is updated to point to the next byte after the last one transferred.

All the other file forms are based around *records.*

Sequential files come in various flavors, but are essentially an ordered set of one or more kinds of records. The records may be of fixed length, or of varying length. Usually the attributes of fixed or varying record length, together with the length itself, must be specified at file creation time. When a sequential file is opened, it is only possible to read linearly through the file, record by record. To go back and retrieve an earlier record, the file must be closed, re-opened and read again.

Relative files are collections of records which may be read sequentially, but which may also be accessed in a "random" fashion, usually by using the *record number.* It is thus possible to read the 4th record, then the 22nd, then the 8th, and so on.

Indexed files are collections of records which may be read sequentially, but may also be accessed in an arbitrary fashion using a special token of information, usually called a *key* or *index.* Depending on the operating system, the key may be a field within a record, or may be specified separately when the record is added to the file. A file may have several different keys, giving the programmer different orderings in which to retrieve the records.

The important thing to realize is that these filetypes are all artifacts of the programming language being used. It is up to the language implementors to provide the appropriate file access semantics and thus create the illusion (to a user or a program) of a file with an underlying structure. Being aware of the existence of this transformation is crucial to understanding many of the problems of file and data portability.

It is a combination of the programming language, callable library routines and the operating system which provide the translations between the two representations. Some operating systems (such as MVS , VMS and AOS) provide implementations of the various file types at the operating system level.. This makes things easier for the language implementors. Others, such as UNIX and MS-DOS, provide only a byte-stream interface. It is then up to the language implementors to provide libraries of code which implement the various file structures on top of the basic interface provided.

If the file structure is provided by the operating system, the disk blocks which are used to perform this filesystem and filetype bookkeeping are usually not accessible to the applications programmer. Such blocks are often only accessible by the operating system itself, or by specially provided low level access programs, for such things as disk image dumps (that is, a literal bit by bit snapshot a disk). As an example, for indexed files some operating systems (such as VMS) store the data records and key information together within the same physical file. Other implementations choose to make two files, a data file and an index file.

Herein lies a major part of the portability problem. Historically, it was the hardware vendors who provided both the compilers for various languages, and the operating system for their hardware. Although they each implement what is ostensibly the same source level version of a language, the actual format of the files underneath is usually entirely different. The representation of varying length sequential records created by a syntactically identical COBOL program on, say, Data General's AOS operating system is not the same as that on Digital's VMS. Even if the data file was physically accessible from one machine to the other, the content of the file would be meaningless to the other operating system.

On UNIX systems, the story is both better and worse. The UNIX operating system does not support any kind of file other than a simple byte-stream. There is no primitive OS operation to "fetch a record", because there is no concept of a "record". This might raise the question as to how such a bland interface can be useful in the world of commercial data processing? The good news is that the application is now free to impose its own structure on the data within a file. This ensures that the structure will be visible to the application no matter what UNIX system the file is accessed on. Furthermore, if the file is moved to a non-UNIX platform in which the operating system supports a byte-stream file format, then the same code can continue to access the file as on the UNIX platform. This is all well and good, but observe that it is now up to the application programmer to create and maintain the code to provide the file structure required. It does not need a great stretch of the imagination to see that supporting complex file structures such as variable length records, and (worse still) indexed files can greatly increase both the size and complexity of an application. Note also that if each application uses a different implementation method for file formats, then there can be no simple interchange of these files among different applications.

This has led to the existence of commercial product offerings which are libraries of file access routines which support the filetypes discussed on a variety of platforms, UNIX and MS-DOS in particular.

3 Data Fields

3.1 Representing Numbers and Characters

Now we shall turn our attention to the components of records, namely the fields. Each field represents some data item, and is composed of one or more bytes (fields smaller than one byte will not be discussed).

The eight bits within a byte can have a total of 256 different combinations. We could assign a numerical value to these combinations in any way we care to, but the logical assignment is to follow the mathematical base-2 (binary) numbering scheme

```
|0|0|0|0|0|0|0|0|     0
|0|0|0|0|0|0|0|1|     1
|0|0|0|0|0|0|1|0|     2
     .......         ...
|1|1|1|1|1|1|1|1|   255
```

It should be fairly clear how this scheme expands to larger integers, by grouping bytes together. For example, the number 1,000,000 is represented in 4 bytes on a 32 bit machine, as 0000000000001111010000010001000000.
We can represent numbers up to 4,294,967,295 in this manner. But what about representing text characters? For these we invent an alternative relationship between the bit pattern in each byte and single text characters. This gives us a possible 256 representable characters, which allows us to represent the alphabet in both upper and lower case, the digits 0 to 9, and a bunch of special characters, such as !@#$%^&. Some values are allocated to represent actions to do with the printed page, such as "new-line", "new page" and "carriage return". (Clearly we have a problem if we wish to represent more than 256 characters, such as in Asian languages. The discussion of the special requirements for such character sets is beyond the scope of this paper). Two different methods of encoding the character in a byte have emerged to be in common use today; *Extended Binary Coded Decimal Interchange Code* (EBCDIC) used mainly in IBM mainframe products, and *American Standard Code for Information Interchange* (ASCII), used by almost all other computer products (PCs and UNIX in particular). An example of the mapping of bytes in these two character sets is shown below.

	Numeric Value	ASCII character	EBCDIC character
0|1|1|0|1|0|1|1	107	k	,
0|1|1|0|1|1|1|0	108	l	%
1|0|1|0|0|0|0|1	193	*undefined*	A
0|1|0|1|0|0|1|1	053	5	*undefined*

It is crucial to see the difference between the (ASCII) character representation of numbers, such as the "5" above represented by the bit pattern 01010011, and the numeric integer 5 represented in a byte as 00000101, or in a 32-bit word as 00000000000000000000000000000101.

3.2 Integers

We have seen two ways of representing numbers; one using the mathematical base-2 value of bits in the byte, and the other using one of the character encoding schemes. Consider the number one million, represented as follows;

32-bit binary		00000000	00011110	10000100	01000000			
ASCII	byte values	00110001	00110000	00110000	00110000	00110000	00110000	00110000
	character value	1	0	0	0	0	0	0
EBCDIC	byte values	11110001	11110000	11110000	11110000	11110000	11110000	11110000
	character value	1	0	0	0	0	0	0

Notice that we need more storage to store the number in the text-based form than in its "natural" form. Because the mapping of numeric characters to the byte representation in somewhat arbitrary, calculations cannot be easily performed on the text form as it can on the mathematical form. Consider the sum 1000000 + 1 in the two representations

```
32-bit binary            00000000 00011110 10000100 01000000
                  +      00000000 00000000 00000000 00000001
                  =      00000000 00011110 10000100 01000001

ASCII                    00110001 00110000 00110000 00110000 00110000 00110000
                  +      00110000 00110000 00110000 00110000 00110000 00110001
                  =      00110001 00110000 00110000 00110000 00110000 00110010
```

In the binary version everything makes arithmetic sense, whereas the text version makes no sense arithmetically. It is simpler to design hardware to add the bits arithmetically than as text, and such hardware will also perform these arithmetic operation several orders of magnitude faster. This is why it is usually more efficient and preferable to use the natural representation for numbers in order to perform calculations.

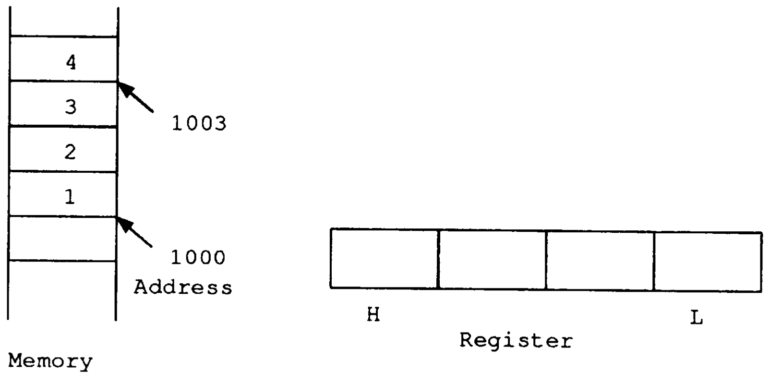
3.3 Representational Problems

In the examples above we have already introduced representational ambiguities. The first is caused by the multiple meaning (or overloading) of the value in a byte. If we look at a byte of data on a disk, does it represent a text character, or part of an arithmetic number? The truth is that it is impossible to know simply by examining data with no contextual information.

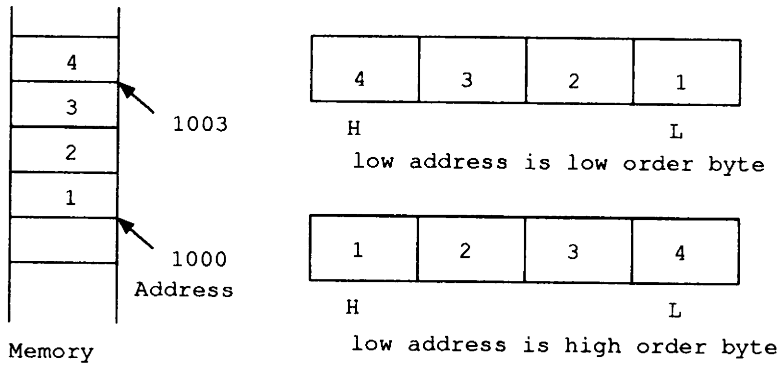
The second representational problem is that of multi-byte numeric values. Remember, each byte in memory is accessed by its unique address. If several bytes are conceptually joined together to represent a single object such as a number or a text string, we must decide which byte is to hold which part of that value, and what the address of the object should be. In the case of text strings or the textual representation of numbers, the characters are assigned to increasing addresses from left to right. The character string "Uniforum 1991" takes thirteen characters (we must include the space), with the "U" in the lowest address, and the final "1" in the highest. The digits are text characters, not a binary number. If the memory address of the "U" was 1000, then the address of the "1" would be 1012. The address of the complete thirteen character object is (by definition) the lowest memory address, or 1000.

Representing arithmetic numbers is not so obvious. We shall examine representing numbers on a 32-bit machine. Each different bit pattern of the 32 bits represents a different integer value. The range of number which can be represented is 0 through 4,294,967,295 (we need not consider negative numbers to illustrate the issues at hand). Our problems begin when we try to choose the order of the bytes within the word. This is probably the most well-known of the data portability problems, and certainly the most confusing; it is usually referred to as *byte-ordering*.

Consider a 32-bit word at address 1000; this contains 4 addressable bytes, at addresses 1000 through 1003. Suppose we represent the number we wish to load from memory in the following picture, where the value in each byte is simple represented as a single number. The address of the 4-byte word is the lowest address of its bytes, namely 1000. To make things clearer (hopefully!), rather than showing the bit pattern within each byte, we'll simply give each byte a numerical value, from 1 to 4 in ascending address order. Furthermore, we shall take liberties with the base-2 numbering mathematics, and pretend that the 4 byte values (1,2,3,4) together equate to the 32-bit number (1234). Do not confuse this with text character strings representing numbers, as in the preceding discussion.



We now have the question, should the value of "1" go in the left hand byte, or the right hand byte? The answer first of all depends on how we view the register picture. A natural way to do this is to consider the 32 bits within it as corresponding to the natural representation of a binary number. That is, the most significant bit is on the left, and the least significant bit is on the right. So the answer is, if byte 1000 is the low order byte it should go in position L, with 1 through 3 following it to the left, or if it is the high order byte, then it should go in position H, with the other bytes following to the right. Thus we have two answers, the number 1234 or 4321, depending on how our machine allocates integer words; low order byte at low order address, or high order byte at low order address

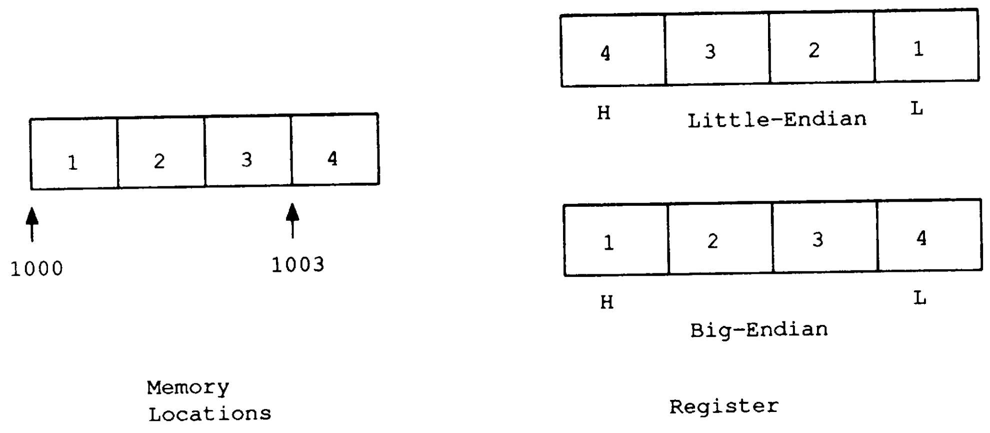


The unfortunate problem for the industry today is that both architectures are commonplace. If the low order byte goes in the low order address ("little" byte value at "little" address value), then the architecture is said to be *little-endian*. If the opposite is the case, then it is *big-endian*. Below is a list of the "endian-ness" of some of today's computers

Big-endian	*Little-endian*
68000	VAX
IBM 370	80x86
SPARC	MIPS
MIPS	

(Note that the MIPS processor appears on both lists. This is because the chip has a hardware switch to run in either mode. In itself this does not easily solve any portability problems for us though, as will become clear.)

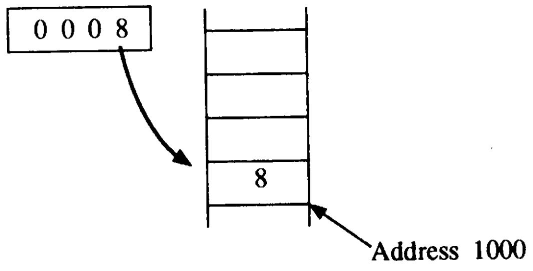
Let us now return to our example. Lets look at the two cases together, and draw memory increasing from right to left



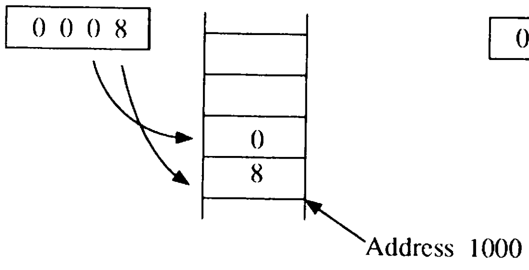
Now we see how the term *byte-swapped* arises; the little endian hardware appear to have "swapped" the bytes! On closer inspection however, we remember that the bytes in memory are pictured in order of increasing address from left to right. There is no particular reason to represent the memory this way, other that the fact that it seems more natural for most users of the Arabic numbering system. The computer does not understand such nuances, and if we draw memory increasing from right to left, then it would be the big-endian register which appeared byte-swapped.

From the example above, the reader may conclude that the big-endian representation feels more "correct" than little-endian. The following example shows an inconsistency of the big-endian representation, and helps provide a rationale for the evolution of both representations. It also exhibits the endian problem for 16 bit integers.

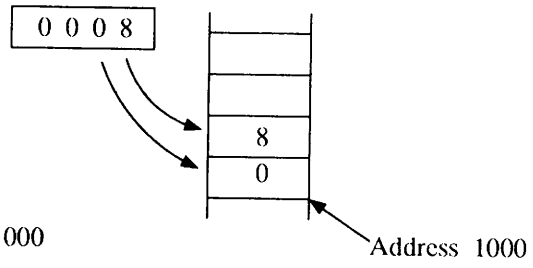
First consider the integer value 8, in a 32-bit register. To store the value as an 8-bit quantity at memory location 1000 is simple.



Now, suppose we wish to store the same value as a 16-bit quantity, also at address 1000. The picture below shows the answer for both little and big-endian.

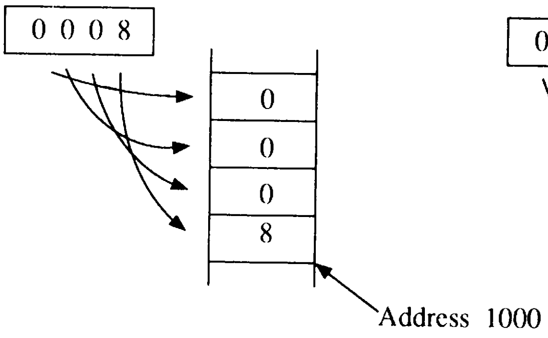


Little-endian

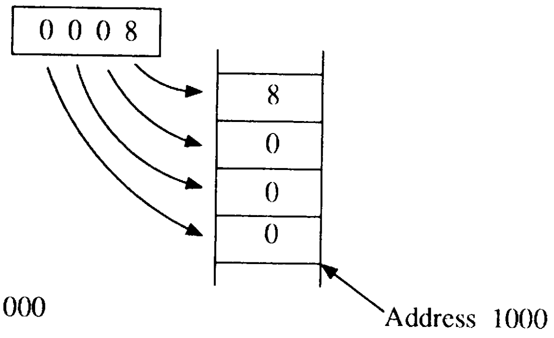


Big-endian

Finally, for a 32-bit integer,



Little-endian



Big-endian

Notice that in the little-endian case the bytes of the register always go into the same memory location relative to the start address of the numeric value, whereas for the big-endian case the memory order varies according to the size of the integer object.

The most important observation to make is that the phenomenon of byte swapping is caused by the loading and storing of data from memory locations to CPU registers. No "swapping" is taking place when loading data from disk files, or from network packets. When data is exchanged between memory and disk, the bytes remain in increasing address order. The next example will underline this point.

Consider a byte-stream file consisting of multiple records each containing two data fields; a text name field and a numeric data field (say, an employee number in a payroll record). We'll just consider a four character name for simplicity, and continue representing each byte of a numeric field with a single digit. (remember that the actual file is just a sequence of byte values. The names in the example are ASCII text strings, and the numbers are 4-byte numerical values, and *not* strings. To help reinforce this point, the numerical value of each byte will be represented as italicized digits).

Suppose our employee data is:

John	*1234*
Mick	*0042*
Eric	*2001*

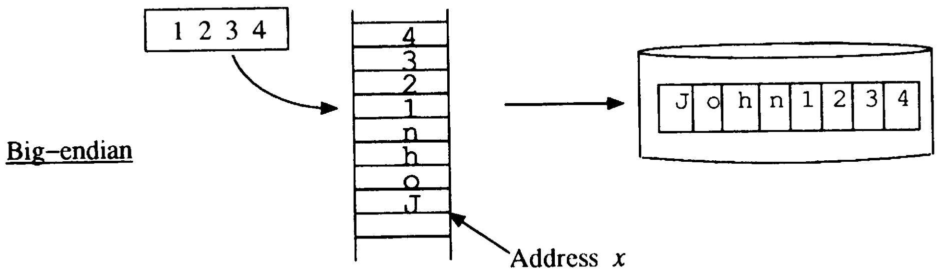
If we create the file on a big endian machine, the data in the disk file will be represented as follows:

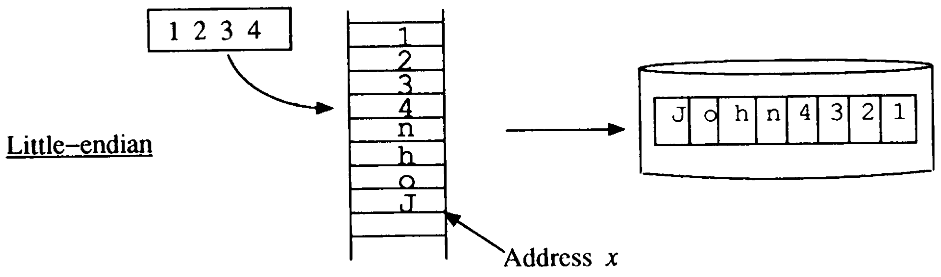
```
J o h n 1 2 3 4 M i c k 0 0 4 2 E r i c 2 0 0 1
```

On a little-endian machine the bytes will look like this:

```
J o h n 4 3 2 1 M i c k 2 4 0 0 E r i c 1 0 0 2
```

The following picture shows why. Imagine creating the data structure (as defined earlier) at some address *x*.





The files still represent the same logical data. But when created on two different endian machines, the files *are* physically different. When the text value "John" was loaded into memory, the "J" went into the lowest address, *x*, and the "n" into *x+3*. The numeric value is loaded at address *x+4*, and occupies the next 4 bytes. As we have seen, the ordering is machine dependent. To write out the record, we code a statement to write out the record to disk, and memory locations *x* to *x+7* are written out to the disk file A byte for byte comparison of the files will show a difference for bytes 4-7, 12-15 and 20-23. This is a real difference, and is the root of most data portability problems. If we take the big-endian file to a little endian machine and run a simple report program on it, the following data will be printed (it is interesting to note that exactly the same output will be produced on the big-endian machine, when given the little-endian file to work with);

```
John      4321
Mick      2400
Eric      1002
```

This is clearly a big problem in a heterogeneous environment, even when the operating systems are the same, as in the case of UNIX. In fact, in an environment where NFS is used to share files, this can cause serious problems if data files such as this are shared without regard for the underlying machine architecture.

If one were presented with this file and a description of its fields, but with the byte order information omitted, there is no way to know how to correctly interpret its contents; is John's employee number 1234 or 4321? This is precisely the problem with interpreting data in NFS mounted files.

3.4 Field Layout in Records

Another portability problem which is especially acute among UNIX platforms, is the issue of the layout of fields within records. Most languages require that the fields are ordered in the record in the same order as declared in the language statement. However, the actual placement of each field with regard to byte boundaries is often left by the language to be *implementation-defined*. This allows the language implementor maximum freedom to allocate the fields in either the most efficient form for speed of access on the particular platform, or for efficiency of saving space. The usual reason for different allocations is due to the addressing capability of the hardware. Some architectures do not support the addressing of a 32-bit integer on an odd byte boundary, or even on any address not divisible by 4. Consider the following C structure (32-bit machines):

```
struct  s {
            char      c;
            int       i;
};
```

The structure itself will normally be aligned at the most restrictive level, usually a 4-byte or 8-byte boundary. Depending on the hardware, there may be 0, 1 or 3 bytes of unused space, or *padding*, between the `char` and the `int`. In the case above, 0 pad bytes will result in the integer being on an odd byte boundary, 1 pad byte puts it at an even byte boundary, and 3 pad bytes puts it at a 4-byte boundary.

As a result, the same record declaration on different platforms can result in different record lengths. Even if the programmer has taken steps to ensure the code will port correctly to another platform, by using compile-time sizing constructs such as the sizeof operator, the structure padding problem is still present, and worse yet, not detected or flagged by the compiler, because no syntactic or semantic error is present. Consider the following C-like code fragment

```
while (records to write)
{
          write(fd, (char *) &rec, sizeof(rec));
}
```

If writing 10 such records as defined in structure `s` above, the records will be of length 5, 6 or 8, and the file created will be of length 50, 60 or 80 bytes. The problem with sharing this file is obvious.

4 Solutions

The intent of this paper is to exhibit the problems of data file portability, and in doing so answer the question posed by the title. There are many different ways to solve or work around these problems, and the detailed exposition of these are beyond the scope of this paper. It is reasonable, however, to give a brief description of the techniques available with which to attack some of the problems described; the next sections will attempt to explain these approaches.

We shall examine two of the solutions proposed in the introduction. One is the translation approach, where data files are transformed by a program (or programs) from the representation on one machine to the representation on another. The other we shall call the "intelligent application" approach, whereby there is a single copy of the data, and it is up to the application to interpret that data as appropriate. The techniques used by either approach are often identical. In particular, the translation program could be considered to be a program of the "intelligent application" class itself, performing operations on an input file, and writing data to an output file. Both have advantages and drawbacks.

4.1 Common data representations

Using the "intelligent application" approach requires that all data in files be stored on disk in a common format or representation. The choice of that format is determined by the answer to the following questions; Is it physically possible to convert from all platforms to the common representation? How expensive in terms of compute resources will that conversion be? Is there any loss of information in the conversion? The order of importance of these questions may well depend upon both the application itself, and the target environments being considered. For instance, financial traders would probably be somewhat upset if numerical information was rounded or truncated during a trip around different machines.

4.1.1 ASCII representation

One of the simplest forms of common representation is to use ASCII byte stream files. All numeric data, integers or floating point, are converted to the ASCII representation of the number. Different fields can be separated by a special character, or a fixed field length, and record delimiter characters can also be introduced. Using this technique yields files which are almost universally portable. The drawbacks of this choice are several; the size of the data can greatly increase over the "natural" representation; the time taken to convert between ASCII and the internal representation can become significant, depending upon the application.

Now there is no byte swapping issue, as the numeric values from memory are stored as sequences of ASCII characters 0 - 9 on disk. One can invent many alternative formats for the disk data. One could choose fixed length fields for the names and numbers, and thus save having to recognize the delimiting characters. The trade-offs must be balanced by the application designer. The ease of creating the output and parsing the input is one of the most important in this case.

A good real-world example of the use of this technique is found in many database implementations. The actual format of the data within the on-line database files is hidden from the user, and is normally implemented to best make use of the available hardware resources on the execution platform. However, when database data files are transported from one machine to another, they are usually translated to a special format, by a process commonly known as *exporting* the data. These exported data files are usually in ASCII format. This file can then be relocated to another machine and read in, or *imported*, with no loss of information. There are mappings form ASCII to EBCDIC representations, allowing these files to be passed from smaller machines to IBM mainframes and vice versa. Any execution overhead due to the translations in this process can be tolerated, as it is not an operation which is performed frequently.

4.1.2 Explicit Byte Order

Another common representation is to specify the byte order on disk, as being little-endian or big-endian, regardless of the execution hardware. A good example of this is the "network standard byte order" imposed in the header packets of the Internet Protocol Suite. All 16- or 32-bit numbers found in Internet network packet headers are in big-endian format, regardless of the originating platform. This is handy for big-endian machines, which can then pass on the data directly from memory data structures to the network. Little endian machines draw the short straw, and have to swap the bytes explicitly if the numbers are being generated within the application in their machine-natural form. Fortunately, this operation is relatively cheap.

To return to our employee example, we could choose big-endian as our intermediate format, and specified exactly four byte fields for name and number. The output file would then be in a well-defined format no matter what machine it was created on. Good programming practice would dictate that the actual conversion from internal to external representation be performed by a user-written library routine, for example:

```
strncpy(&emp.name, "Michael", 7);
emp.number = 42;
write_emp_record(&emp);
```

4.1.3 Self describing data

Another useful technique is to write data records in which each data item is preceded in some manner with information specifying the format of the following data. For instance, one could specify that each data field in a record is preceded by two 16-bit numbers; the first could specify the data type of the item, and the second its length. The conceptual layout of our employee record would then be

|*string:7*|Michael|*integer:4*|1234|

It is easy to see that there can be an infinite number of variations on this theme. The data type specification could potentially be expanded to include complex data structures, and even expanded dynamically. The disadvantage of this method is that it increases the size of the data to be exchanged, and impacts execution speed due to the encoding or decoding of the data stream.

It is an implementation of this scheme which is used for passing data in the Network Computing System (NCS) remote procedure call mechanism developed by Apollo, in the early 80's. In this implementation, there is a set of data type definitions which cover most of the basic data types, such as 16 and 32-bit integers in big-endian form or little-endian form, various floating point representations, and so on. When data is passed across the network, it is written in the natural form of the sending machine, leaving it up to the recipient to translate the data as appropriate.

4.2 Alignment of fields within data structures

Besides considering the format of each individual field, we must also address the layout of the fields in the record. The safest way to avoid problems with layout differences on different machines is to avoid writing out data to disk as exact copies of the data in memory. This means avoiding constructs such as

```
struct s employee;

write(fd, (char *) &employee, sizeof(struct s));
```

Unfortunately, the alternatives to to this very useful and common practice are awkward and inefficient, so the usual practice is to resort to the workaround of attempting to explicitly specifying the layout of the structure. This is achieved by defining explicit data values where the compiler (on some machine) may insert padding. This usually means ensuring that all data items are aligned at a byte boundary which is a multiple of the data object's size; a 32-bit integer must be aligned on a 4-byte boundary, a 16-bit integer on a 2-byte boundary, and so on.

For example

```
struct s {                                          struct s1 {
          char      c1;                                       char      c1;
          int       i;                                        char      pad1[3];
          char      c2;           becomes                     int       i;
          short     s;                                        char      c2;
};                                                            char      pad2[1];
                                                              short     s;
                                                    };
```

It is also interesting to note that the layout of the fields themselves can affect the total size of the structure. By arranging the structure as

```
struct s {
          int       i;
          short     s;
          char      c1;
          char      c2;
};
```

we incur no padding at all (or so we hope!). A heuristic for wasting the least space is to place the data items in the structure in descending size order. However, in a large data structure the grouping of related data items together may be deemed more important for readability and maintainability than saving a few bytes here and there. The example above indicates other deficiencies of this approach. The explicit padding makes the declaration less readable; the padding must be declared properly to ensure no further alignment is done by the host compiler; there is no guarantee that one day, some new architecture will align things differently, making it impossible to declare a structure which maps to the required alignment. In the real world, however, the benefits of adopting this approach are usually found to be sufficient to outweigh the potential drawbacks.

4.3 Translation Programs

For any execution platform, there is a "natural" layout for data; e.g. little-endian or big-endian, IEEE floating point format or some other, structure elements aligned on even boundaries versus some other alignment. There is also a "natural" file format, namely the one provided by the language and operating system of execution. It is clearly a simple task to read or write data in the natural format of the machine. The hard part is to read or write it in a non-natural form. This can require the programmer to jump through a number of hoops, many of which have just been described.

A translation program usually executes on either the data file source machine or the destination machine. Thus it is usually either reading the data file or the writing that file in the natural representation of the execution machine. This fact can be used to make the program easier to write, but it probably makes the program itself non-portable!

As the complexity of the data in the file increases, the conversion process too becomes more tricky. For instance, if the format of the next field, or next record depends on some data already read (or worse, still to be read, or in another file), then at some point the task of writing the utility raises to an equal level of complexity as writing the application itself. At this point, the only practical way to proceed is to fix the application to become "intelligent". We shall close this section with a particularly nasty example of how context dependent data can occur in two common languages, using a `union` in C, and a `REDEFINES` in COBOL.

```
struct  {
    union {
          char name[4];
          int  value;
          } item;
} rec;


01  REC.
  03  ITEM1.
          05 NAME    PIC XXXX.
  03  ITEM2 REDEFINES ITEM1.
          05 VALUE   PIC 9(9)    USAGE IS COMP.
```

In this example, the data stored in the C union `item`, or the COBOL ITEM1 or ITEM2, occupies the same bytes of the record. One of the items is a character string, and the other a machine dependent integer. Only if the data value in this record is an integer should the bytes be swapped when transferring to an opposite format machine. To determine whether this is the case may be very hard indeed, or even impossible for practical purposes.

5 Conclusion

This paper has posed the question as to why data files are not easily portable from one machine to another. We have seen that there are many different answers to this question, and that any particular file may contain more than one of the problems described above. Although the problems are many, it has been shown that several solutions and workarounds do exist. This is confirmed in the computer marketplace itself, where many applications are capable of exchanging the same data files from platform to platform, or concurrently accessing shared files in a network of heterogeneous machines.

The architecting of portable data files is best done when an application is designed, and even then there are traps for the unwary and uninformed. Hopefully this paper has gone some way towards ensuring the reader will be aware of the traps and informed of the solutions.

System Administration Using Artificial Intelligence

Elsie L. Yip
NCR Corporation
9900 Old Grove Road
San Diego, CA 92131
(619) 578-9000

System Administration Using Artificial Intelligence

Elsie Yip
NCR Corporation
Systems Engineering - San Diego
9900 Old Grove Road
San Diego, CA 92122

1.0 Introduction

The decade of the 1990's will bring drastic changes to the information processing environment. A key element to this transition is the move from an environment that consists of closed, proprietary, vendor-specific systems to open, vendor-independent systems, communicating with each other via industry standard protocols.

While this change offers the user total flexibility to incorporate and integrate different technologies to cater to the specific needs, effective system administration becomes the critical element to success.

In the closed, proprietary environment, a system administrator acquired the skills through the training provided by a specific vendor. In the open, multi-vendor environment of the future, a system administrator requires training from different vendors, on multiple disciplines. In addition, the administration of the interconnections between the numerous hardware and software components further adds to the complexity of the task.

Making matters worse, today's technology is evolving at an unprecedented pace. Knowledge acquired can become obsolete even before training has been completed. This makes it necessary for any system administrator to "refresh" his or her skills constantly. From the business perspective, the penalty in productivity for constant training and the threat of lost investment through attrition all add to the calamity.

Hence, the challenge to the system administrator of the 1990's can best be described as a constant challenge of rapid knowledge acquisition. This paper examines how the artificial intelligence technology can bring a remedy to this challenge.

2.0 Basic Concepts on Artificial Intelligence

The field of artificial intelligence began in the 1940's. Similar to other disciplines of computer science, the objective of this technology is to facilitate the task of problem solving that can be more productively handled by computers than humans.

While other computer science disciplines solve problems through the manipulation of data, artificial intelligence solves problems by the manipulation of knowledge. Though both data and knowledge represent information, there is a fundamental difference between the two.

The information contained in data is passive and self-contained. Problem solving by data manipulation is achieved by algorithms built external to the data. The information contained in knowledge is active and includes the information on how the different items of data logically relate to each other.

The difference between data and knowledge can be easily demonstrated in the problem of selecting a candidate among a number of job applicants. The data describing each candidate's educational background and job history provides a passive description of the candidate's credentials. However, the data by itself is insufficient to solve the hiring problem without the knowledge of the recruiter.

The additional information required to solve this problem lies in the recruiter's thought process. Based on the understanding of the qualifications needed for the specific position, a recruiter will selectively look for the pertinent data from the applications (often subconsciously), correlate the information to compare with the requirements to see if there is a fit.

In general, knowledge can be represented by a continuum: everyday common sense on one end of the spectrum, clearly defined facts on the other end, and judgement in between. Judgement is the elusive knowledge that comes with the accumulation of experience based on past decisions. Most problems solved by humans will involve all three facets of knowledge. While facts are black and white, judgement and common sense are often depended upon to resolve most problems, which involve some unknown and uncertainty.

Artificial intelligence provides a framework to efficiently acquire, organize and retrieve this knowledge for problem resolution. In many cases, an explanation can also be provided to justify the thought process and information applied to reach resolution.

Artificial intelligence has been successfully applied in a number of areas: natural language processing, speech recognition and expert systems. The concept of expert systems is further explored in the following section.

3.0 Expert Systems

Expert system applications address problems that require knowledge from human experts. The applications emulate the problem solving process of human experts: problem isolation, searching for relevant information to support problem classification, matching the optimum solution to the problem, and verification of the fix.

Expert system can significantly increase the availability of the expertise, providing up-to-date knowledge when and where it is needed. By encapsulation of the knowledge from a human expert, it also prevents the loss of expertise due to attrition or retirement in an organization. The knowledge from multiple experts can also be consolidated and validated, and integrated if it pertains to different fields.

An expert system application consists of two independent components: the knowledge base and inference engine. The knowledge base consists of the data, rules and techniques to solve the problem. The inference engine consists of the control mechanism to traverse the knowledge base to come up with the solution. Some inference engines also have uncertainty factors built into them, so that each conclusion drawn has a confidence level associated with it.

Though their designs are closely tied, the inference engine and the knowledge base remain two independent entities. This independence allows the knowledge base to be updated quickly and easily without altering the basic architecture of the inference engine. This attribute is especially desirable to applications that require rapid updates, as in the knowledge base required for system administration.

The advantages of the independence of the two components can be demonstrated by an expert system application for hardware diagnosis. When a new peripheral device is added to a system, the only update required is adding the pertinent data on the new device's characteristics to the knowledge base. The inference engine and the rest of the knowledge base can remain intact.

4.0 Development of Applications

From the late 1950's through the 1970's, languages to build artificial intelligence applications required unique training on programming techniques, which was often described as a brain washing exercise. Both development and execution of these applications demanded so much system resources that they required specialized, dedicated, heavy-duty mainframes to run efficiently. Hence, applications that were developed in that period were limited to the resolution of highly complex problems with very narrow scopes, e.g. geological survey.

By the late 1980's, with the advent of more powerful computers and development tools, expert system applications can now be easily developed and run on a variety of low to medium range of computers, including personal computers. This significantly reduces the investments required to develop and support these applications. The ability to run expert system applications on general purpose platforms further integrates the use of expert systems with other disciplines of data processing

Many of these tools include an intuitive, graphical user friendly interface to reduce the training necessary for development, as well as user interaction. The progress in this area has significantly reduced the intimidation of this technology on the user community.

The combination of lower hardware cost and ease of development has brought about a proliferation of expert system applications in the 80's, as demonstrated by Figure 1.0.

In the coming decade, not only is the dramatic growth in the number of expert system applications anticipated to continue, the scope of these applications will also be vastly diversified to be integrated into the day to day information processing environment. The use of expert system applications may become a de facto, standard component in data processing, similar to the availability of spreadsheet and word processing applications in the business environment.

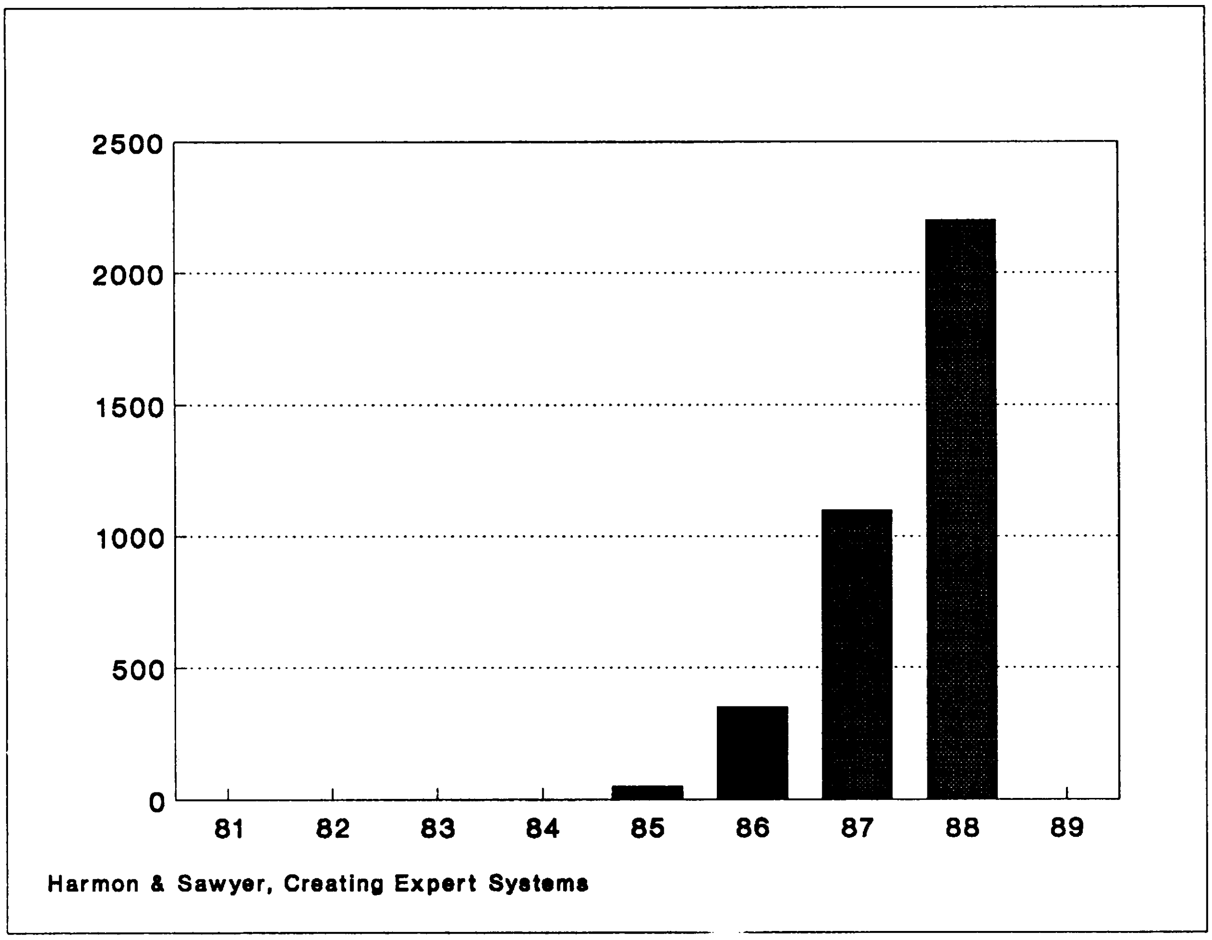


FIGURE 1.0 FIELDED EXPERT SYSTEM APPLICATIONS

5.0 System Administration Functions

As discussed in previous sessions, the information processing environment in the 1990's will make the role of system administration significantly more critical and complex.

Many of the system administrator's labor intensive tasks, e.g. file backup, can be and will be automated without the use of the artificial intelligence technology. This first step of productivity improvement will be necessary to free up resources to concentrate on the more technically oriented tasks which will require expert knowledge. In the rest of this section, an expert system application to assist a system administrator to achieve high system availability is examined.

System availability is one of the key requirements for information processing environment of the 1990's. Beyond the previous efforts to enhance the reliability of the system components through more stringent development process and quality control, the systems in the future will include self diagnostics for preventive maintenance. The ability to predict system failure so that corrective actions can be initiated to maintain high system availability, sometimes referred to as preventive maintenance, will soon become a mandatory system attribute.

While some system faults are totally unpredictable, many of them have identifiable symptoms before the eventual failure impacts the user. This provides a perfect opportunity for an expert system application.

An expert system application can achieve preventive maintenance by monitoring system error logs to detect the relevant symptoms, compare the nature and frequency of these symptoms to a predefined set of criteria to determine the what and when of the problem's potential occurrence. Built into this application is the capability to alert a system administrator of such potential faults. In addition, advice will be provided on how corrective actions can be taken to circumvent the problems.

In an extensively networked enterprise environment, requests for support can be automatically routed to the responsible party. Escalation of problem reporting to external support organizations can also be achieved. For example, an excessive frequency of retries to a disk may indicate that the device will soon faii. The local system administrator will be notified to begin backing up or moving files. In addition, the enterprise's system administrator or the external support can be notified to order a replacement device which can be installed in off hours to avoid impacting the users.

Similar to hardware faults, software (both kernel and application) faults that trigger system messages can be intercepted and deciphered to identify problem areas. Hence, instead of getting a system message on the system console, a system administrator can rely on the expert systems to assess the problem and make recommendations on corrective actions. As in the previous scenario, messages can be routed to the appropriate organization for effective management.

A third and more subtle indicator of system availability is system responsiveness. To maximize system availability, a system has to provide users with the needed resources in a timely fashion. A system running with degraded performance with poor response time is perceived to be marginally better than a system that is unreliable.

In this area, an expert system application can monitor and analyze the pertinent system performance statistics to provide a system administrator with timely information on the utilization of the various system resources. If a system resource is consistently utilized beyond an acceptable threshold, the expert system application can provide recommendations to the system administrator on actions to be taken to relieve the bottleneck. By carefully managing system resource utilization, severe response time problems can be avoided.

In addition, based on the same performance statistics, the expert system application can offer advice on capacity planning and support what-if analysis that can help the system administrator project future needs.

Figure 2.0 gives an overview of such an expert system application for system administrator.

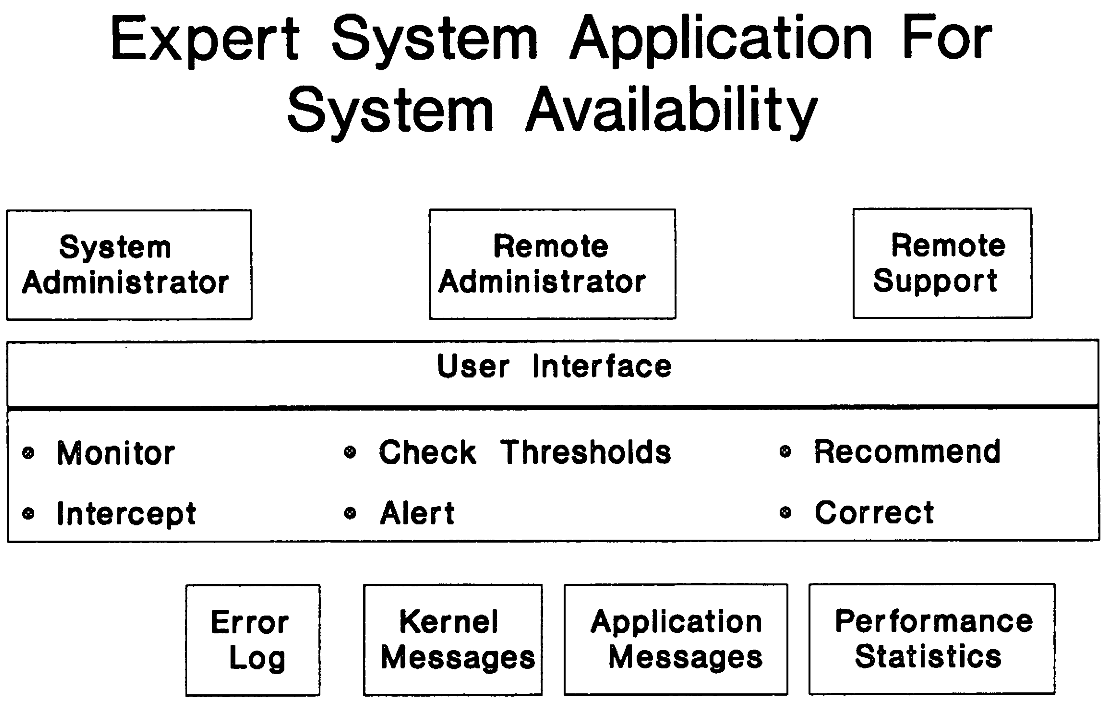


FIGURE 2.0 Expert System Application For System Availability

6.0 Conclusion

In this paper, the opportunity to use the expert system technology to assist a number of system administration tasks was explored. The 1990's will see a beginning of integration of artificial intelligence with the other components of information processing. It will eventually provide the framework to build knowledge into an integral part of any enterprise system, redefining the boundary between the user and the machine.

A Comparison of Network Queueing Systems

David Wright
Hewlett-Packard Company
3404 East Harmony Rd.
Fort Collins, CO 80525
(303) 229-6307
dww@hpfcla.hp.com

A Comparison of Network Queueing Systems

David Wright
Hewlett-Packard Company

Introduction

Many computer vendors today are touting Workgroup Computing and interoperability in a heterogenous environment. However, many of the capabilities they provide are limited and not practical for the end-user. The majority of tools provided are for the application developer, such as distributed computing environments, and the end-user must wait for applications to take advantage of networked environments. Users are hungry for tools that allow the various machines on the network to work together to solve a problem without requiring expertise on many different machines. Simple tools that provide access across the network to run a user program are a first step toward Workgroup Computing.

One such tool is a network queueing system that provides batch capabilities through the network. A network queueing system allows users to run programs in a "batch" manner, where the programs are usually run in the background rather than interactively, and jobs are queued when resources are not available. A network queueing system operates in a network of heterogenous computer systems, providing a common user interface which allows the user to access different machines without being confronted by the differences in operating systems and environments.

There are several network queueing systems in existence today that run under UNIX®, including PROD from Los Alamos National Labs, and MDQS from the US Army Ballistic Research Lab.* The two most prevalent, though, and available from major computer vendors, are NQS and Task Broker. NQS (Network Queueing System) is available in the public domain with proprietary versions available from Cray, Convex, and others. Hewlett-Packard offers a different system called Task Broker that provides network load balancing in addition to batch queueing. NQS is a de facto standard in the supercomputer world, while Task Broker is becoming entrenched in workstation/server environments.

The reason the two tools are prevalent in different environments stems from their design philosophies. NQS adds network extensions to a mainframe batch model, where the goal is to provide fair share scheduling of scarce resources. Emphasis is placed on managing local resources through the use of standard batch system quotas, with network extensions providing the routing of requests through the network. In addition, NQS offers a large number of user options that provide flexibility for different user needs on general purpose computers.

Task Broker, in contrast, is designed for a distributed environment of workstations and servers. The primary goal is to distribute user jobs to the network node best suited to run the job. The load balancing scheme is programmable, allowing users to match jobs with the most appropriate resources while taking advantage of the idle machine cycles available on the network. Instead of asking the user to specify options to control the job and data flow through the network, Task Broker transparently transfers the job and its data to a server, and transfers the output back to the client node upon completion.

User and vendor demand for interoperability between network queueing systems has led to the creation of an IEEE POSIX (Portable Operating System Interfaces) standards committee that is tasked with creating a standard for network batch systems. These efforts are on-going, but expect to generate a standard that supports users of either a centralized or distributed computing model. Originally NQS-based, the standard work now reflects the need for interoperability between all network queueing systems.

* UNIX is a trademark of AT&T.

This paper examines the similarities and differences between NQS and Task Broker. It shows why each system is best suited to a particular usage of network compute resources, and why each has become prevalent in that kind of environment. It further examines the on-going work of standards committees in the area of network queueing systems, and describes the current status of these efforts.

The Need for Network Queueing Systems

Due to the volatility of both technology and vendors in the computer industry, many computing environments have evolved into a heterogenous mix of computer hardware, applications software, and operating systems. Users are faced with the need to integrate the various components into usable systems while protecting their existing investments. Computer vendors are responding with promises of distributed processing solutions that operate in mixed-vendor environments, but in many cases the technologies provide only part of the solution. These technologies are slow to gain acceptance for several reasons, including the lack of open systems standards, and the difficulty level associated with their implementation.

One reason for the slow migration to distributed computing environments is that the majority of tools available are designed for the application developer. Tools such as Remote Procedure Calls (RPC's) must be designed into an application to make effective use of network resources and data. Directory services that provide location and routing independence are services used by other tools and applications, not typically by the end-user. In most cases, the end-user must wait for the solutions to propagate from vendor tools into their applications. But end-users have immediate need for tools that allow the various network machines to cooperate to solve a problem rather than waiting for the full solution from application developers.

A initial step toward Team Computing is a set of tools that allow the user to make more effective use of network compute resources. UNIX® developers have built in commands such as *rsh* and *rlogin* (remote shell and remote login, respectively) which provide connectivity to other network machines, but UNIX® has not addressed such problems as selecting the correct machine, transferring application data to and from the selected machine, or conflicting resource needs of multiple users.

A network queueing system is a tool that provides batch capabilities across the network. Users run programs in a batch manner, where the jobs are usually run in the background rather than interactively. Jobs may run on network machines other than the local machine, and as the name implies, a network queueing system provides queueing of jobs when appropriate resources are not available. The primary difference between a network queueing system and mainframe batch systems of the 1960's and 70's is network access -- jobs may be processed on network machines other than the local mainframe.

There are two prevalent network queueing systems available today: NQS (Network Queueing System), and Task Broker. NQS, developed by Brent Kingsbury et al at NASA, is available in the public domain from COSMIC. NQS is the de facto standard for batch job entry in the supercomputer market, with enhanced, proprietary versions bundled with Cray and Convex machines. Task Broker takes network queueing a step further, providing programmable load balancing amongst available batch servers. Task Broker is offered by Hewlett-Packard on heterogenous workstation platforms, and is becoming quickly entrenched in workstation/server environments.

What is NQS?

NQS (Network Queueing System) is a UNIX®-based queueing mechanism designed to support both batch (non-interactive) and device (eg., printer, plotter) processing. It was developed at NASA to support processing on a heterogenous set of networked UNIX® machines, so the user could see a single interface across the different architectures. Primary design goals included[1]:

☞ Support processing of shell scripts requiring only CPU resources and a command interpreter (eg., */bin/sh*),

- ☞ Support processing of device requests (eg., line printer),
- ☞ Provide resource quotas for batch requests and queues,
- ☞ Support remote queueing and routing of requests in a network,
- ☞ Provide access control for queues.

To support these goals, NQS implements three basic types of queues: batch queues, device queues, and pipe queues. Batch queues accept and execute batch requests; device queues accept and execute device requests; and pipe queues transfer requests to other batch, device, or pipe queues, usually at remote network machines. When a user submits a request, the submittal command specifies the name of the queue (an administrative default may also be used). For example, a batch request may be submitted to a local batch queue for processing, or it may be submitted to a pipe queue for subsequent transfer to a remote queue for further processing.

NQS treats batch queues and device queues differently. Batch queues are assigned resource limits to manage fair usage of limited resources. A given server may have multiple batch queues, each with differing limits on file space, memory, CPU time, core and data segment sizes, etc. The user submits to the appropriate queue matching the job's resource needs. Device queues do not have resource limits, but rather a set of associated devices. The administrator sets up a queue-to-device mapping, and the user specifies the queue for a device request.

Pipe queues provide the mechanism for transferring either batch or device requests to other queues on (usually) remote machines. Pipe queues require the creation of network server processes to transfer requests and handle a multitude of possible failures. The destinations for a pipe queue are also allowed to reject pipe queue requests, similar to a batch or device queue rejecting a user's request.

The processing model for NQS is similar to the familiar batch model from the 1960's. The user submits a request to a specific queue with a set of options specifying execution limits, priority, etc. If the specified queue is a pipe queue, the request is propagated across the network until it reaches a batch or device queue. The queues that accept and execute requests reside at the server in a client/server model (the client and server may be the same machine). The specified queue (or the one at the end of the pipe queues) may accept or reject the request based on factors such as queue type versus request type, queue resource limits, user access control, etc. Once accepted, the request is queued for processing, with each queue limited administratively to how many concurrent requests it may execute.

The program executed once the request is removed from the head of the queue is the shell script provided by the user in the submittal command. NQS assumes a user login exists on the serving machine, and executes the request shell script in the user's home directory in a selected shell. Standard out (*stdout*) and standard error (*stderr*) from the job are returned to the submitting directory upon completion. The shell script may also contain NQS options to set submittal options (eg., queue the job later), execution limits (eg., maximum CPU time), and user notification (eg., send mail upon job completion).

While the user may set execution limits on individual jobs, the NQS administrator sets up batch queues each with their own associated limits. For example, a queue may be assigned limits for job *nice* values, per process CPU time limits, and program size limits. The user may specify more restrictive limits than the queue defaults, but requests specifying less restrictive limits are rejected. Both inter-queue and intra-queue priority mechanisms are provided, with all entries in a higher priority queue completed prior to execution of entries in a lower priority queue.

Access control within NQS is also controlled by the administrator, who can set individual queues as unrestricted, or limited access by user or group. A user may submit a job to any unrestricted queue, or to any queue where the user is in the user or group access list.

What is Task Broker?

Task Broker is also a UNIX®-based network queueing mechanism. It was developed as an internal tool for Hewlett-Packard by Gary Thunquest, and is now offered as a product on multiple workstation platforms. Task Broker was developed as a tool for an integrated circuit design group to offload compute-intensive simulation and verification jobs from designer's workstations to the best available network resource. These resources might be a set of compute servers, or perhaps idle workstations. Task Broker was designed with the goal of selecting the "best" network resource for a job. Primary features include[2]:

- ☞ Load Balancing: Task Broker provides programmable mechanisms for brokering the selection of the best resources to achieve network load balancing.
- ☞ Transparent Data Access: Task Broker transfers files to and from the selected server, or performs Network File System (NFS) mounts of client file systems to the selected server.
- ☞ Fault Tolerance: Task Broker understands what to do if the network goes down, or if either a client or server machine goes down.
- ☞ Distributed Queueing: Task Broker provides a distributed queueing mechanism with no critical point of failure or centralized queue bottleneck.

Task Broker provides network load balancing through a programmable bidding mechanism. When a user submits a job request, each network computer that might be able to run the job is asked to submit a bid based on its current ability to execute the job. Each bid may be either a pre-defined constant, or the output of a program specific to a server and service that calculates a bid based on dynamic factors. The highest bid is selected, and Task Broker transparently transfers the job and its data to the selected server for execution.

Task Broker uses named services rather than the user-defined shell scripts required by NQS. Within the Task Broker environment, there are "producers" of services (servers) and "consumers" of services (clients)[3]. The user at the client machine provides Task Broker with an installation-defined name for an application in the submittal command. For example, a software developer might request a "*cc_service*" to run a large compile (user-defined programs can easily be run in a "*shell_service*"). Task Broker translates the name into a set of possible servers, and requests a bid from each member of the set. At each potential server, Task Broker decides whether to actually bid on the job based on a number of factors such as current load, number of jobs already running, available resources, etc. If a Task Broker server decides to bid, it runs a pre-defined program for the named service to calculate the bid (the bid may also be a pre-defined constant). The bid is returned to the Task Broker client machine that originally received the user request, where the highest bid is selected.

Upon selection of the "best" server, Task Broker makes the appropriate data files available at the server where the service will execute. Since the server selection process is dynamically provided by Task Broker, data access must also be provided automatically and transparently. Task Broker transfers any files requested by the user to the selected server, and performs any NFS mounts requested by the user. The user specifies files to be transferred to the server prior to execution, files to be transferred back to the client machine after execution, and any file systems to be mounted. Task Broker automatically selects the server, and performs the file transfers or file system mounts requested by the user.

The server selection process provided by Task Broker is essentially a greedy algorithm that selects the best network computer to run a job at that time. This approach results in increased network resource utilization, and therefore increased overall job throughput. When a job can not be immediately placed (the number of client requests exceeds network server capacity), the request is queued at the client machine. When a Task Broker server completes a service, it solicits queued requests from the various clients, and selects the request it is best suited to run based on the bidding programs.

Task Broker differs from NQS in its queueing strategy by having requests queued at the client machines from which they were submitted rather than at the server machine where they will execute. The purpose behind the distributed queues is to avoid central bottlenecks and potential single point failures. If a server becomes congested or unavailable, Task Broker, through its normal job placement process, routes requests to other server machines without user intervention. The only jobs affected by a server crash are those currently being executed.

Task Broker relies on direct communication between client and server machines when negotiating job placement and transferring data. The advantage of such a network protocol is that it supports robustness. Task Broker has built in recovery capabilities for cases when either the client, server, or network goes down. For example, if the server machine crashes while transferring files prior to job execution, the client will retry the transfer, and eventually select another server if the machine does not reboot.

Differences Between NQS and Task Broker

As described above, NQS and Task Broker are both network queueing systems that provide job execution across a network of heterogenous machines. Beyond that point, though, they are quite dissimilar. The primary emphasis within NQS is to provide background processing on fully utilized machines. The role of the queueing system is to prevent jobs from monopolizing scarce compute resources. This is achieved through prioritized sets of queues, each with associated execution limits (CPU time, program sizes, etc.). Although NQS works well in many environments, it has primarily taken root in environments with centralized computing resources that fit this emphasis. For example, supercomputer vendors such as Cray and Convex have extended NQS to provide greater control over specialized resources[4][5], and NQS is the de facto standard in the supercomputer arena.

In contrast, Task Broker has a smaller emphasis on compute resources, and a smaller set of execution limits associated with its services (eg., CPU time). Instead of preventing jobs from using scarce compute resources, Task Broker looks to leverage the unused compute cycles on the network. It views the network as a heterogenous set of compute resources that require load balancing, and the role of the queueing system is to match each job to the best available resource. This is achieved through the programmable bidding mechanism described above. Task Broker works well in many environments but is best suited to:

- ☞ A set of client machines accessing a pool of high-end servers which require load balancing.
- ☞ A distributed environment of specialized computers, where users need occasional access to specific resources.
- ☞ Achieving as much concurrency as resources permit in the running of a set of similar jobs such as Monte Carlo simulation, variation of parameter analysis, or large *make*'s.

Task Broker is quickly becoming entrenched in the workstation arena where these types of distributed environments are most likely to be found. For pools of high-end servers, selecting the best compute resource is usually based on machine load and number of tasks being serviced, with the overall goal of load balancing the server pool. For specialized compute resources, the algorithms used to select the best resource may be based on resource availability, easiest access to data (move the program to the data rather than the data to the program), or organizational reasons (eg., sharing an expensive server with other groups). For concurrent tasks, the distribution tends to be among groups of similar workstations, recovering unused compute cycles by running the tasks in the background at reduced priority.

Another major difference between NQS and Task Broker is their approach to user control. NQS has a large set of options available on the command line to allow the user significant flexibility in controlling where and how the job will be run. For example, on the submittal command line (*qsub*), NQS allows the user to specify the queue where the job is to be placed, when to run the job, what to do with *stdout/stderr*, what the execution limits are, when to notify the user via mail about job status, and what shell to use to interpret the batch request shell script. With this large set of options, and further extensions by supercomputer vendors, the user has control over the job, including where it runs and how the queueing system will manage it. The implicit intent is to provide user flexibility for a wide range of needs.

Task Broker takes a different approach, offloading control from the user to the system itself. In order to reap the benefits of transparency[6], the Task Broker philosophy is that the network and its division of resources should be hidden from users, in a similar manner to virtual memory masking the boundaries between primary and secondary storage. Task Broker assumes that selecting the best network resource for a job is the responsibility of the system, not the user. Not only does the algorithmic approach result in higher overall throughput, but it relieves the problems a user faces when working in a large, distributed compute environment. The user is no longer faced with knowing and understanding the myriad of network resources potentially available for a task.

Task Broker also provides an important addition to the NQS functionality: it provides the file transfers or file system mounts required for a job to run on another network node. While NQS requires the user to transfer data either within the request shell script (assuming appropriate privilege for a *rcp* or similar command) or external to NQS prior to submittal, Task Broker assumes that getting the data there is part of the system's responsibility. Task Broker supports file transfers in both directions between client and server, and also automatically mounts client file systems using NFS on the selected server.

Arguments abound as to which is the better approach: extended user flexibility versus transparent access. Those espousing user control are quick to point out that there are always reasons for a user to select the network server for a job (the user can access more information than a pre-programmed approach). For optimal performance of the queueing system on a single job, the user many times will be able to make a better decision.

Those in favor of transparent access generally point out that the algorithmic approach results in throughput gains not on a single job, but on all jobs submitted to the system. In addition, knowledge of the network need not reside with each user, but can be programmed into the system. This is especially important for novice computer users who simply use the computer as a tool (eg., a mechanical engineer using design and analysis tools), and do not want to understand UNIX® and its workings. Even experienced users will have some difficulty keeping abreast of changes made throughout large networks of heterogenous computing resources.

Other functionality differences exist between NQS and Task Broker. As previously stated, the two systems employ different queueing models. NQS maintains centralized queues at server machines, while Task Broker uses queues distributed at each client. Distributed queues make it much harder for a user to see when a particular job will run next, but they are an effective mechanism to handle server failures. NQS includes administrative commands[7] to allow requests to be moved from one queue to another to help manage this problem.

Less apparent, but important differences lie in the networking capabilities of each queueing system. The NQS emphasis is on managing queues and jobs on a specified computer. For example, NQS requires a user to track where on the network a job is in order to get its status. Task Broker, on the other hand, maintains job status locally on the client due to its direct communications between client and server. Having the queueing system manage job status is easier for the user, but Task Broker will not function in situations where direct communication paths are not available,

The most obvious difference in network functionality exists in the administrative aspects. *Qmgr*, the NQS administrative tool, executes on the local machine to manage queues. The administrator must log in to each network machine to manage or modify the configuration, using an extensive set of subcommands. Task Broker, in contrast, is configured by a configuration file read by the daemon at each node. Administrative commands to enable/disable services, show queue or service status, etc., may be run from any computer on the network (with appropriate privilege). Single point network administration like Task Broker is desirable in larger, distributed networks for ease of use. Management of configuration files that define the Task Broker offerings at each node, though, also needs to be provided from single network nodes to more effectively manage the network.

A final difference is access control and user mapping across network nodes. NQS administratively restricts queue access to sets of users and groups. While effective for user authorization (neither NQS nor Task Broker perform user authentication), this method plus the execution mechanism of running the user's login shell creates the need for global mapping of users across network nodes. While common in smaller networks, there are obvious limitations to such a mapping on a large scale basis. For example, current practice for a supercomputer offering cycles to users across the country is to set up a user id and and home directory for each user, or set of users, which becomes cumbersome for larger numbers of users.

Task Broker supports global mapping if it exists, or allows jobs to be run as a virtual user. While this obviates the need for global mapping of users, it may not be enough to provide adequate accounting on server machines. Task Broker does not execute user's login shells on the server; rather, it allows the configuration to define a path to any executable program for the service. This allows services to be either user programs or third party applications, and adminstrators can ensure that the service tasks they run are tested, "trusted" programs.

In summary, NQS and Task Broker are network queueing systems best suited to different environments. NQS has become the de facto standard for supercomputers because of its emphasis on resource and execution limits, and its flexible user interface. Task Broker is used more in larger, distributed environments where its load balancing and transparency are most effective. Both systems offer the capability to run jobs across the network in differing fashions depending on user needs.

Standards for Network Queueing Systems

One important criteria that should be considered when evaluating products for heterogenous environments is adherence to standards. Products that demonstrate compliance with standards offer the best opportunities for interoperability in a mixture of varied hardware and software platforms. Users interested in protecting existing and future investments do not want to purchase proprietary solutions available only on a limited set of platforms.

Standards for network queueing systems are currently in the development stage. Users may view this as a problem, because the likelihood of differing solutions (eg., NQS and Task Broker) is increased. However, it may also be viewed as an opportunity for users to develop standards that meet their needs.

The only standards work for network queueing systems this author is aware of is in the IEEE POSIX (Portable Operating Systems Interfaces) arena. Originally part of the IEEE 1003.10 Supercomputing Applications Environment Profile committee, the subcommittee started in 1987 to examine needs for batch processing for supercomputing applications. Standardized commands such as *&*, *at*, and *nohup* were quickly rejected as not meeting the needs of a full batch system[8]:

☞ Controlled processing of production work in a network of computers,

☞ Optimum usage of resources,

☞ Equitable sharing of resources,

☞ Simple user access,

☞ Sophisticated administration:

- Optimum workload scheduling,
- Load leveling across network,
- Automatic or manual operation.

After examination of existing queueing systems such as PROD, CTSS, and MDQS (the forerunner to NQS), the group settled on a standard base of NQS, with the goal of eliminating known deficiencies such as a "cumbersome and inconsistent administrator interface" and "limited networking functionality"[8].

The primary participants of this subcommittee expended significant effort into extending NQS into a more usable network queueing system. New commands were added, and efforts were made to integrate the various proprietary features of existing NQS installations. While primarily populated by supercomputer vendors, the group also included workstation vendors (eg., Intergraph, primarily interested in device queueing), and users (eg., NASA Ames, the originators of NQS). This group generated a draft standard based on existing NQS implementations, plus extensions desired by users.

The standard was modified many times during its development, and eventually a change from a standard base of NQS occurred for several reasons:

- ☞ NQS was found to be limited in ability to provide related services such as resource management, file transfer, and security "due to its rigid design and coding structure,"[9]
- ☞ The NQS network protocol was unable to handle the extensions required by the draft standard and vendor-supplied added functionality, and thus required redefinition,
- ☞ Hewlett-Packard introduced Task Broker, the first network product to match jobs with the most appropriate available resource. The group felt this functionality important considering the growth of networking capabilities, and wanted to ensure the standard did not preclude such a product.

A new model for a draft standard was adopted by the committee, which had spun off into its own POSIX group, IEEE 1003.15. The model[9] outlines a standard that is primarily concerned with command line interface and resultant action, and does not specify an underlying implementation (such as NQS). This model, and the subsequent standard under development, define the basic functionality required by a network queueing system while leaving vendors room to provide capabilities required by different environments.

The POSIX standard is under development, and optimistic predictions forecast balloting in the first half of 1991. Still under development are a networked administrative interface and the entire network protocol. Definition of a programmatic interface to commands, and resource control issues will be addressed in future standards[10]. Interested parties are welcome to attend and contribute at POSIX meetings.

Summary

This paper has compared two network queueing systems, NQS and Task Broker, describing and contrasting their primary features. Through these comparisons, it has shown why they are best suited to different computing environments. This paper has also described the on-going efforts in the standards arena for network queueing systems.

Tools such as network queueing systems are an important first step toward the realization of true Team Computing. While the future of distributed computing environments will probably evolve from tools developed for applications developers, there are usable, smaller steps that can be taken to help end-users better utilize network resources in the short term. NQS and Task Broker are just two examples of Team Computing in its infancy.

References

[1] Kingsbury, Brent, "The Network Queueing System", COSMIC Program # ARC-11750, August 1986.

[2] Thunquest, Gary, "Qless: A Queue-less Remote Task Spooling System", Hewlett-Packard internal documentation.

[3] Thunquest, Gary, "A Methodology for Building a Distributed IC Design Tool Using Qless", Proceedings of the 1988 Design Technology Conference, June 1988.

[4] "CONVEX CXbatch Concepts", CONVEX Computer Corp., Document No. 710-000003-201, February 1989.

[5] "Introduction to NQS", Cray Research Inc., Document No. SG-2018 C.

[6] Walker and Popek, "Distributed UNIX Transparency: Goals, Benefits and the TCF Example", Proceedings of UNIFORUM 1989, January 1989.

[7] "Network Queueing System", FORTRAN User's Guide, Multiflow Computer Corp.

[8] Evans, Dave, "Batch Standard", Cray Research Inc., Minutes, IEEE P1003.10, July 1989.

[9] "The POSIX Batch Model", Minutes, IEEE P1003.10, April 1990.

[10] "Scope for P1003.10 Batch Proposal", Minutes, IEEE P1003.10, April 1990.

A Study of Version Control Systems

Brian O'Donovan
Dept. of Computer Science
Trinity College, Dublin
Republic of Ireland
+353 91 51271
odonovan@cs.tcd.ie

Jane B. Grimson
Dept. of Computer Sciences
Trinity College, Dublin
Republic of Ireland
+353 1 772941
grimson@cs.tcd.ie

John Haslett
Dept. of Statistics
Trinity College, Dublin
Republic of Ireland
+353 1 772941
jhaslett@uaxl.tcd.ie

A Study of Version Control Systems

Brian O'Donovan

Jane Grimson

Dept. of Computer Science
Trinity College
Dublin
Rep. of Ireland

John Haslett

Dept. of Statistics
Trinity College
Dublin
Rep. of Ireland

ABSTRACT

RCS is one of the most commonly used version control systems under UNIX. We have developed a system called Distributed RCS (DRCS) which allows geographically dispersed users to have shared access to a common set of RCS files. In order to gain a better understanding of how to optimize the performance of DRCS we undertook a very comprehensive study which aimed at characterizing the typical pattern of accesses to a version control system. This paper summarizes the results of that study. These results are equally useful to the developers of any system which involves version controlled files.

1. Introduction

The authors of this paper have been involved in a project to build a distributed version control system suitable for use in Wide Area Networks.[1,2] The first release of this system has already been implemented. In order to make sensible choices about how to improve the overall performance of this system it was first necessary for us to gain some insight into typical access patterns to a version control system. Since there was no adequate data available in the literature we decided to perform our own study to determine the typical accesses to a version control system.

UNIX® is a trademark of AT&T
VAX®, VMS® and DEC/CMS® are trademarks of Digital Equipment Corporation
NSE® is a trademark of Sun Microsystems Incorporated

The main part of this paper is divided into four sections. We start with an introduction to version control systems. This is followed by a quick overview of Distributed RCS. The next section summarizes the results of the study which we performed in order to characterize the pattern of accesses to a version control system, it also compares out results with the results of previous similar studies. Last but not least we describe what future work we intend to do in this area.

2. Version Control

The UNIX file system does not support versions. This means that it is only possible to read the current contents of a file; the contents at any time in the past are lost once the file is written to. In some applications such as software development, it is vital to maintain a copy of old versions of the files. Since the UNIX file system does not support versions, many UNIX software developers use a specialized version control system. A version control system is a special type of database which is geared towards storing multiple versions of source files while they are under development. When the user "checks-in" a file to the version control system the current contents are stored in a special library file along with an identifying version number. The user can later retrieve any version of the file by performing a "check-out" on the appropriate version number. As well as storing the contents of each version the version control system will store additional information about each version such as the creation date and author. Many systems also have a feature called branching wherby two or more development paths (called *branches*) may be stored within one library file : typically branches are used to store distinct variants of the

The first version control system developed for UNIX systems was called the Source Code Control System (SCCS).[3,4] The initial system has been substantially enhanced and SCCS is still widely used. More recently a system called the Revision Control System (RCS) has been developed.[5,6,7] RCS has a much more user friendly user interface than SCCS and it is growing in popularity. Part of the reason for the popularity of RCS is the fact that it is available from the Free Software Foundation (GNU). Software developers working on VAX/VMS frequently use a package called DEC/CMS[8] which is essentially an enhanced version of SCCS that runs under the VMS operating system.

All three of these version control systems represent the stored versions in terms of the changes (*delta*) between this version and a neighbouring version. This delta storage scheme can result in significant savings in storage requirements as compared to storing each version explicitly. Both CMS and SCCS use a scheme called *merged deltas* which involves storing the change commands interspersed with the text itself. RCS uses a scheme called `reverse deltas` which involves storing the plain text representation of the most recent version and a seperate set of changes required to regenerate each of the old versions. The `reverse deltas` scheme is more efficient when retrieving the most recent version, but the `merged deltas` scheme is more efficient when retrieving an old version. A study[6] has revealed that the `reverse deltas` scheme has best overall performance because the most recently created version is retrieved much more frequently than any old version.

If a specialized version control system is being used, other applications must interact with it by performing explicit "check-ins" and explicit "check-outs. In general this entails making changes to the applications. A more recent trend is to develop an enhanced file system [9,10] which incorporates version management facilities. Existing applications can run unchanged and will always read the most recent version. Application wishing to access other versions of the file may do so by way of additional system calls.

Another recent trend has been for an increasing number of developers to use an Integrated Programming Support Environment (IPSE). Many of these environments provide a version control service as part of the overall package. Some of these systems call upon a specialized version control system to provide the version control service but others such as NSE[11] and DSEE[12,13] provide their own version control sub-system.

3. DRCS

The authors of this paper have developed a system called DRCS[1,2] (Distributed RCS) which is implemented as an enhancement to RCS. DRCS can be used to provide shared access to a common set of RCS files for a development team who are spread over a wide area network.

In addition to all of the standard RCS features, DRCS allows users to create replica files at remote nodes. The original DRCS file is the master replica ; all subsequent copies are slave replicas. Users may use any of the RCS commands on their local replica of the DRCS file and (apart from performance) they will not notice any difference from standard RCS. When a user requests a copy of a particular version of a DRCS file, the system will first look to see if that version is stored locally, if the version is not available locally it will send a request for the data to the site containing the master replica. All requests to update the DRCS file must be passed to the master site for synchronization purposes.

Because it is aimed at use in a wide area network, DRCS is very careful to minimize the amount of communication required. If a new version is created the system will notify all of the replicas about the new versions existence, however, the actual content of the new version will not be propagated unless someone actually requests a copy of it at the remote site. As an additional performance enhancement, DRCS will whenever possible transfer the new version in terms of the changes that need to be made to an old version. This is because transmitting the changes will require less data than transmitting the entire contents of the new version.

In a local area network it is normal to provide shared access to version controlled files by using a distributed file system. However, cost and performance issues preclude the use of a distributed file system in a wide area network. DRCS minimizes the communication costs by making intelligent use of replication. DRCS also minimizes the performance penalty involved in using a wide area network by allowing communication between nodes to take place asynchronously whenever possible. DRCS was developed with uucp type networks in mind but it can be easily configured to use any available communications protocol. In fact DRCS sites do not even require a direct link because DRCS can be used to share files between any two sites that have a mail connection.

One of the benefits of the way that DRCS is implemented is that it can be used to share files between two heterogeneous systems. It has been installed and tested on VAX systems running ULTRIX and Sun systems running SUN-OS, which communicate using both uucp and TCP/IP communications protocols. It should be trivially easy to install it on any other variant of UNIX.

A company named MKS has ported RCS to run on IBM PCs and compatible computers.[14] This software is available for both MS-DOS and OS/2 operating systems. We are currently investigating the possibility of extending this work by porting DRCS to the IBM PC so that DRCS can be used to share version controlled files between a network of IBM PCs and/or UNIX machines.

4. Access History

A number of researchers have studied the access patterns of conventional file systems. The results of this research have been very useful in helping the developers of file systems to optimize their design in response to "typical" access patterns. If the performance of version control systems is to be optimized there is a need for similar studies of the access patterns of version control systems. The following list summarizes the existing studies which have analyzed the typical use of a version control scheme.

- David Leblang's paper about DSEE[12] included some figures concerning the efficiency of the delta representation scheme used.

- Mark Rochkind carried out an analysis [3] of some SCCS files on an IBM 370 system. This study analyzed the size of the SCCS files, the number of versions per file and the efficiency of the delta representation scheme used.

- Walter Tichy analyzed the access histories of two RCS systems.[7] This study repeated some of the analysis that Rochkind had done but it also analyzed, the relative frequency of reads and writes, the frequency of access to old versions and the frequency of use of the branching feature.

None of the existing studies attempted to analyze some of the issues which are important with regard to optimizing the performance of a version control system such as : the time between subsequent accesses to a file, the extent to which files are shared between users, the interleaving of reads and writes and the extent to which there is a locality of reference describing which files are likely to be accessed next. In addition, the previous studies were limited to a small number of systems and did not asses the effect of different application environments.

We decided that there was a need for a much more extensive study of the access patterns of a version control system. To ensure a comparability with the earlier results published by Tichy we used the same access histories as he studied. In order to see if his results were also applicable in other environments we collected access histories from four other systems. Three of the systems studied used the DEC/CMS system and the other three used RCS. The six systems traces studied were identified as follows.

`tichy_one`	The record of the accesses to RCS files on the *`arthur`* node. This system was one of those studied in Tichy's earlier research. The application environment consisted of university researchers.
`tichy_two`	The record of the accesses to RCS files on the *`mordred`* node. This system was one of those studied in Tichy's earlier research. The application environment consisted of university researchers.
`development`	The record of accesses to the DEC/CMS files used by a commercial software development group.

laboratory	The record of accesses to the RCS files used by researchers at a commercial research laboratory.
sources	The record of the access to the DEC/CMS files containing the source code for stock control software used by a manufacturing site. These DEC/CMS files were accessed whenever maintenance was required on the software.
scripts	The record of the access to the DEC/CMS files containing the DCL routines (shell scripts) used for system management tasks at the same manufacturing site as used the **sources** system. These DEC/CMS files were accessed whenever the system management scripts were to be run (on a periodic basis) or whenever the routines needed to be updated due to a change in system management policy.

In total these access histories contained records of over 70,000 accesses to over 7,000 files. They represent a broad mix of the applications to which a version control system might be put. The complete results of the analysis are too lengthy for inclusion in this paper : hence they will be published in a forthcoming Ph.D. thesis by Brian O'Donovan. The following points summarize the important findings of this study.

File Size: We found that the average size of the most recent revision of the files in the **development** system was 15.7KBytes but the median size was only 4.5KBytes. We noticed a trend for files to grow in size between versions. Files grew an average of 1.6Kbytes (median 0.13Kbytes) between the first and second version.

Deltas: Representing the versions of a file in terms of their deltas is very efficient. The average overhead involved in storing all old versions in the **development** system is 27% of the space required to store the most recent version only : this compares with an overhead of 467% required to store all old versions explicitly. It is possible to store 10 versions of a file with an average overhead of only 40% more space than storing the latest version of a file. On average the delta representation of a file occupies only 22% of the space required to store all of the versions explicitly : this compares with 33% for a conventional Lempel-Ziv type compression.[15]

Versions: While there was an overall average of 3.75 versions per file, approximately half of all files had only one version. When files with only one version were excluded from the analysis we found that the average number of versions per file was 6.78.

Old Versions: On average 1.88% of reads from multi-version files (1.44% of all reads) were reads of old versions. However, there was a wide variation between systems with regard to how often old versions of files were read. Only 0.05% of reads to multi-version files in the **scripts** system read old versions, while 13.68% of reads to multi-version files (8.60% of all reads) in the **tichy_one** system read old versions. There was a certain consistency between the systems with regard to which old version was accessed : the version immediately preceding the current version was twice as likely to be accessed as the versions before that again.

Branches: There was also a wide variation between systems with regard to how often the branching feature was used. In the **laboratory**, **scripts** and **sources** systems the branching feature was never used and in the **development** system the branching feature was hardly ever used (less than 0.2% of all accesses were to branches). On the other hand 1-3% of all accesses to the systems analyzed by Tichy (**tichy_one** and **tichy_two**) were accesses to branch versions. Some informal discussions with users of version control systems revealed that most users avoided using the branching feature because they felt it to be too complex : the users of the systems analyzed by Tichy included the developers of RCS who were presumably very knowledgable about RCS and how to use the branching feature.

Read/Write Ratio: The read/write ratio averaged out at 2.21, however the ratio varied significantly from system to system. The **development** and **laboratory** systems had approximately the same number of reads and writes but the **scripts** system had approximately 10 times as many reads as writes. In no system was there significantly more writes than reads.

File Lifetime: While we could not make an accurate measurement of the file lifetime, there was evidence to indicate that the vast majority of files had an active lifetime in excess of the period of our study (almost 3 years).

Locality: There was a certain amount of locality of reference but there was no well defined working set. Approximately 20% of all accesses were to the file in the top position in the LRU stack, while approximately 30% of all accesses were to files in the top five positions in the LRU stack. There did, however, seem to be a large amount of sequentiality in the access patterns (i.e. files tended to be frequently accessed in the same sequence). On average 35% of times when a file was accessed for a second or subsequent time, the next file to be accessed was the same as the file which was accessed after it the last time.

Time between Accesses: We found that the probability distribution curve for the time between subsequent accesses to a file was a Weibull distribution [16] with a β factor of less than 0.5. This means that the likelihood of a file being accessed in the near future decreases rapidly as the time since the last access increases. This is consistent with the intuitive belief that the files accessed in the recent past are also likely to be accessed in the near future. The median time between subsequent writes to the same file varied between 50 days for the **development** system to over 6 months for the **tichy_one**, **tichy_two** and **sources** systems. The median time between subsequent reads to the same file varied between 1 and 3 days for most systems, the exception was the **sources** system where the median time between reads was 19 days (this may have been due to the fact that many of the command scripts were used on a monthly basis).

Write Read Interleaving: For many of the files the accesses alternated between writes and reads. This means that if the last access to a file was a write there is a 94% probability that the next access to the file will be a read. However the reverse is not always true, if the last access was a read this gives us no clear indication of whether the next access will be a read or a write.

Data Sharing: In general we encountered a very high level of data sharing although there was a variation in the amount of data sharing from system to system. In the **scripts** and **sources** systems, over 90% of the files were read by more than one user. The probability of a file being shared seemed to increase in proportion to the number of accesses to the file with an average of between 10% and 40% of all accesses (reads and writes) being by a user other than the primary user of that file.

We would like to compare our results with the results of the previous studies of version control systems. However, none of the previous studies looked at locality of reference for a file system or version control system, hence we will compare our results on locality with the results of a study by Kearns et al.[17] which analyzed the extent of locality of references to a relational database system. In order to highlight how references to a version control system are different from access to a standard UNIX file system we will also compare our results with the results of a study by Ousterhout et al.[18] which studied the accesses to various BSD UNIX file systems. This study is often regarded as the definitive characterization of typical accesses to a UNIX file system. The following points summarize the difference between our findings and the results of previous studies :

File Size: Tichy reported that the average size of the latest version of the RCS files on his system was 5.5Kbytes (5.9KBytes when single version files were excluded). This is significantly less than the average size of 15.5KBytes which we measured, however, the fact that the median size of our files was only 4.5KBytes probably indicates that a small number of large files are distorting our average. Rochkind measured the size of his files in lines rather than bytes, we could not therefore make a direct comparison with his results. Neither of these studied the growth of subsequent versions of the same file, however Tichy's observation that the latest version of multi-version files is on average larger than a single version file would tend to support our findings. Ousterhout reported that UNIX files have a median size of 2.5 Kbytes, this means that version controlled files are on average larger than the average UNIX file.

Deltas: Rochkind reported an average overhead of 37% required to store the deltas for old versions : since he also reported an average of 4.88 versions per file this works out at an overhead of 9.5% per extra version. Tichy reported an average overhead of 34% required to store the deltas for old versions : since he also reported an average of 3.10 versions per file this works out at an overhead of 16.2% per extra version. Leblang reported an overhead of 1-2% per version required to store the deltas for old versions, however his scheme also involved blank compression and it is not therefore directly comparable. While we found that an average overhead of 27% was required to store the deltas for old versions in the **development** system, we also found that the average overhead per extra version was inversely correlated with the number of versions in the file. We found an average overhead of 14.4% for one extra version, but an average overhead of only 7.7% per extra version for files with 5 versions and an average overhead of only 4.6% per extra version for files with 10 versions.

Versions: Rochkind reported an average of 4.88 version per file, approximately 40% of his files had only one version, when files with only one version were excluded he reported an average of 7.5 versions per file. Tichy reported an average of 1.39 version per file, approximately 80% of his files had only one version, when files with only one version were excluded he reported an average of 3.10 versions per file. The systems which we studied had an average of 3.75 versions per file, approximately 50% of them had only one version, when files with only one version were excluded we found an average of 6.78 versions per file. Our findings are closer to what was found by Rochkind than what was found by Tichy, however this is probably due to the relative maturity of the systems when they were studied (as time passes the number of versions per file tends to grow).

Old Versions: Tichy reported that 6% of reads to the RCS files he studied were reads of old versions. He also stated that he felt that accesses to old versions would be much more common in a non-academic environment. Our experience is the exact opposite, the systems used in a commercial environment had a much lower rate of access to old versions. Incidentally, our results give even stronger support to the use of reverse deltas as used by RCS rather than merged deltas as used by SCCS.[5]

Branches: Tichy reported that an average of 2-3% of all accesses involved branch versions. However, we found that there was practically no use of branching in any of the other four systems we studied.

Read/Write Ratio: Tichy reported a read/write ratio of 2.27 for the RCS files he studied. Ousterhout et al. reported a read to write ratio of approximately 2.34 for a conventional file system. Both these figures are pretty close to our average figure of 2.21.

File Lifetime: We could find no previous analysis of the lifetime of version controlled files. However, Ousterhout reported a median lifetime of approximately 3 minutes for files in a conventional UNIX file system. This seems to be in sharp contrast with the fact that the files we studied seemed to have lifetimes measured in years rather than in minutes.

Locality: Kearns et al. reported that there was a substantian amount of locality in a typical program's references to data in a relational database system. For example he measured that between 96% and 99% of all accesses to the database system were to the data blocks in the top two positions on the LRU stack. He reports very little sequentiality of reference. While we did find a certain amount of locality of reference we found that only 24% of references were to the files in the top two positions in the LRU stack. However, we also found that there was a substantial amount of sequentiality of reference.

Other Factors: We are not aware of any previous study that has analyzed the **Time between Accesses** to the same file, **Write Read Interleaving** or **Data Sharing**.

We intend to use the results of our analysis to build a simulation model which will assess the impact on the performance of DRCS of using various replication strategies. However, before we even build this complex model we can already see how some of the results of our study might influence how we adapt the replication control mechanisms of DRCS. By making these adjustments we can improve overall performance and hence reduce cost.

- The fact that files tend to be repeatedly accessed in the same sequence, means that the previous access history can be used to predict which file is likely to be accessed next. There might be a performance benefit in pre-fetching files based upon this prediction.

- The fact that the time between accesses follows a weibull distribution with a β factor less than 0.5 leads us to conclude that the least recently used files are also the least likely to be used in the near future. We could use this information to decide that certain files should have their replicas completely purged.
- It is wastefull of disk space to store replicas which will never be accessed. The fact that old versions are very rarely accessed might lead us to consider it safe to delete replicas of old versions since they are unlikely to be ever read.
- The fact that the branching feature is rarely used would lead us to conclude that it is reasonable to ignore branch operations when optimizing performance.

5. Future Work

DRCS has been implemented in prototype form and is available to anyone interested in trying out its capabilities. It is hoped that users will provide feedback about possible enhancements to the system. This feedback will be used in conjunction with the data about typical accesses patterns presented in this paper to generate plans for the future development of DRCS. Possible enhancements that have already been identified include

- Changes in the replication control policy
- Improvements in security
- Porting DRCS to IBM PCs

6. Conclusions

In recent years there has been much research into version control systems. As a result their concepts and potential benefits have become well understood. This should mean that version control systems will be more widely used in the future. The general improvement in communication technology has made it possible for a geographically dispersed group of developers to work together as one cohesive project team. It is therefore our opinion that geographically dispersed project teams will become much more common in the future. These teams will need tools that facilitate data sharing in a wide are network environment : the DRCS system described in this paper is such a tool and should be widely used in the future.

It is now becoming universally recognized, that the full benefit cannot be derived from a version control system unless it is well integrated with other tools such as a configuration manager. As a result there should be an increase in the number of integrated Programming Support Environments (IPSEs) [19,20,21] and enhanced systems interfaces (such as PCTE) [22,23] which will provide version management facilities as part of a much wider service. There will, nevertheless, still be a constant demand for traditional stand-alone version control systems both to handle non-standard applications and to be integrated with other utilities as part of a larger package.

As version control systems are likely to be widely used in the future it is important that their performance should be optimized. The research described in this paper has totally characterised the typical pattern of accesses to a version control system. This knowledge should help the developers of future version control systems in their efforts to optimize system performance.

7. Acknowledgments

The authors would like to thank the various members of the Department of Computer Science in Trinity College who provided helpful suggestions. We would also like to thank Walter Tichy who gave us access to the original RCS code and to the data he used in his earlier study. Last but not least we would like to thank Digital without whose financial support this research would not be possible.

References

1. Brian O'Donovan and Jane Grimson, "Development of a Distributed Revision Control System," *Proc. UKUUG Summer '90 Conference*, pp. 207-214, London, UK, 9-13 July 1990.
2. Brian O'Donovan and Jane Grimson, "A Distributed Version Control System for Wide Area Networks," *Software Engineering Journal*, vol. 5, no. 5, pp. 255-262, September 1990.
3. M.J. Rochkind, "The Source Code Control System," *IEEE Trans. on Software Engineering*, vol. SE-1, no. 4, pp. 364-370, December 1975.
4. Eric Allman, "Unix text file management tools," *UNIX Review*, vol. 7, no. 3, p. 72(6), March 1989.
5. Walter F. Tichy, "RCS : A Revision Control System," in *Integrated Interactive Computing Systems*, ed. P. Delgano and E. Sandwell, pp. 345-361, North-Holland, 1983.

6. W.F. Tichy, ''Design Implementation and Evaluation of a Revision Control System,'' *Proc. of 6th Intl. Conf. on Software Eng.*, pp. 58-67, Tokyo, Japan, 13-16 September 1982.

7. W.F. Tichy, ''RCS - A System for Version Control,'' *Software : Practice and Experience*, vol. 15, no. 7, pp. 637-654, July 1985.

8. Digital, *Guide to VAX DEC/Code Management System,* Digital Equipment Corporation, Maynard, Mass., April 1987.

9. Ellis S. Cohen, Dilip A. Soni, Raimund Gluecker, William M. Hasling, Robert W. Schwanke, and Michael E. Wagner, ''Version Management in Gypsy,'' *SIGPLAN Notices*, vol. 24, no. 2, pp. 201-215, February 1989.

10. David G. Belanger, G. David Bergland, and Mike Wish, ''Some Advanced Research Directions for Large Scale Software Development,'' *Bell Systems Technical Journal*, vol. 67, no. 4, pp. 77-92, August 1988.

11. Theresa Barry, ''Sun Microsystems shows CASE Network Environment,'' *Datamation*, vol. 33, no. 22, pp. 144-148, 15 November 1987.

12. D.B. Leblang and R.P. Chase, ''Computer-Aided Software Engineering in a Distributed Workstation Environment,'' *ACM Sigplan Notices*, vol. 19, no. 5, pp. 104-112, May 1984.

13. David B. Leblang and Robert P. Chase, ''Parallel Software Configuration Management in a Network Environment,'' *IEEE Software*, vol. 6, no. 4, pp. 28-35, November 1987.

14. Jim Vallino, ''MKS RCS 4.2.c (Product Review),'' *PC Technical Journal*, vol. 6, no. 10, pp. 132-135, October 1988.

15. Terry A. Welch, ''A Technique for High-Performance Data Compression,'' *IEEE Computer*, vol. 17, no. 6, pp. 8-19, June 1984.

16. James R. King, *Probability Charts for Decision Making,* Industrial Press Inc., New York, USA, 1971.

17. John Kearns and Samuel DeFazio, ''Locality of Reference in Hierarchical Database Systems,'' *IEEE Trans. on Software Engineering*, vol. SE-9, no. 2, pp. 128-134, March 1983.

18. J.K. Ousterhout, H. DaCosta, D. Harrison, J.A. Kunze, M. Kupfer, and J.G. Thompson, ''A trace-driven analysis of the UNIX 4.2BSD file system,'' *ACM Operating Systems Review*, vol. 19, no. 5, pp. 15-24, December 1985.

19. Maria H. Pendeco and E. Don Stuckle, ''PMDB - A Project Master Databse for Software Engineering Environments,'' *Proc. 8th ICSE*, pp. 150-157, IEEE, London, August 1985.

20. John Kador, ''Change Control and Configuration Management,'' *System Development*, May 1989.

21. Interactive Development Environments, *Software Through Pictures User Manual,* 4.0, Interactive Development Environments, Guildford, Surrey, UK,

September 1988.

22. Gerard Boudier, Fernando Gallo, Regis Minot, and Ian Thomas, "An Overview of PCTE and PCTE+," *Sigplan Notices*, vol. 24, no. 2, pp. 248-257, February 1989.

23. Ian Thomas, "PCTE Interfaces: Supporting Tools in Software Engineering Environments," *IEEE Software*, vol. 6, no. 6, pp. 15-23, November 1989.

An Object Model for Distributed Systems

James Waldo
Hewlett-Packard Company
250 Apollo Drive
Chelmsford, MA 01824
(508) 256-6600
waldo@apollo.hp.com

An Object Model for Distributed Systems

Jim Waldo, Ph.D.
Cooperative Object Computing Operation
Hewlett–Packard
250 Apollo Drive
Chelmsford, Ma. 01824
...decvax!apollo!waldo

Abstract

In this paper we describe a basic object model that allows the construction of distributed systems. The object model specifies how objects are identified, located, and how messages are sent from one object to another. Also discussed is how the system conserves resources by allowing automatic activation and deactivation of objects. We further show how certain parts of the object model are necessary for interoperability, while others can be changed. We discuss how we allow such changes to be made without access to the source of the system. Finally, we discuss the status of the project, and the directions for our future research in this area.

Introduction

Distributed systems promise a future in which a combination of machines of various power (and cost) can function together in a way that makes everyone more productive without requiring anyone to spend more on hardware than is necessary. However, programming distributed systems has turned out to be difficult and error prone, and hence the promise of distributed systems has yet to be fulfilled.

Object oriented systems have shown a way to cut down on software costs by allowing developers to make use of a natural design metaphor, reuse existing code, and cut down on the costs of changing and maintaining software systems. However, the object metaphor has yet to be fully exploited in the realm of distributed systems.

Before these two technologies can be combined, we need an object model that supports distributed computing. Most object based systems assume that communication between objects takes place in a single address space or at worst within a single machine. However, for true distributed systems to exploit object oriented techniques, the objects being used must be able to exist and communicate on different machines, perhaps with different architectures.

This paper will describe an object model that is designed to solve just such problems. The model is centered around supplying the basic mechanisms for object oriented programming in a distributed environment. We take those mechanisms to include:

Object identification;
Object location;
Sending a message from one object to another;
Activating an object (including restoring its state from persistent store)
    when needed to process a message; and

deactivating an object (including writing its state to persistent store) when the object is no longer needed;

In addition, we assume that such a system must be extensible, allowing the introduction of multiple different sorts of objects that correspond to the basic object model but that differ in other ways one from the other.

This paper will describe a system we are implementing that is based on an object model for distributed systems. The system, based on the Network Computing Architecture, is designed to allow objects residing on different machines (perhaps with different architectures) to identify, locate, and send messages to one another. Most of the complexity of the network and object schemes are hidden from the application programmer, allowing the building of applications without the usual complexities of distributed programming being introduced.

The paper will describe the identification and location scheme, and give details about how the messaging system is built on top of these. It will be shown how the system can scale from networks of two machines to those that connect all of the machines in an enterprise of as many as 10,000 machines. We will also discuss the requirements placed on any object model to allow that model to fit within the system, and show how object models that differ in ways other than those that form these requirements can be added to the system without changing (or, for that matter, even having access to the source of) the original system. Finally, we will show how the design allows different object models to be introduced and work within the overall system given that those object models meet a small number of constraints.

Object Identification

In approaching the construction of an object model that can be used for the construction of distributed systems, the first problem is determining the best way to identify objects in such an environment. Two paths to a solution immediately come to mind. On the first, each object is identified by a name, represented by an arbitrarily long string, that can be nested in a hierarchical fashion to insure uniqueness. The second approach is to assign to each object a unique identifier of some fixed length.

The first solution, that of naming, is initially attractive for a number of reasons. Names are a natural way of identifying objects, and are easy for the ultimate users of the system (people) to use. The namespace of arbitrarily long strings is large enough to deal with as many objects as we might wish. Further, the hierarchy that is used to name the object might also, at first blush, serve as an aid in the location of that object, thus allowing us to solve two of our problems with a single approach.

This initial attractiveness, however, does not bear serious scrutiny. While arbitrarily long names allow for a namespace that is potentially infinite, they also introduce serious complexity in interpreting those names. When passing identifiers from one machine to another in a possibly heterogeneous network, it is difficult to pass variable length parameters as identifiers. The first problem has to do with possible conflicts in character sets, making it difficult to insure that there will be any marked character that can be guaranteed as a terminator. Nor can one assume the ability to pass the length as the first parameter, as the integer formats can vary from one architecture to another. Further, the overhead involved in parsing arbitrarily long strings is not something that you want to have as part of your base identification scheme.

Nor does the promise of using the possible hierarchy in the naming scheme to help locate objects pan out. This would only work if objects were guaranteed not to move, and if machines were never taken out of the network. Neither of these possibilities are realistic. So the best that the name could provide would be a hint as to where the object might be in terms of where the object was created. While we will see in what follows that this can be a powerful aid in the locating of objects, it can be provided in ways that do not require the parsing of arbitrarily long strings in the location scheme.

The final problem with a naming approach to object identification is the problem of insuring uniqueness when two name domains are merged. Unless great care was taken before the merge, there can be no guarantee that the names in the two merged domains are disjoint. The only way to guarantee uniqueness is to change the names, perhaps prefixing them with a unique domain identifier, at the time of the merge. This, however, would require not only going through all of the objects in both domains and renaming them, but also going to the internal state of all objects in both domains to look for references to other objects and changing the names used in the reference. It is not clear that such a process would be possible; even if it were, it would be prohibitively expensive in both terms of time and computing resources.

Because of these problems, we took the approach of using fixed length object identifiers that would be assigned to objects on creation and used from then on to identify those objects. The problem that confronted us was to come up with a method that would generate an identifier that would be guaranteed over all time and space, of a length that would allow a sufficient number of identifiers to be created to suffice for the size systems that we envision over the next decade.

The problem was to come up with an identification scheme that could be localized to each machine (potentially in the world) but that would guarantee the uniqueness of the identifiers generated, even if those identifiers were generated by two machines that had never been in contact. The solution was to use Universal Unique Identifiers (UUID) as defined by the Network Computing Architecture (NCA)[1,2]. These identifiers are 128 bit, fixed length entities. The first 48 bits of the UUID are an identifier of the machine on which the UUID was generated; while the architecture is neutral on how this identifier is chosen, the obvious choice is to use IEEE network identifiers. This has the advantage of freeing the scheme from any particular hardware vendor, and using information that is likely available to the machine in virtue of it being part of a network.

Most of the rest of the UUID is used to store a time stamp, with the granularity of 10 milleseconds, that indicates the time of creation. This combination of machine identity and time of creation is, we believe, sufficient to uniquely identify the object. Note that this scheme does not require clock synchronization throughout the network, as the time stamp is relative to the machine.

There is the possibility, on this scheme, of two objects getting the same UUID. If the clock on a machine is set back, and an object is created at just the wrong time, the combination of machine–id and timestamp could appear a second time. To guard against that, we also have a sequence number in the UUID, that is incremented every time the machine is reset in such a way as to affect the clock.

It should be noted, by the way, that the result of having two objects with the same object identifier would not be catastrophic (although it would be hard to track down). The result would be confined to the two objects that had the same UUID, and the problem would be the impossibility of being able to distinguish between them. While such a result would be confusing, it would be totally localized.

Object Location

One of the reason for identifying objects is to allow them to be located no matter where they were created or subsequently moved within the system. We will now turn to how this location is accomplished in the system.

The base of our location scheme is a local location service, that is responsible for finding any object that exists on the machine in which it is running. Oversimplifying for a moment, this object (for in our system all services are provided by objects) keeps a database of all of the objects that exist on that machine. Included in this database is the location of the persistent data that is associated with the object, and the location of the code that can be used to manipulate that object. This object finding object listens for message requests on a well known network port on every system. Thus the problem of finding an object reduces to the problem of finding the object locater on that system.

Given the technique we have chosen to identify objects, we would need only minimal location support if we could assume that objects did not move from their machine of creation and that the set of machines on the network only expanded. Since the object identifier includes an identifier of the birth machine for the object, all that would be required in such a static situation would be access to a database that could map from these machine identifiers to network address of the location service on that machine. Given this mapping, an object could be found simply by sending a message to the local location object on that machine asking for the location (i.e., the network address) of that object.

However, objects do move and machines do get taken out of networks. it is, therefore, necessary to add further mechanisms to find those objects that are no longer located on their birth machines.

The approach we have taken is intrinsically object oriented, in that at each level of attempting to find an object some other object is asked. If the object can find the requested object, an answer is returned. If not, that object (not the original requesting object) can ask another object to help in the location task. This continues until either the object is found, or there is no one left to ask. In the latter case, the original requestor is told that the object cannot be found.

The first level at which any object is looked for is the **local location service** on the machine on which the requesting object resides. This approach is taken for two reasons. First, it is assumed that a large minority if not a majority of object interactions will take place on a single machine, even when that is not required. Secondly, going to the local location service allows the request for finding an object to always begin at a known place. Otherwise, the location services might have to get involved to find the first object to request a location, that clearly would not be possible.

If the object whose location is being requested is not local to the machine, the local location service on that machine will request location aid from the next entity in the location hierarchy. This is the **Object Region Broker**, or **ORB**. The ORB is responsible for mapping from all of the machines in an object region to the addresses of the local location services on that machine, and of keeping track of all of the objects that were born on machines within the object region that have moved to other machines. An object region is an administrative entity, and is designed to contain all and only the machines that commonly communicate with each other. It is expected that object regions will vary in size from a minimum of about 10 machines to a maximum of 1,000.

When an ORB is asked for the location of an object, it first looks to see if it is responsible for the machine on which the object was created. If it is, it looks in its table of moved objects to see if the object has moved out of the region or to a new machine within the region. If the object appears in this table, the table will also contain the identifier of the machine to which the object was moved. The ORB will then ask that machine to deal with the location request.

If no entry appears in the ORBs table but the ORB does own the machine on which the object was born, the ORB will send the location request to the local location service on the birth machine. It is expected that this will be the second most common case of location (after the case of location resolution by the local location service on a single machine) since it is assumed that the majority of the objects created will not be moved from their birth machine.

If the object was not created on a machine that is owned by the ORB, the ORB will send a message to the ORB that owns the birth machine of the object being sought. To do this requires that the ORB find the network address of the ORB that currently owns that birth machine. The ORB gets this information by making a request of the **Locater of Object Regions**, or **LOR**.

The LOR is a server that grants access to the information held in a database that maps the identifiers of all machines in the larger network universe to the network address of the ORB that governs that machine. This universe is expected to be up to 100,000 machines, and thus would include all of the machines in an enterprise, perhaps connected via a wide area network. Given the number of entries in this database, the information contained in each entry needs to be kept to a minimum. The current scheme calls for an entry of about 24 bytes per entry; thus the global database would be 2.4 megabytes if the number of machines never exceeded the number expected by the design requirements, and would only be 24 megabytes if the number exceeded those requirements by an order of magnitude.

In fact, the database used by the LOR will grow to somewhat more than the above sizes would indicate since other information must be kept in the LOR. As well as keeping information allowing a mapping of all machines to their current ORB, the LOR database will include a set of entries that serve as machine tombstones. As part of the administrative task of removing a machine from the system, a different machine can be assigned the role of being the "birth" machine for all of the objects that were actually owned by the machine being retired. This is required since the location scheme keys off of the birth machine of the object to be located—if that birth machine is removed from the location databases, there is no way to find the objects that were born on that machine. By allowing another machine to act as the surrogate birth machine, the location services can continue to function correctly even when objects survive longer than their birth machine.

It should be noted that, while a particular ORB and the global LOR are conceptually single entities, the design allows and the system implements all of them as replicated objects to increase availability. The objects work on distributed and replicated databases. Given the slowly changing nature of the databases, they are currently not guaranteed to be strongly consistent. However, as traces are left at various points during moves to allow the tracking of an object even when the ORB and LOR databases are out of date, the weak consistency of those databases should not result in the inability to find an object that has been recently moved.

This location scheme requires that the act of moving an object include updating of the location information about the object. When an object is moved, the ORB that governs the birth machine of the object is always informed. If the object is moved from its birth machine, the ORB simply adds an entry into its relocation tables, noting the new location of the object. If the object is moved from a machine that is not its birth machine to another machine within the object region, the existing entry in the ORB database (created when the object was first moved) is simply updated. When an object moves outside of its birth object region, the machine id of the new location is entered into the birth ORB database, along with a sequence count. Moves of an object once that object has been moved out of its birth ORB immediately cause the ORB of the object region the object is being moved from to have a tombstone entered. That ORB then sends a message to the object's birth ORB to update with the new location, along with a sequence number indicating the move sequence. The tombstone is not removed until the birth ORB acknowledges the receipt of the move call and updates its own database with the new location of the object. The possible delays in updating the birth ORB, caused by network problems or machine crashes, are the base cause of the sequence number for all such moves, as the ORB may reject a move update if it has received a higher sequence move update for that object, indicating that the object has been moved again.

While all of the details make the location scheme appear rather complex, when looked at in a more general way it is in fact simple and extensible. At each level of the location scheme from the local case on up, the location service involved can either find the object whose location is being requested or can hand the location request off to some other object. If neither of these is possible (i.e., the LOR is unable to find a birth machine or birth machine surrogate) the location services can answer that the object cannot be found.

The extensibility of the system is implicit in the strategy of only answering that the object cannot be found if there is no other object to whom the request can be directed. Thus if we wished to scale the system beyond the enterprise level, all that would be required would be to add to the LOR an identifier of another object (perhaps a Locater of Object Region Locaters) that could handle the next level of request. Adding this next layer would be transparent to the other layers of the location services.

This ability to extend also points out a basic philosophy concerning the cost of the location requests. It is assumed that objects will most often communicate with other objects that are close by. Thus the cost of locating an object grows in direct proportion to the distance away of the object—location of an object on the same machine is cheap; within the same object region somewhat more expensive, and outside the object region more expensive still. Note that this philosophy is not required—we could cut the expense of finding objects that are far away at the price of increasing the expense of objects that are close. At this point, we have no way of proving that our assumption of communication frequency being inversely proportional to object distance. While we feel that the assumption is sensible, if we find through experience that it is false we can and will change the location services accordingly.

Sending a message

The actual sending of a message from one object to another is, in its simplest form, just the issuing of a Remote Procedure Call (RPC). In fact, most of the hard work that is required to send the message, including type checking, parameter marshalling, and data format conversion of individual parameters, is left up to the underlying RPC system, that in our implementation is the Network Computing System. Hence we will not discuss those mechanisms here.

What is of interest is the way we have combined message passing and location. Most RPC systems require that the issuing of a RPC call be separate from the locating of the destination of the call; further, few RPC mechanisms have anything approaching the notion of an object as the destination of a call. The system we have built includes the locating of an object in the making of the call, hiding the fact that the object is remote from the programmer.

Thus, from the programmer's point of view, the sending of a message to a remote object looks like a regular function call. Within the RPC stub that is generated by the RPC compiler and included into the program are the calls to the location service that find the object, the calls to marshal the parameters to the call, the calls to the RPC mechanism that make the call, and the calls that unmarshall any return values to the call. Thus the programmer does not need to know any of the particulars of the system to make the call, and the actual mechanisms that are used are hidden from both the programmer and the code written by the programmer.

Of course, the system does not precede every message send with a call to the location services. Internally the location of an object is stored once it has been found, and it is this location that is first tried when an object sends a message. If the object tends to be long–lived (i.e., tends to be active for long periods of time) the location that is cached can be the actual location of the object. The more usual case, however, is that the location of an object that is guaranteed to be active on the machine on which the target object lives is stored. This will be discussed more fully in the section on object activation and deactivation.

This not only simplifies the programming process, but allows the system to be changed in the future without requiring change in any of the code that uses the system. The worst possibility facing the developer is the need to recompile an application that uses the system; as shared libraries become more common even that requirement can be avoided, allowing the system to change without requiring change or even recompilation of applications that make use of the system.

Activating and Deactivating an Object

Conceptually, the message delivery system is such that objects are always available for the processing of messages. However, to actually keep all objects active at all times would be prohibitively expensive in terms of system resources. Remember that we are assuming a system of 10,000,000 objects on 100,000 machines. Keeping an average of 100 objects active at all times on all machines and assuming that the usual object will be contained in its own process would swamp most existing machines and Unix implementation. Even if keeping all objects active at all times were feasible, it would not be a wise use of resources, as it is expected that most objects, like most data files, will be accessed intensively for relatively short periods of time, and then not accessed for other periods of time. Thus efficient use of resources requires that we be able to activate an object when necessary, and deactivate that object when it seems likely that it will not be receiving any more messages for some time.

Rather than require that objects be told explicitly to become active, we have taken the approach of activating an object when a message is sent to the object and that object is not then active. While this appears to be an obvious strategy, it does require that messages get sent to some third party rather than directly to the object whose services are being requested.

Fortunately, the location services described earlier provide an excellent object to act as the third party. Location, remember, is the process of translating from an object id to a network address. The usual network address that will be returned by the location services will be the address on which the local location service is listening for messages.

When a message arrives for an object that lives on the machine on which the local location service is running, the local location service object consults a table of active objects to see if the requested object is currently active. If it is, the message is just forwarded to that object. Otherwise, the local location service will fork a process and exec a system supplied proto–object.

This proto–object will then call back to the local location service, telling it the network address on which it is listening. The local location service will then send the proto–object a list of the object files that constitute the code that manipulates the object, and a string that we generally expect to be the name of the data file that has been registered as the repository for the object's persistent state. How the list of files that constitute the code that manipulates the object is determined is discussed in the next section. The proto–object will then load the indicated code into the process space it occupies, and call a required initialization routine, supplying it the string that had been passed by the location services. Note that the persistent data need not all be stored in a single file—that is up to the routine that is part of the object that will restore the persistent state. All that is required is that the routine that restores the object's state from persistent storage be able to do so from the information supplied by the string handed in during the activation process.

When this is finished, the object can now said to be active. It then sends a message to the local location service, telling it that messages to the object may be redirected to it.

Deactivation can occur either by a direct request from some remote object or by a request from the local location services. For example, someone using an object could send a deactivate request when finished with the object. The local location service also keeps track of how often messages arrive through it for the object, and can request that an object deactivate itself if no messages have come through for a period of time. The length of time is determined by a configuration parameter that can be changed at runtime.

Deactivation requests are generally issued as optional on the part of the object being deactivated. On receipt of such a request, the object may either honor the request, saving any changed persistent state to long term storage and freeing up its process space, or deny the request. The denial may be because the object is in the middle of doing something, or because the object feels that it is likely to receive additional requests for service in the near future. Again, it is part of the object oriented flavor of the system that deactivation is internal to the object. All the system requires is that the object be able to be deactivated; it does not dictate how that deactivation occurs or when it occurs.

However, the system does require that an object be able to honor a forced deactivation request. Such a request is necessary to guard against rogue or simply rude objects, who inappropriately hog system resources or who have lost control over their own actions. The forced deactivation protocol will have the local location service first tell the object that it is about to be deactivated. After a short period of time, that will allow the object to save itself to persistent store, the process that contains the object will be killed by the local location service, that is able to do that because of its status as the parent of the process. While this approach opens the door to an impatient local location service causing object corruption (by shutting down an object while that object is in the middle of saving its persistent state), it is necessary in a world where object creators make mistakes.

Since this scheme allows objects to be deactivated and activated without the objects that might be trying to contact those objects being informed, the usual model of contacting an object will be via the local location services on the machine on which the object resides. While it is possible for an object to advise other objects sending it messages to send those messages directly to the object, any object attempting such direct communication would be subject to a timeout wait if the object being sent the message is deactivated. Such a wait is not, it should be pointed out, catastrophic, as the sending object can simply resend the message in such a way that the location services are again invoked, that will send the message to the local location service and reactivate the object.

Binding Code and Data

Objects have been traditionally defined as data and the code that manipulates that data. While this is conceptually true, there are very few object implementations that actually copy the code that manipulates an object for each object.

Instead, objects are grouped together based on the code that manipulates the data within those objects, and that code is shared.

In a distributed system, we want to use the same sort of technique, but new complexities are introduced. We need to have the code that manipulates the data shared by all of the objects, but there may actually be different versions of that code on different machines, depending on, for example, the machine architecture.

As with most object models, we use the notion of the **class** of an object to bind data and code. The class of an object determines the operations that can be performed by the object. The term **operation** is being used in a technical sense here; it means (roughly) the identification of a call that can be made on an object along with the call signature (i.e., the set of parameters). In fact, there is a temptation to say that the real semantics behind the abstract notion of a class is that a class uniquely defines the set of operations that can be performed on an object and the semantics of those operations.

Such a definition, while attractive, will not suffice in a distributed environment of machines that can differ in instruction set, object code format, and data representation. Since objects can potentially be moved from from one machine to another, there needs to be a finer grained way of binding data to code than the simple identification of class as the set of operations and their semantics.

To see this, suppose that we wish to move an object, call it O, from machine A to machine B. We have no idea if A and B are the same sort of machine. To move the object, we must move the data that comprises the persistent state of the object and bind that data with the appropriate code to allow manipulation of that data.

Given the abstract notion of class that we are discussing to bind data and code, we are presented with only a limited number of alternatives. One alternative is to move the data from machine A to machine B, but leave the class identification to code on machine A. Doing this would require that any time we actually perform an operation on object O the code would run on machine A, since we have no guarantee that the code will run on machine B. It is not clear that this counts as a move, since machine A would be the "location" of the object when it was actually being manipulated. This would more properly be labelled a move of the persistent state of the object rather than a real move of the object.

We could also try moving the code to the target machine along with the object's persistent data. However, this assumes that the same code could run on all machines, that cannot be assumed in the environment we are talking about. Even if it were possible to make this assumption, such an approach would lead to duplication of code, as every object that is moved would have to have it's own copy of the code that manipulates that object.

A better solution is to require that the code for the class of an object be installed on the machine the object is to be moved to. However, in so doing we find that the original notion of an object's class needs to be extended. Knowing that there exists code on a machine that can be used to manipulate an object on that machine only allows us to move the object if the persistent data for the object can be read by that code. This may mean that the two machines have the same data format (or, more precisely, that the two instances of the code can interpret the same set of bytes the same way) or that part of the code on each machine can turn the data into some format that can be interpreted by code on the other machine.

While the two situations are logically equivalent, they have decidedly different consequences in the real world. If the data can be interpreted by the code on the two machines in exactly the same way, the move of the data can be done without the intervention of the object by simply copying the bits from the source machine to the target machine. If, on the other hand, the data can simply be massaged by code on each of the machines to allow interpretation, the code for the object needs to be actively involved in the copy or move from one machine to another.

The solution we took, therefore, was to distinguish between two sorts of class. The first notion can be defined as follows:

A class, C1, is the same as a class, C2, iff
    1) every operation in C1 is an operation in C2, and vice versa;

2) the semantics of the operations in C1 is identical to the semantics of the corresponding operation in C2; and
3) The data format of an object's persistent state for C1 is bit–for–bit identical to the data format of an object's persistent state for C2.

If two classes meet the above definition, we say that those classes are instances of the same implementation class, and copies and moves of an object from machines that have installed on them the same implementation class can be done by doing a byte copy of the persistent state of the object. The second notion of a class that we have included is defined as follows:

A class, C1, is the same as a class, C2, iff
1) every operation in C1 is an operation in C2, and vice versa;
2) the semantics of the operations in C1 is identical to the semantics of the corresponding operation in C2; and
3) The data format of an object's persistent state for C1 can be translated into a format understood by C2 using well known operations of C1.

If two classes meet this second definition, we say that those classes are instances of the same abstract class, and copies and moves of an object from machines that have installed on them the same abstract class (but not the same implementation class) can be done by doing a copy that involves the objects themselves massaging the object's persistent data.

Clearly the notion of an implementation class is more specific than that of an abstract class. We therefore store only the implementation class identifier with an object, and access a system–wide database that allows mapping of an implementation class into an abstract class.

The notions of implementation class and abstract class do not, obviously, contain all of the relations concerning the operations and data formats that might hold between two objects. Additional relations might include the ability to convert from one class to another (even though the two classes might have different operations) without loss of data or, perhaps, the permanent conversion from one class to another without the ability to convert back. Such relations, however, are considered more specialized than those that we wished to introduce into our base object model. While they could be added in a more specialized model built on our system, they do not form part of the base model that defines when communication can take place.

Extending the object model

In the system we have described thus far, we can separate the requirements made on objects into two basic categories. In the first category are the requirements made for the purposes of identifying and sending messages to objects. The second set of requirements center on the ability to activate and object, deactivate an object, and bind the persistent state of an object with the code that manipulates that object.

We consider the first set of requirements to be absolutely basic, as they define the bottom most communication mechanism that is used in object interactions. The second set of requirements, however, are not so basic. Two object systems that agree on the way in which objects are identified and how messages are sent might disagree on how objects are activated or deactivated or how the code that manipulates an object's state is connected to that state could still interact. The object infrastructure we have constructed allows for the addition of alternate object models that differ in the second dimension.

To see how we have accomplished this requires that we look inside the entity that we have referred to as the local location service. This entity is actually made up of an object manager (OM), that encapsulates the full object model described thus far, and a manager of object managers (MOM), that is the actual recipient of messages directed to the local location services.

The MOM receives messages directed to objects, and asks the OMs that it knows about if the object the message is directed to is owned by that object manager. If it is, the object manager is handed the message, to deal with as it sees fit. The object managers that the MOM knows about are determined by the contents of a configuration file, that tells the MOM where the code for the various object managers is located, and the name of the initialization routine for each of the object managers. When the MOM is started, it reads through the configuration file, dynamically loading the code for each object manager and calling on that manager to initialize itself.

This allows the addition of new object managers, encapsulating other object models, to the basic system. These object managers must export a small set of operations that allow them to work with the MOM, but beyond that are unconstrained in the way they model objects. These managers can be added at any time simply by changing the configuration file of the MOM and restarting the local location services. Thus the system can be extended to new object models without changing the underlying system or having access to the source code that was used to create the system.

Current status and future direction

We have implemented a system that embodies the object model described above, and it is currently undergoing internal alpha test. The system is being used as the base for a distributed object management facility for the next generation NewWave product on Unix.

Current research continues at both the base object level, and at enriched object models that can be built on top of this base level. A central concern at the base level centers on security. Current client/server based security mechanisms do not scale well to the sort of peer to peer communication scheme embodied in our distributed object model. Nor is it clear how a security system based on objects should interact with the underlying security provided by the file–based operating system.

Research on enriched object models centers around the extensions needed to the base object system outlined in this paper to meet the need of various users and application tasks. For example, it is unclear that an object model that is suitable to office automation is also suitable for a software engineering environment. Research into how the base model can be extended into these domains, and what interoperation between the extended models can be retained, is ongoing.

A third area of research concerns the relationship between this object model, which is centered on object oriented systems, and that needed for object oriented programming.

Bibliography

[1] Zahn, et al., Network Computing System Reference Manual, Prentice–Hall 1990 [ISBN 0–13–617085–4].
[2] Kong, et al., Network Computing System Reference Manual, Prentice–Hall 1990 [ISBN 0–13–617085–4]

Note

Unix is a trademark of AT&T.

Author Biography

Jim Waldo has been an engineer with Apollo Computer (now a division of Hewlett– Packard) for the past five years, working in the areas of object oriented programming, user environments, and distributed systems. An early user of the C++ programming language, he has published and talked extensively on the uses of object oriented programming languages and techniques in production settings. He is currently a Consulting Engineer in the Cooperative Object Computing Operation of Hewlett–Packard, where he is the lead architect of the system described in this paper.

The paper is intended for engineers, technical managers, and others interested in the future merging of distributed systems and object oriented technology.

Perspectives on NFS File Server Performance Characterization

Bruce E. Keith
Digital Equipment Corporation
129 Parker Street PK03-1/D18
Maynard, MA 01754
(508) 493-8889
keith@oldtmr.enet.dec.com

Perspectives on NFS File Server Performance Characterization

Bruce E. Keith

Digital Equipment Corporation
Systems Engineering Characterization Group, Unix-Based Systems and Software
129 Parker Street PKO3-1/D18
Maynard, Massachusetts 01754
keith@oldtmr.enet.dec.com

ABSTRACT

Two major approaches to Network File System (NFS[1]) file server performance characterization exist today. One approach, denoted the "synthetic-workload, single-client" (SWS) approach, uses an NFS workload abstraction in terms of an NFS operation request mix and an NFS operation request rate as input to a load generator utility running on a single, or small number of NFS clients [LEGATO89], [SHEIN89]. Another approach, used within Digital Equipment Corporation and denoted the "actual-workload, multiple-client" (AWM) approach, is to execute an actual workload on multiple NFS clients. In both approaches, various performance parameters are monitored while an NFS load is applied to the server.

This paper discusses the results of an initial evaluation of the SWS approach's ability to generate server and network loads and associated client response that are equivalent to those generated by the AWM approach. The paper further discusses the fundamental reason for investigating file server performance: helping a computing facility answer the question "How will our application (workload) perform using this file server?"

1 INTRODUCTION

Network File System (NFS[1]) file servers are key components in today's distributed computing environments. During a computing facility's NFS file server selection process, the individuals making the selection decision need to know how a given NFS file server configuration performs so that comparison can be made among servers from different vendors. Ultimately, the individuals need to know how well the computing facility's particular application, or workload, will perform using a given server configuration.

There have been two major approaches to NFS file server performance characterization to date. One approach, denoted the "synthetic-workload, single-client" (SWS) approach, requires a computing facility to develop an ad hoc benchmark that is considered representative of the computing facility's workload (application). The benchmark is then executed on a single (or small number of) client(s) of the NFS file server under test, which in turn impose(s) a load on the file server under test. Thus, the single client emits NFS requests as if it were a larger number of clients.

[1]NFS is a trademark of Sun Microsystems, Inc.

Another approach, denoted the "actual-workload, multiple-client" (AWM) approach, involves the simultaneous execution of an actual workload on multiple clients of the NFS file server under test, which in turn impose a load on the file server supporting the clients.

In both approaches, client and server performance are measured while the load is applied to the server, with performance typically expressed in terms of the clients' resultant response time and throughput and the server's corresponding resource utilization.

Refinements to the SWS approach have been made recently by [LEGATO89] and [SHEIN89]. These refinements are:

1. Abstraction of a file server's workload in terms of its NFS operation mix and NFS operation request rate.
2. Development of utilities that can generate an NFS load based on input expressed in terms of this workload abstraction.

This paper discusses the results of an initial evaluation of the refined SWS approach to NFS file server performance characterization. Specifically, the paper discusses the ability of the refined SWS approach, as instantiated by the *nhfsstone*[2] NFS load generating program [LEGATO89], to generate server loads and client response that are equivalent to those produced by the AWM approach used within Digital Equipment Corporation over the last 3 years.

1.1 NFS Functionality

NFS provides the ability to share file systems transparently in a heterogeneous environment of processors, operating systems, and networks. Sharing is accomplished through a cooperative mechanism in which a computer system, denoted a "server," exports (or offers) some or all of its file systems to other computer systems on a network. A "client" is any other computer system on the network that references any of the file systems exported by the server. A given computer system may act as either a server, a client, or both.

Once a client has remotely mounted a file system exported by the server, the client can then access the server's file system as if it were locally available on the client. This eliminates the need to copy files from one system to another before a client can reference a file. Consequently, only one copy of a file need be maintained on the server rather than several copies of the file on several different computer systems.

Internally, an NFS client makes requests of the server for files within an exported file system through the NFS protocol. The NFS protocol defines specific types of operations that a client system may request of a server [SANDBERG85].

1.2 SWS Approach

The types of operations defined by the NFS protocol provide the key to one of the significant refinements made by [LEGATO89] and [SHEIN89] to the ad hoc benchmark, SWS approach.

Both [LEGATO89] and [SHEIN89] abstract an NFS file server's workload in terms of a mix of the types of NFS operations expressed as a percentage of the total number of requests made by all of the clients supported by the server. The abstraction is further extended by both authors to include the rate at which the NFS clients make requests of the server. Thus, [LEGATO89] and [SHEIN89] suggest that an NFS server's actual workload can be abstracted in terms of an NFS operation mix and an NFS operation request rate. Both NFS operation mix and NFS operation request rate are quantities that can be readily determined through the *nfsstat* NFS statistics reporting utility.

[2] *nhfsstone* source code is a copyrighted product of Legato Systems, Inc.

The second significant refinement from [LEGATO89] and [SHEIN89] is the development of utilities to generate an NFS load based on the NFS workload abstraction of an NFS operation mix and an NFS operation request rate. The utilities are executed on an NFS client to generate a load on an NFS server.

The *nhfsstone* utility is a suitable choice for an NFS load generating tool. Thus, the *nhfsstone* NFS load generating utility developed by Legato Systems, Inc. was used as the SWS approach to NFS file server performance characterization within this evaluation.

The *nhfsstone* utility requires the following input parameters:

- An NFS operation mix file containing the results of an execution of the *nfsstat* NFS statistics reporting utility.
- An NFS operation request rate.
- A list of remote file systems to serve as targets for the generated NFS requests.
- Either the length of time for which the utility is to generate the synthetic load or the total number of NFS requests that are to be generated by the utility.

Upon completion of execution, the *nhfsstone* utility reports the actual NFS operation mix generated, the elapsed execution time, the total number of NFS operation requests generated, the resultant NFS operation request rate, and the average service time for all of the generated NFS operation requests. Consult [LEGATO89] for a description of additional output generated by the utility.

Thus, the SWS approach characterizes NFS file server performance in terms of NFS request service times and NFS request throughput experienced on an NFS client.

1.3 AWM Approach

The AWM approach to NFS file server performance characterization used within Digital Equipment Corporation during the past 3 years applies a real user-level task workload to ULTRIX[3] NFS diskless client workstations, which in turn apply a load to the NFS server. As many as 50 ULTRIX NFS diskless client workstations have been configured on a private Ethernet and supported by a single ULTRIX NFS file server.

As the workload is applied to progressively larger numbers of client workstations, user-level task service time and throughput are measured on the client workstations. Server resource utilization is simultaneously measured in terms of CPU utilization and disk and network interface I/O operation rates. Further, network utilization is simultaneously measured on the Ethernet.

The workload applied to the client workstations emulates a software engineering environment in which a single software engineer is assigned to each NFS client workstation. Each client workstation executes a unique, repeatable sequence of user-level tasks (commands) selected from a common pool of the most frequently executed user-level tasks encountered in a software engineering environment. Thus, the workload is not a lock step workload in which each workstation executes the same task at the same time. Rather, the workload follows typical usage patterns in which individual users collectively execute a variety of user-level tasks at a given point in time.

Hence, the AWM approach characterizes NFS file server performance in terms of user-level task service times and throughput experienced on NFS clients. The AWM approach also characterizes NFS file server performance in terms of the associated utilization of server and network resources required to provide a particular degree of user-level task response.

[3]ULTRIX is a trademark of Digital Equipment Corporation.

2 EVALUATION METHODOLOGY

2.1 Focus

The focus of the initial evaluation of the SWS approach as implemented by the *nhfsstone* NFS load generator utility was to determine *nhfsstone's* ability to duplicate NFS file server utilization, network utilization, and client response levels previously measured by the AWM approach. Further, the evaluation was to determine if the SWS approach could be used as a substitute for the AWM approach as an NFS server sizing method. The evaluation was not intended to be a critique of the *nhfsstone* load generator utility itself.

The following primary questions were addressed by the evaluation:

1. Is the NFS operation mix of an actual workload invariant with respect to scaleability? Specifically, is the NFS operation mix generated by a single NFS client equivalent to the NFS operation mix generated by "N" clients? This is the fundamental assumption of the SWS approach that provides the basis for using a single client to simulate "N" actual clients.
2. Does the NFS operation request rate of an actual workload increase linearly with respect to scaleability? Specifically, is the NFS operation request rate of "N" clients equivalent to the average NFS operation request rate of one client multiplied by "N"? This point addresses the issue of how a server responds as its load (client NFS operation request rate) increases.
3. Is the workload offered by the *nhfsstone* load generator equivalent, in terms of server and network utilization, to the offered load generated by the AWM approach. Further, can the load offered by "N" actual clients be simulated by adjusting *nhfsstone's* NFS operation request rate while holding the operation mix constant?
4. Do NFS operation service time trends, as reported by *nhfsstone*, have a relationship to the user-level task service time trends measured by the AWM approach?
5. What issues, if any, exist concerning the usage of the *nhfsstone* load generator utility?

2.2 Strategy

The evaluation strategy was to configure an NFS file server that had been previously characterized by the AWM approach. The client NFS operation mix and operation request rates measured during the previous characterization of the server were used as input to *nhfsstone*. Server and network resource utilization were measured while *nhfsstone* applied the load to the server for a 1-hour interval. The client NFS operation mix and operation service times reported by *nhfsstone* were recorded and verified with parallel measurements made by the *nfsstat* NFS statistics reporting utility on the *nhfsstone* platform.

2.3 Metrics and Tools

Wherever possible, standard ULTRIX and UNIX[4] system performance monitoring utilities were used to monitor server and *nhfsstone* platform performance. Table 1 summarizes the performance metrics and associated performance monitoring utilities used during the evaluation.

[4]UNIX is a registered trademark of American Telephone and Telegraph.

Performance Monitoring Tool	Metric
vmstat	• Average server CPU idle time • Simultaneous average and maximum server disk I/O operation rates
netstat	• Average and maximum server network interface packet rates
nfsstat	• NFS operation request mix and rate
nhfsstone	• Average NFS request service time
LAN Traffic Monitor	• Ethernet network utilization

Table 1 Metrics and Tools

2.4 Testbed

Table 2 summarizes the hardware and software used during the evaluation. A private Ethernet network was used to interconnect the NFS server and clients. Note that a local disk was used for the ULTRIX operating system on the *nhfsstone* platform so as not to place the load of an additional diskless client on the server during testing.

	AWM/SWS Approach File Server	**AWM Approach Diskless Clients**	**SWS Approach *nhfsstone* Platform**
System	DECsystem[6] 3100	DECstation[6] 3100	DECstation 3100
Memory	24 Megabytes	16 Megabytes	16 Megabytes
Disks	3 RZ55[7]	N/A	1 RZ55
Software	ULTRIX V3.1 (RISC)	ULTRIX Worksystem Software V2.1 (RISC)	ULTRIX Worksystem Software V2.1 (RISC)

Table 2 Testbed

[6]DECsystem and DECstation are trademarks of Digital Equipment Corporation.
[7]An RZ55 disk is a 332 Megabyte SCSI disk drive.

3 RESULTS

3.1 Summary of Findings

The following list summarizes the findings of the evaluation, with further detail in the following sections.

1. NFS operation mix was found to be invariant with respect to scaleability. Thus, the assumption that a single client can simulate 'N' actual clients in terms of a constant NFS operation mix was validated. Additionally, *nhfsstone* accurately reproduced the requested operation mix.
2. NFS client operation request rate was found to be non-linear with respect to scaleability, due to server response degradation. This implies that a rate other than "N" times the average NFS operation rate of one client must be specified as input to *nhfsstone* for each level of clients (load) in order to simulate the actual load. This prohibits *nhfsstone* from being a total replacement for the AWM approach.
3. Average server and network resource utilization levels were equivalent between the AWM approach and the SWS approach, however maximum resource utilization levels were significantly lower with the SWS approach than with the AWM approach.
4. A relationship between the NFS operation service times of the SWS approach and the user-level task service times of the AWM approach was found. Thus, *nhfsstone* results can serve as a rough, ballpark indicator of client application performance.
5. No major configuration or capacity issues arose concerning the *nhfsstone* platform throughout the evaluation.

3.2 Invariance of NFS Operation Mix

NFS operation mix was found to be invariant with respect to scaleability by inspection of measured data obtained through the AWM approach. The NFS operation mix varied no more than two percent across the range of clients. Thus, the fundamental assumption of the SWS approach that a single synthetic client can simulate "N" actual clients in terms of NFS operation mix for a given actual workload was validated.

Invariance of NFS operation mix is an important property since a server's response will vary as NFS operation mix is varied. Since NFS file server performance characterization is primarily interested in the server's response to increased load for a given NFS operation mix (workload), any change in server response due to variance of NFS operation mix would skew the desired characterization results.

A server's response changes as NFS operation mix is varied since not all NFS operation types are equally expensive in terms of the work that the server must accomplish to process the NFS request [BRIGGS88]. The underlying issue is that, since an NFS server is stateless, the server must relegate any modified data to non-volatile storage before notifying the client that the requested operation was completed [SANDBERG85]. For example, [BRIGGS88] reports that NFS operations involving file system changes are much slower in terms of server response than operations that can be resolved from various caches on the server.

Figure 1 presents a bar graph of NFS operation mix for those NFS operation types utilized by the actual workload. The *fsstat*, *link*, *readdir*, and *rename* NFS operations each constituted less than 1 percent of the total NFS operation mix.

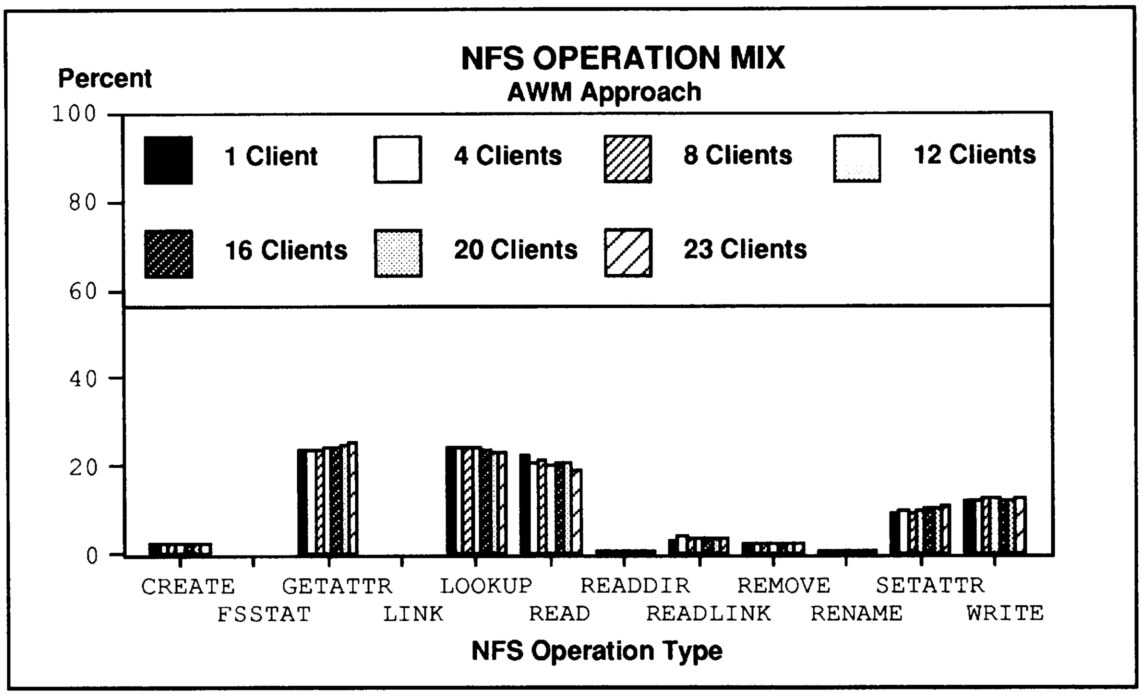


Figure 1 NFS Operation Mix

3.3 Non-linearity of NFS Operation Request Rate

NFS operation request rate was found to be non-linear with respect to the number of clients by inspection of measured data obtained through the AWM approach. The non-linearity is attributable to server response time degradation and a second-order effect of this degradation. In particular, the actual workload, reacting to slower server response, is unable to generate NFS requests as quickly. The net effect of both these issues is that NFS operation call rate increases due to increased numbers of clients, but individual client call rate decreases due to increased service time at the server.

Figure 2 illustrates a graph which plots an "ideal" NFS request rate and the actual NFS request rate versus the number of active NFS clients as measured through the AWM approach. The "ideal" NFS request rate is defined by the case where the NFS operation request rate of "N" clients is equivalent to the average NFS operation request rate of one client multiplied by "N". In this situation, an "ideal" or "responsive" server [KLEINROCK75] responds to increased client request rates with no performance degradation whatsoever.

The difference between the "ideal" and "actual" server curves is attributable to the aforementioned reasons. The difference implies that a rate other than "N" times the average NFS operation rate of one client must be specified as input to *nhfsstone* for each level of clients (load) in order to simulate the actual load. This means that degradation of NFS operation rate of the workload per unit of load (clients) due to server response degradation must be known empirically and supplied as input to *nhfsstone* through the requested NFS operation rate, in order for *nhfsstone* to duplicate the actual workload. This prohibits *nhfsstone* from being a total replacement for the AWM approach.

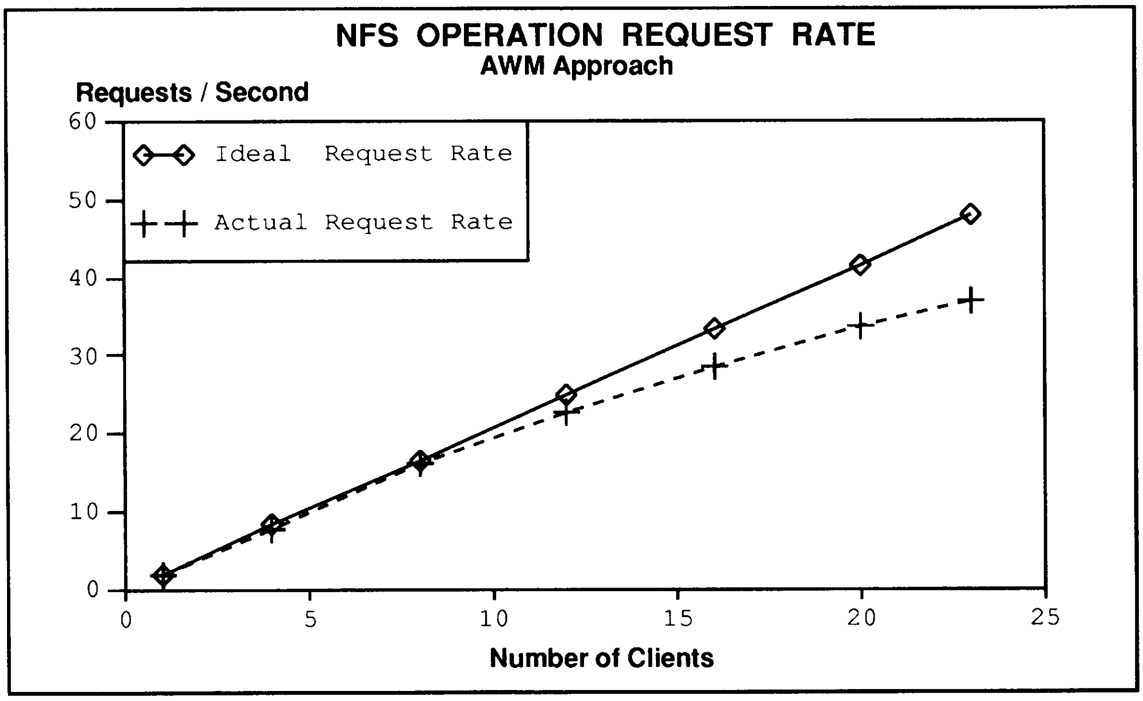


Figure 2 NFS Operation Request Rate

Figure 3 illustrates NFS request service time as measured by *nhfsstone*. The specified *nhfsstone* input parameters were set according to the results of the AWM approach, which implicitly contained a measure of server response degradation.

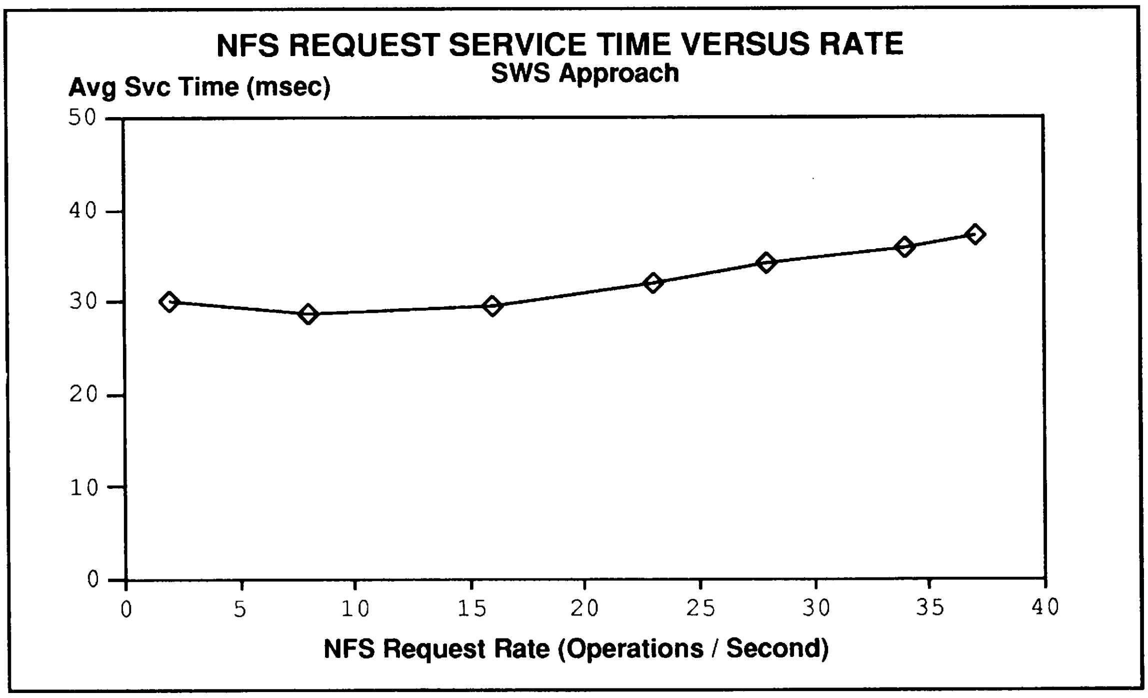


Figure 3 NFS Request Service Time

3.4 Equivalence of Server and Network Resource Utilization

Nhfsstone generated **average** server and network resource utilization levels that were comparable to those obtained through the AWM approach. The average load offered by "N" actual clients was successfully simulated by adjusting *nhfsstone's* NFS operation request rate while holding the NFS operation mix constant. The resultant average load and associated server and network utilization levels can be used to investigate server performance issues by holding NFS operation mix and request rate constant, varying the server configuration, and observing changes in NFS request response time.

However, *nhfsstone* did not produce **maximum** utilization levels that were comparable to those experienced with the AWM approach. The maximum utilization levels produced by *nhfsstone* were significantly lower than those experienced with the AWM approach. This can bias the average NFS request service times reported by *nhfsstone* in that the average service times reported can be lower (more optimistic) than what would be obtained with an actual workload. The reason is that maximum resource utilization levels increase NFS request service times.

The reason for the absence of maximum resource utilization levels in the load generated by *nhfsstone* lies in the manner through which *nhfsstone* allows the NFS operation request rate to be specified. The NFS operation rate is specified as a constant when, in reality, NFS operation rate varies as a workload is executed. Thus, the load and resource utilizations generated by *nhfsstone* are much smoother than what occurs in practice.

For example, at the 23-client level, the NFS clients in the AWM approach requested an average of 37 NFS operations per second over the entire test interval of one hour. However, during a 10-minute peak period of activity within the total test interval, the clients requested 58 NFS operations per second from the server.

Figures 4 through 7 present graphs of percentage server CPU idle, server disk I/O rate, server network interface I/O rate, and network utilization for average and maximum values achieved with both the AWM approach and the SWS approach.

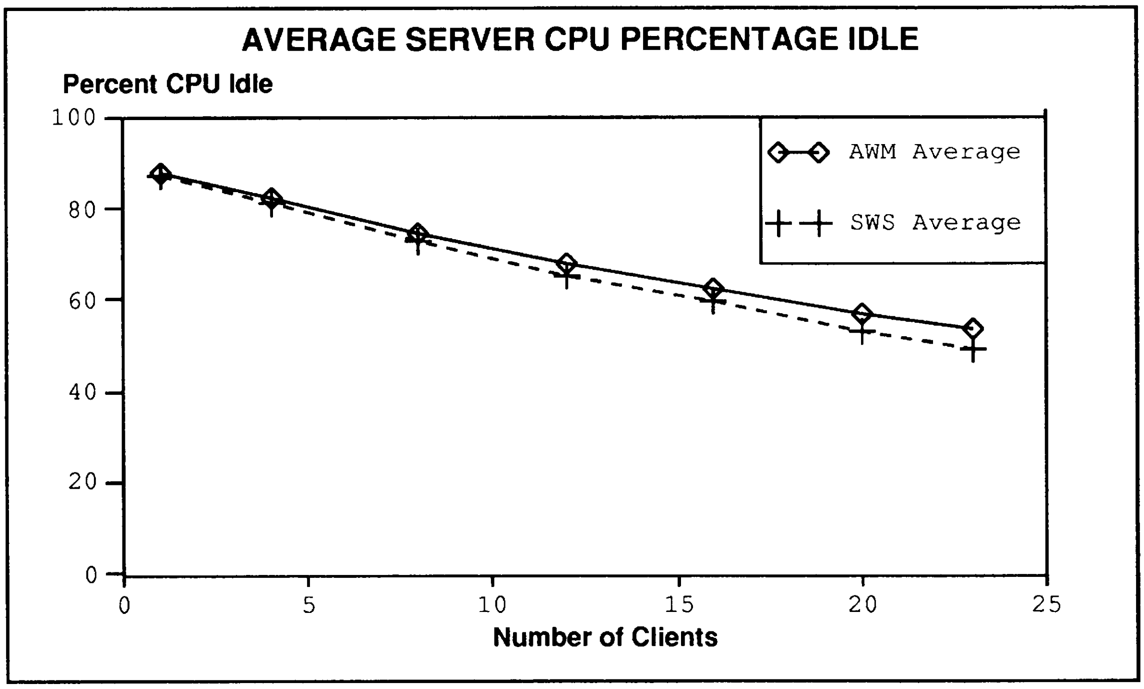


Figure 4 Average CPU Idle

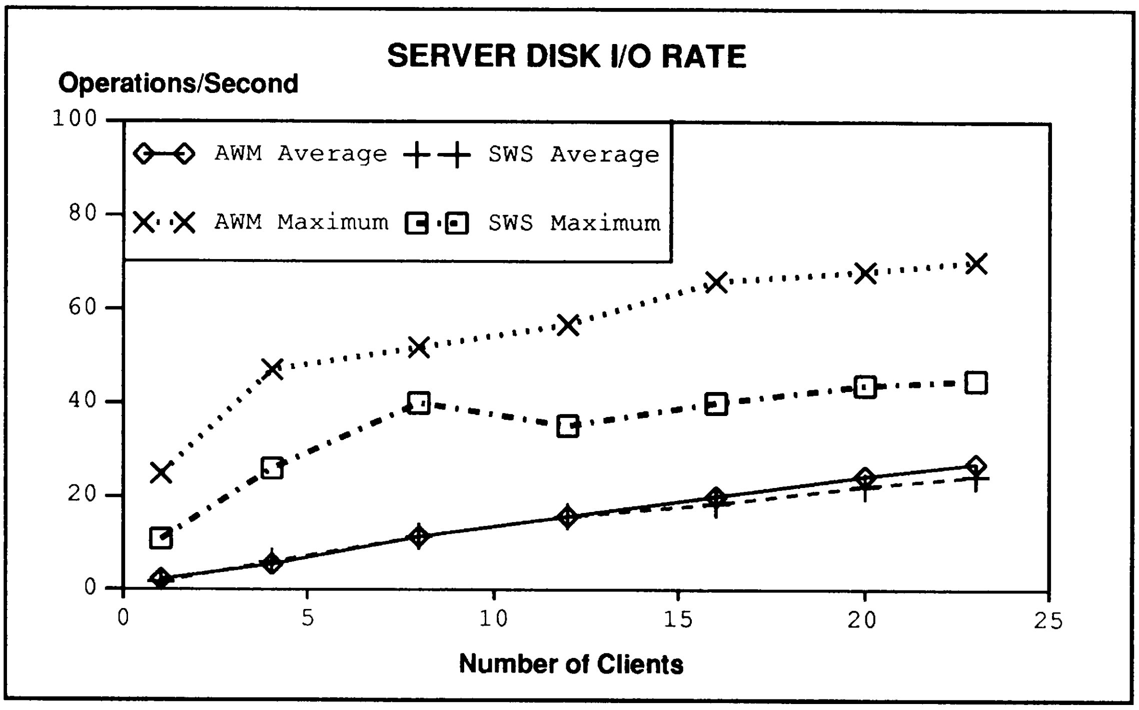


Figure 5 Server Disk I/O Rate

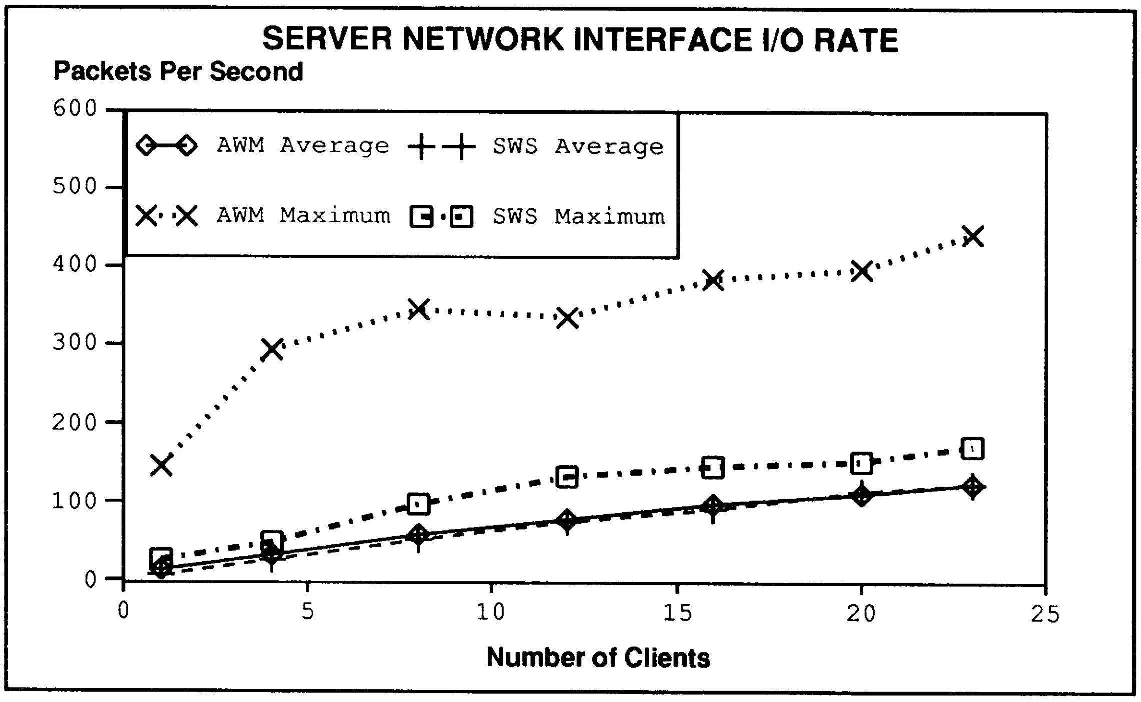


Figure 6 Server Network Interface I/O Rate

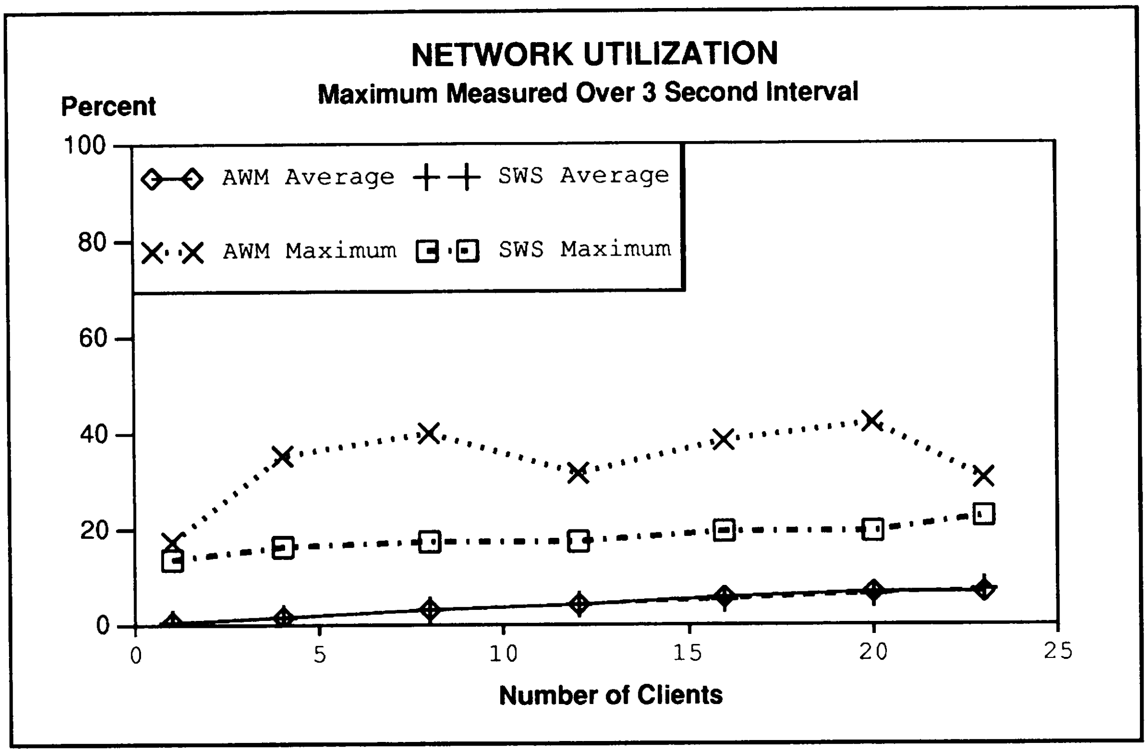


Figure 7 Network Utilization

3.5 Relationship of Client Response Trends

In order to serve as an indicator of client application performance, average NFS request service time reported by *nhfsstone* must relate to the average user-level task service time reported by the AWM approach. To investigate the presence of a relationship, the normalized average NFS request service time reported by *nhfsstone* was plotted with the normalized average user-level task service time measured by the AWM approach for the range of clients tested. Figure 8 illustrates the resultant graph.

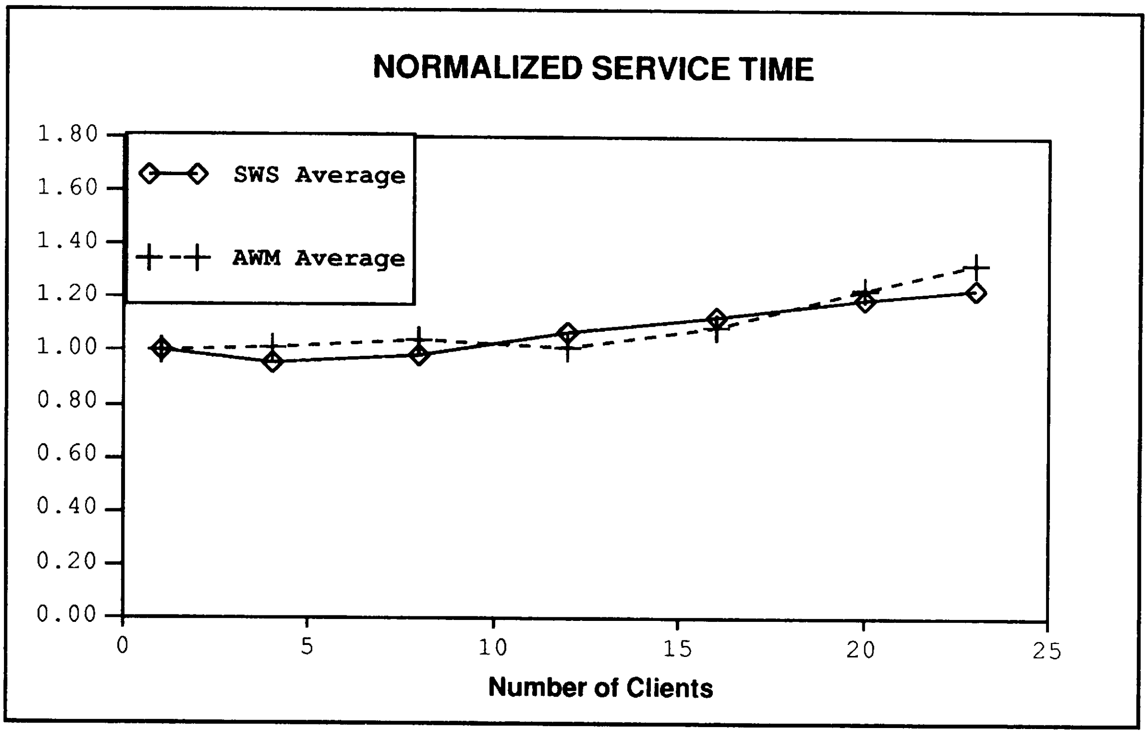


Figure 8 Normalized Service Time

The values of the points on each curve of the graph in Figure 8 were normalized by dividing each of the curves' respective data points by the value of the respective curve at the 1-client level. The desired effect was that the y-axis value of each curve at the 1-client level had a value of 1, to facilitate comparison. The measured average service time values of *nhfsstone's* NFS requests were on the order of 28 to 37 milliseconds while those of the AWM approach's user-level tasks were on the order of 8 to 12 seconds.

The graph suggests that a relationship exists between the average NFS request service time reported by *nhfsstone* and the average user-level task service time reported by the AWM approach. Thus, average NFS request service time as reported by *nhfsstone* appears to be a rough indicator of application performance, and in this case, a ballpark indicator of user-level task service time measured through the AWM approach.

It should be noted that NFS request service times as reported by *nhfsstone* form an upper bound on client NFS-related degradation since *nhfsstone* attempts to defeat data buffer, file attribute, and directory name lookup caches that exist on the client [SANDBERG85]. These caches help reduce the number of NFS requests that the client must issue for the client application to accomplish its work.

In the initial evaluation documented herein, the exact nature of the relationship between the average NFS request service time of *nhfsstone* and the average user-level task service time of the AWM approach could not be determined. One factor that influenced this outcome was that the average NFS request service time of *nhfsstone* and the average user-level task service time of the AWM approach are not independent variables, thus preventing the determination of a statistical correlation. Another factor was one of measurement precision in that NFS request response was measured on the order of milliseconds by *nhfsstone* while the AWM approach measured user-level task response on the order of hundredths of seconds. A test procedure that resolves both of these issues has been identified and will be included in future work.

The determination of the nature of the relationship serves as an ideal starting point for future work. Given that response trends are typically exponential, the relationship here could very well be exponential. Further, Figure 8 suggests that the relationship might be sinusoidal.

3.6 *nhfsstone* Platform Issues

No major issues were uncovered during the evaluation concerning the usage of *nhfsstone*. A 16-Megabyte DECstation 3100 was easily able to duplicate server and network utilization levels associated with 23 clients executing the actual workload and beyond. At no point during the evaluation was the platform unable to deliver the requested load using 12 *nhfsstone* subprocesses.

Server filesystem layouts and the number of server disks can be significant performance factors when configuring a server, however, this was not the case during this initial evaluation. For example, comparable results were obtained when *nhfsstone* referenced three file systems on the server as when four file systems were referenced on the server.

4 CONCLUSIONS

Nhfsstone produced average server and network utilization levels that were comparable to those experienced with the AWM approach. This proved the validity of the NFS workload abstraction in terms of an NFS operation mix and an NFS operation request rate. Further, *nhfsstone* accurately generated requested NFS operation mixes and NFS operation request rates during the evaluation.

The average NFS loads generated by *nhfsstone* can be used to investigate server performance. Changes in average NFS request service time can be observed while modifying the configuration of the server and holding NFS operation mix constant. Graphs similar to Figure 3 which plot NFS operation service time versus NFS operation request rate for a constant NFS operation mix can be used as a means of comparison among various servers and server configurations.

Graphs which plot both NFS operation service time and NFS operation request rate take into account the fact that the actual service time associated with a high server NFS processing rate may not be as acceptable to a computing facility as the lower service time encountered at a lower server NFS operation processing rate. Additionally, NFS operation service time trends can serve as a rough, ballpark indicator of client application performance. Thus, graphs that plot both NFS operation service time and NFS operation request rate will assist a computing facility in answering the question "How will our application perform using this file server?" However, the answer will be approximate rather than exact.

The maximum server resource utilization levels produced by *nhfsstone* were not comparable with those obtained with the AWM approach. This can cause the average NFS request service time reported by *nhfsstone* to be optimistic (low). Optimistic response indications further obscure the relationship between the service times reported by *nhfsstone* and the AWM approach. This optimism is offset by the pessimistic nature of *nhfsstone* in defeating various client caches which tends to increase service times. The degree to which the pessimism balances the optimism could serve as an area for future work.

Nhfsstone complements, but does not replace, the AWM approach to NFS file server performance characterization given the non-linearity of NFS operation request rate identified in Section 3.3. In order for *nhfsstone* to exactly duplicate an actual workload, the non-linear degradation of a workload's NFS operation request rate must be known in advance and implicitly supplied to *nhfsstone* through its requested NFS operation rate input parameter.

The AWM approach continues to have merit in that, in addition to providing user-level client response information, the approach also tests large scale software and hardware interoperability. Further, the AWM approach does not require prior knowledge of a workload's NFS operation request rate degradation to associate client NFS request rate levels with given numbers of clients.

4.1 Possible Future Work

Several areas of future work are possible. As previously mentioned, the relationship between average NFS request service time as reported by *nhfsstone* and user-level task service time as reported by the AWM approach should be further investigated. The ability of *nhfsstone* to replicate other actual workloads should also be investigated.

Measurements of NFS operation distributions (mixes) associated with several computing environments should be made and distributed within the industry so that these NFS operation distributions can be consistently used to characterize an NFS server's performance in different computing environments. A checklist of NFS setup parameters (e.g., NFS timeout interval, etc.) should be defined to ensure that a consistent environment is established when servers are characterized with the various NFS operation distributions.

The *nhfsstone* utility might be further enhanced to more accurately reproduce actual workloads in terms of maximum server resource utilization levels. This would improve the accuracy of the average NFS request service time reported by *nhfsstone* so that more accurate predictions of client application response could be made. Rather than taking a single NFS mix and operation request rate as input, an enhanced version of *nhfsstone* might accept several mixes and request rates as input so that it could dynamically change the generated load during execution. Alternatively, an enhanced version of *nhfsstone* could generate a distribution of NFS operation request rates within limits set by the user. Both of these enhancements would assist *nhfsstone* in expanding its role into an actual workload abstraction/capture/replay tool targeted towards more accurate prediction of client application response than its current role as a load generator. These enhancements would allow *nhfsstone* to grow into a tool that would allow a computing facility to more accurately answer the question "How will our application perform using this server?"

5 ACKNOWLEDGMENTS

I am indebted to many people for their various contributions to this evaluation: Charlie Briggs and Chet Juszczak for their encouragement, insights, and comments; Paul Kotschenreuther for his comments and role as chief sounding-board; Santa Wiryaman for his comments and automation of the evaluation process; Judy Piantedosi for her assistance in test environment setup and WYSIWYG document preparation wizardry; Muhammad Ashraf and Doug St. Clair for their statistical prowess; Lucy Patton and Angela Richard for their technical writing critiques; all of those in USS-SECG Lab Services for their fast and accurate response to test equipment and configuration requests; and last but not least, Lee Allison, Paul Kruger, and Linda Chasey Vincent of USS-SECG management for their strong and constant support of this evaluation.

6 REFERENCES

[BRIGGS88]	Briggs, Charles, "NFS Diskless Workstation Performance", Digital Equipment Corporation Technical Report, February 24, 1988.
[KLEINROCK75]	Kleinrock, Leonard, "Queueing Systems Volume 1: Theory", Wiley-Interscience, 1975.
[LEGATO89]	*nhfsstone* NFS load generating program, Legato Systems Inc., Palo Alto, CA 94306.
[SANDBERG85]	Sandberg, Russel, et al., "Design and Implementation of the Sun Network Filesystem", USENIX Summer Conference Proceedings, pp. 119-130, June 1985.
[SHEIN89]	Shein, Barry, et al., "NFSSTONE - A Network File Server Performance Benchmark", USENIX Summer '89 Conference Proceedings, pp. 269-274.

7 BIOGRAPHY

Bruce Keith is a Principal Software Engineer in the Low End Systems Systems Engineering Characterization Group at Digital Equipment Corporation. Since joining Digital in December 1986, he has been leading a project concerning the performance characterization of ULTRIX NFS file servers. Prior to joining Digital, Bruce developed systems software spanning UNIX, VMS[8], RSX-11M[8], and TOPS-10[8] operating environments for timesharing, academic, and technical OEM businesses. Bruce received his BS degree in Computer Science from Worcester Polytechnic Institute.

[8]RSX-11M, TOPS-10, and VMS are trademarks of Digital Equipment Corporation.

Enterprise Transaction Processing

Terence Dwyer
UNIX System Laboratories
190 River Road
Summit, NJ 07901
(201) 522-5039
attunix!tjd

ABSTRACT

Traditionally, Transaction Processing (TP) has been performed on centralized mainframe computers running proprietary operating systems, and proprietary TP system software. In the last several years, we have seen the emergence of TP software, including high performance Relational Database Management Systems (RDBMS), e.g. [INFORMIX] and [ORACLE], and TP Managers such as TUXEDO® System /T [AT&T], for computers running the UNIX® Operating System. However, the real promise of transaction processing on UNIX-based computers lies not in the ability to provide the "stand-alone" TP model common to proprietary systems, but in the ability to provide the hub of an integrated, distributed Enterprise Transaction Processing (ETP) System. This paper describes the hardware and software architecture of an ETP System. Such a system will enable proprietary TP users and vendors to capitalize on the trends toward decentralized computing, to migrate to UNIX-based TP systems, and to protect their investment in proprietary TP systems.

1. INTRODUCTION

Figure 1 shows a typical ETP system configuration. This configuration is composed of the following components:

- Tier-1: Personal Workstations (WS)
- Tier-2: UNIX TP Servers (UTPS)
- Tier-3: Proprietary TP Servers (PTPS)

The Tier-1 WS machines, running a variety of proprietary operating systems (e.g. MS-DOS™, OS/2™, MACOS™, etc.) as well as the UNIX Operating System, are connected to a network, perhaps a Local Area Network (LAN), as shown in Figure 1. These machines are used to provide user interface processing. They allow the possibility of the attachment of hundreds of users to each UNIX TP Server, and offer the possibility of a wide variety of new interfaces for TP applications, including Graphical User Interfaces (GUI), in addition to the traditional forms-oriented TP-input paradigm.

Tier-2 consists of a networked set of powerful mid-sized computers running the UNIX Operating System and TP system software, such as TP monitors and RDBMS systems. The network connecting Tier-2 machines could be a LAN (perhaps the same one to which the Tier-1 machines are connected, as shown in Figure 1), or a Wide Area Network (WAN). The Tier-2 machines provide distributed TP services in the UNIX environment, including access to a variety of TP applications using the high performance RDBMS systems now available on UNIX platforms. In addition they link the workstations to the proprietary machines.

Tier-3 machines are mainframe class computers running proprietary operating systems and proprietary TP monitors such as IBM's CICS [IBM-1]. Today, such machines do the bulk of TP processing for most corporations, and contain a large investment in programs and stored data. Access to these programs and data will be required as corporations move to TP Systems based on computers running the UNIX Operating System. Increasingly, Tier-3 machines will take on the role of proprietary TP servers. Figure 1 depicts a single Tier-3 machine with point-to-point connections from two Tier-2 machines. The exchange of data between Tier-2 and Tier-3 machines is likely to be carried over special networks supporting the proprietary protocols required to interface to Tier-3 machines.

Thus, an ETP system is composed of a collection of heterogeneous machines (and attendant operating systems), ranging from the personal computer to large proprietary mainframes. The Tier-2 (UNIX-based) machines play the central role in this system, providing local TP services to the workstation community, and connecting them to proprietary environments.

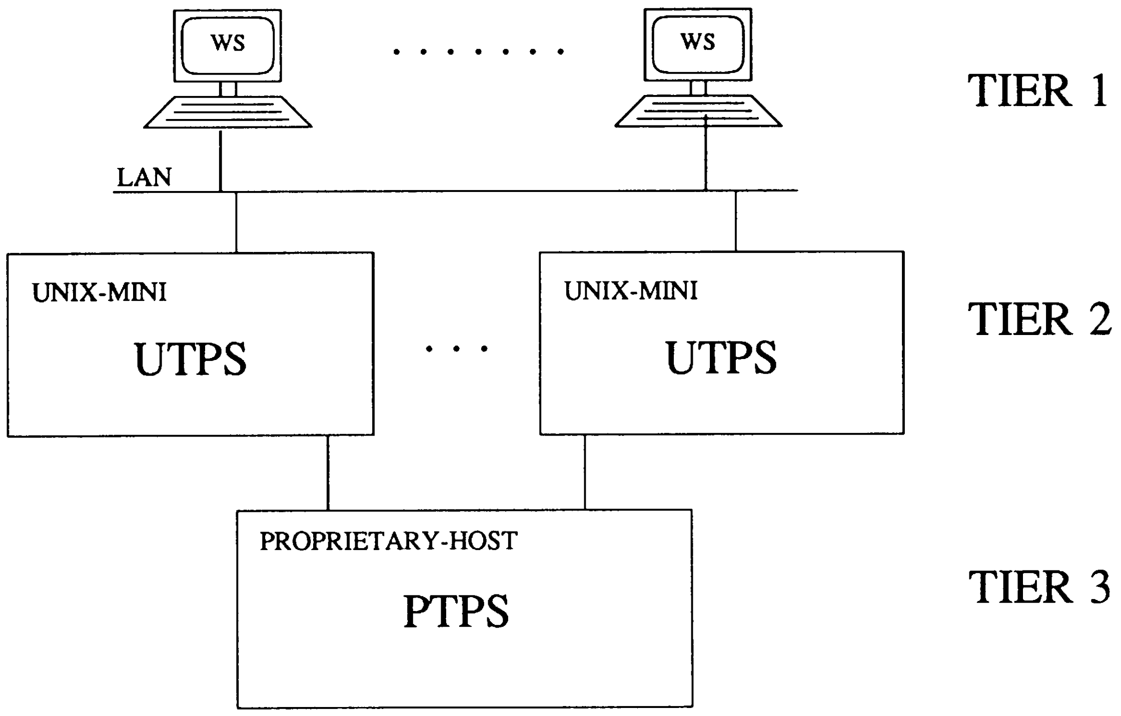


Figure 1. ETP Architecture

The TP platform software of the ETP system is the "glue" which binds together the hardware tiers into a unified TP System. As such, it provides communications, transaction, and administrative services to applications programmers and administrators, and may exist on all tiers. Key to the integration of an application across the tiers is the existence of a common TP Application Programming Interface (TP-API) for intermodule communication and transaction control. At the workstation level TP-API provides for the communication of input requests to the Tier-2 machines. At the Tier-2 level, TP-API provides for the reception of workstation input and the invocation of Tier-2 application services, or the forwarding of the requests on to Tier-3 machines. At the Tier-3 level, TP-API provides for the execution of TP service requests received from the Tier-2-machines. Goals of TP-API include consistency of syntax and semantics, location transparency of invoked modules, and transaction semantics on executed actions throughout the levels.

Subsequent sections of this paper expand on the architecture of ETP. Because of its central importance, this paper begins in Section 2 with a more complete description of Tier-2. Section 3 shows how Tier-1 is incorporated in ETP. Section 4 provides considerations for the inclusion of Tier-3 (i.e. proprietary TP) systems into ETP. Section 5 reports on the status of the construction of a commercial grade ETP system.

2. TIER 2: THE UNIX TP HUB

We start with Tier-2, the "middle tier" of the ETP System. [Landis] provides good insight why UNIX-based computers provide excellent functionality to play the middle role in ETP. This level consists of a networked set of powerful minicomputers running the UNIX Operating System. The use of a set of minicomputers offers several advantages including:

- the growing price advantage of machines smaller than mainframes
- the ability to mix heterogeneous machines, each suitable for particular tasks
- the ability to integrate several department size TP applications, each running on dedicated hardware, into a single application domain.

It should be noted that Tier-2 is more than a switcher of workstation requests to Tier-3 machines. Tier-2 machines themselves contain executable application code and shared databases. As TP applications are made available on Tier-2 machines (either new applications, or migration of Tier-3 applications to Tier-2 machines), many of the requests initiated from Tier-1 machines, or originating from within Tier-2 itself, may be completely satisfied at Tier-2. Tier-2 machines thus contain important, potentially "mission critical", resources for the corporation. As such, they are accorded the security and administration (e.g. backup) due traditional mainframe TP resources.

The TP environment for Tier-2 can be provided by an extension of TUXEDO® System/T. As described in [Andrade], TUXEDO System /T provides a powerful client/server model suitable for building high performance TP systems on Symmetric Multiprocessor (SMP) computers running the UNIX Operating System. Features of the System /T architecture include a high performance, shared-memory based name server, called the "Bulletin Board" (BB), and interprocess communication via System V messages. Requirements for the extension of such an architecture to a distributed Tier-2 architecture include:

- Communications Support for inter-machine client/server interactions
- Distributed Transaction Support (DTP)
- A TP oriented API.
- Centralized Administration

Figure 2 depicts a two node, Tier-2 system built upon this extended architecture.

2.1 Communications

Extension of System /T to the distributed case can be implemented by:

1. the distribution of the "Bulletin Board"
2. the extension of the messaging system

2.1.1 Name Server Distribution The performance constraints of TP systems require a fast method for determining client/server rendezvous. The shared-memory implementation of the BB in the SMP implementation of TUXEDO System /T fulfills this requirement. In the distributed case, it is desirable for each node to retain this fast access. One way to do this is to replicate the BB on all of the nodes. However, the BB contains two types of information: name-to-address mapping, and statistics. The former is used to provide location independence for client-server requests, and the latter is used for both administrative purposes and for load balancing. The name-address mapping information represents (relatively) stable information in a TP system, while the statistics are much more volatile.

The propagation of the BB's stable data can be accomplished through a special set of distributed administrative server processes called "Bulletin Board Liaison" (BBL) processes. Statistics, on the other hand, are kept locally at each site, and are made available administratively. These statistics are too volatile to be propagated throughout the system, and are not used as the basis of load balancing. Instead, a round robin method is used at each site to balance service requests originating from that site.

2.1.2 Inter-Machine Messaging As described in [Andrade], System V messages have very good properties for TP systems. In particular, they provide for a priority-based, reliable datagram service upon which efficient client/server interactions may be built. Using System V's networking facilities and providing a generalization of the name space for message queues, it is possible to provide a robust

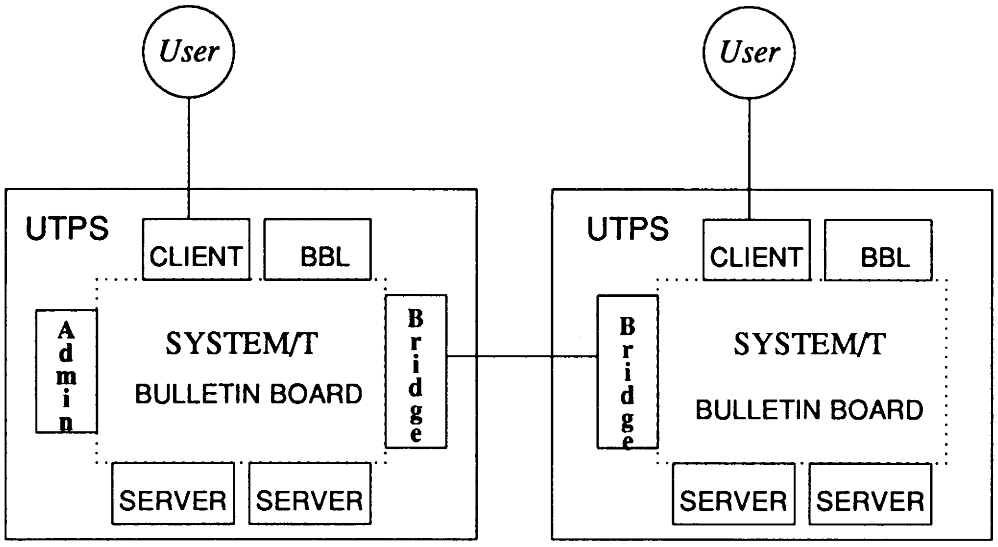


Figure 2. Tier-2, Two Node System /T Configuration

inter-machine messaging facility. The key implementation vehicle is a set of cooperating bridge processes which act as message forwarders. Since the bridges utilize reliable transport mechanisms, such as Systems V's TLI, the effect is to provide a "reliable datagram" service between client and server processes on different machines.

As in the SMP case, services are requested by name. To application programmers, the network is invisible (as are message queues in the SMP case). When a client requests a service, System /T selects a server by using the local copy of the BB, and then sends its request message to the selected server, using the bridges when the server is not co-located with the client. Likewise, System /T routes the reply to a service request to the originating client, whether it be local or remote.

2.2 DTP

The distribution of client/server interactions across the Tier-2 machines heightens the need for (distributed) transaction control. Transactions [Bernstein] provide a method to encapsulate a set of actions into a single atomic unit of work. This unit of work either wholly succeeds or has no effect. The results can be used to advance a set of distributed, logically related resources, e.g. data base systems, from one consistent state to another. One way to provide transactions for the Tier-2 machines is to implement the model of transaction control described in [X/OPEN-1]. In this model, a Transaction Manager (TM) coordinates transactions throughout a set of computers by providing communications paths with transaction semantics, and by interfacing to Resource Managers, e.g. DBMS systems, for the purpose of transaction control. In order to do this, the TM:

1. Generates Global Transaction Identifiers
2. Tracks sites participating in the transaction
3. Executes a two phase commit protocol when the application signals that all of the work is done
4. Executes a recovery protocol when a site is restored to operation after an outage

System /T has been extended to provide the transaction facilities according to the X/Open model. In addition to the library routines which provide the application transaction management functions, the implementation of transaction control is as a special set of administrative server processes (not shown in

Figure 2) which coordinate commit and recovery. These processes utilize data structures both in volatile and persistent memory.

2.3 TP-API

The use of a set of computers at Tier-2 places additional requirements on the application. For example, requests which effect permanent resources on multiple sites need to be grouped into transactions. If the computers have heterogeneous cpu architectures, it will be necessary to convert data types as data is exchanged. The following facilities are required of an API suitable for the high performance distributed TP interactions which occur within Tier-2:

- Transaction control
- Client/Server Communications
- Presentation Services (data conversion)

System /T provides these functions to applications running on Tier-2 through a TP-API called the Application Transaction Manager Interface (ATMI).

2.3.1 ATMI Transactional API The transaction control functions of ATMI allow an application to delimit a series of requests as comprising a single unit of work, called a transaction [Bernstein]. Generically, they consist of the procedures to begin work, signal completion of work, and undo work. In ATMI the functions which provide these services are called tpbegin(), tpcommit() and tpabort(), respectively.

2.3.2 ATMI Client/Server Communications API The client/server communication functions of ATMI allow the invocation of distributed services by name. Providing requests by name provides location independence of the requester from the server, thus allowing the server to be relocated without compromising the requester's ability to direct requests to it. Useful request/response paradigms include:

- synchronous calls
- asynchronous calls
- one-way calls

Synchronous calls block until the results are returned and are used when the requested results are required immediately, Asynchronous calls may be used to improve throughput when an application has several operations which may be performed in parallel. One-way calls are used when the results of the operations need not be known by the requester. In ATMI these services are provides by the functions tpcall() and tpacall().

Figure 3a depicts the standard stacking paradigm of request/response interactions. Here, SERVER 1 receives a request, does some processing, and calls SERVER 2 to do some more processing. While SERVER 2 is processing, SERVER 1 is blocked, waiting for its reply. When this is received, SERVER 1 replies to the requester. Efficiencies may be gained in TP applications by providing facilities for a "bucket brigade" style of processing, as depicted in Figure 3b. In particular, if SERVER 1 has no more processing to do after calling SERVER 2, it can drop out of the request processing, and pass responsibility for responding to the requester to SERVER 2. SERVER 1 then becomes free to handle other requests. ATMI provides this paradigm via the function tpforward().

2.3.3 Communicating Transactions A key concept in the ATMI model is that transactions accompany communications. For example, if a client begins a transaction, and then communicates with a server (e.g. by issuing a tpcall() function), the work done by the server becomes part of the transaction started by the client. Likewise, if the server makes requests of other servers, their work is also encapsulated by the transaction. In effect, transactions are propagated to all called services, whose work is then either committed or rolled-back when the originator calls tpcommit() or tpabort(), respectively.

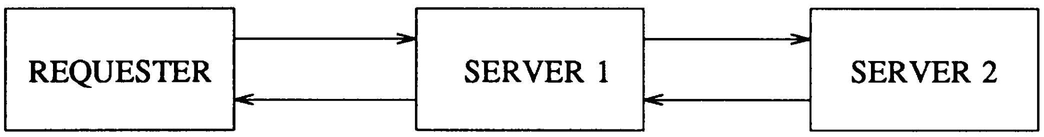


Figure 3a: Stacked Requests/Replies

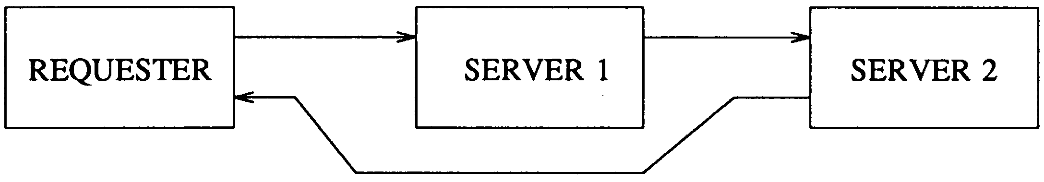


Figure 3b: Forwarded Request/Reply

2.3.4 ATMI Presentation Services While it is desirable to allow for Tier-2 to be comprised of computers of different cpu architectures, it is also desirable:

1. to minimize those differences for application programmers
2. perform conversions only as necessary

The messages sent between requesters and servers by ATMI calls are images of "typed buffers" [X/Open-1]. A typed buffer is a buffer with an associated string-named handle, called its "type". A typed buffer is created via a call to the function tpalloc(), and is destroyed by a call to the function tpfree(). When a message is sent, the type of the associated buffer is used to select a conversion function. Likewise, the type is used to invoke an "unconversion" procedure when the message is dequeued in the server. Typically, the supplier of a type, e.g. a system's programmer, provides its conversion routines, so that its user, e.g. an application programmer, can use it in client/server calls without regard to the architectures of the communicating machines.

System /T provides several built-in types, including character arrays, null terminated ASCII strings, C structures, and an attribute-value abstract data type called a Field Manipulation Language (FML) buffer. This latter type is a kind of heap data structure, in which elements are referenced by name. Built-in types, except character arrays, are automatically converted when passed between machines of dissimilar architecture. Character arrays are passed through without any conversion. Applications are free to add their own buffer types, but in so doing must supply the associated conversion functions.

System /T calls conversion functions only when source and destination machines are of different architecture, as indicated in a configuration file. Thus, if all of the machines at Tier-2 are of the same architecture, no conversion will be done for exchanged data.

2.4 Administration

Although Tier-2 consists of a network of machines, it is often required that they be administered as a unit, allowing an administrator to tend to the entire system from a single terminal. Such administration would typically consist of booting or shutting down the system, monitoring its performance, adjusting parameters, making backups, etc.

3. TIER 1: INCORPORATING WORKSTATIONS

There are several reasons to incorporate workstations in a TP environment. One of the most important is to offload CPU processing for a human interface. The asynchronous terminal, the traditional UNIX input device, imposes a significant burden on the UTPS for TP applications. The reason for this is that each input character requires the servicing of an interrupt. Additionally, most TP input is forms-oriented, and since forms packages most often read the screen in "raw mode", the forms handler, an application program, must be scheduled by the operating system on a per-character basis. The CPU overhead to accommodate such processing is enormous, perhaps consuming as many cycles as the rest of the software combined, including application, dbms, networking protocol and operating system logic.

Workstations, on the other hand, can be used as block mode devices, a type well suited for TP system input. In this mode, an entire message is received with one interrupt. In addition to relieving the UTPS of per-character processing, the cpu and memory provided by the workstation can be used to provide alternate forms of interface, including GUIs.

The model of workstations assumed in ETP is that workstations are intelligent devices (i.e. ones with cpu and memory) that are requesters of TP services. They need not be machines running the UNIX Operating System, but must be capable of generating the protocol required to talk to a Tier-2 machine. In the ETP model, workstations are assumed to contain no shared or persistent resources, such as databases, do not themselves offer any services, are personally administered, and have no security features.

3.1 Gateway to Tier-1

An important goal for ETP is to allow the connection of the large numbers of users typical of proprietary TP systems to Tier-2 machines. The offloading of the cpu cycles for the forms interface is not sufficient to provide this connectivity. It is usually the case that each user logged on to a UNIX system has one or more processes attached to his or her terminal. The context associated with these processes, including memory, file descriptors, process table slots, etc., is unacceptably large. What is required is a method of connecting many TP terminals with much less context. One way to provide the needed functionality is by providing a special gateway process, called the Workstation Gateway (WSG) to provide communications with the workstation community.

Figure 4 shows the architecture of WSG. WSG is a multistated process, which provides connectivity for many workstations with a minimum amount of context per workstation. Its primary job is to act as a surrogate client for the "real" client software modules, which are executing on the workstations. As a surrogate for many workstations, WSG cannot afford to block while waiting for replies to service requests, and must be specially constructed to handle the blocking calls of its connected workstations.

3.2 Workstation TP-API

One way to integrate the tiers of ETP is to provide the same TP-API on them, when practical. Unlike Tier-3 machines, workstations have not traditionally been used for TP applications, and do not have existing TP-APIs. A natural choice then is to provide the Tier-2 TP-API on Tier-1 machines. However, as workstations serve only a requester role in ETP, the API provided on them need only be the "client side" of TP-API. For Tier-2 configurations running TUXEDO System /T, this means providing the client calls of ATMI, called WS-ATMI, for the workstation machines. WS-ATMI includes transaction demarcation and control functions (tpbegin, tpabort, and tpcommit), typed buffer manipulation functions (tpalloc and tpfree), and service request functions (tpcall and tpacall). The client/server paradigm of Tier-2 is thus extended to Tier-1 machines via a uniform API for service requests. As a set of library routines, WS-ATMI may be used with a variety of forms and graphics packages executing on the workstations to inject inputs into, and receive outputs from Tier-2 machines.

3.3 WS Transactions

The role of workstations in transaction control needs particular attention. Since workstations are considered to be personally administered, and thus may be turned off for extended periods, they should not be counted upon to provide transaction coordination for two-phase commit [Bernstein]. Instead, when an application on a workstation calls commit, transaction coordination needs to be delegated to a Tier-2 machine.

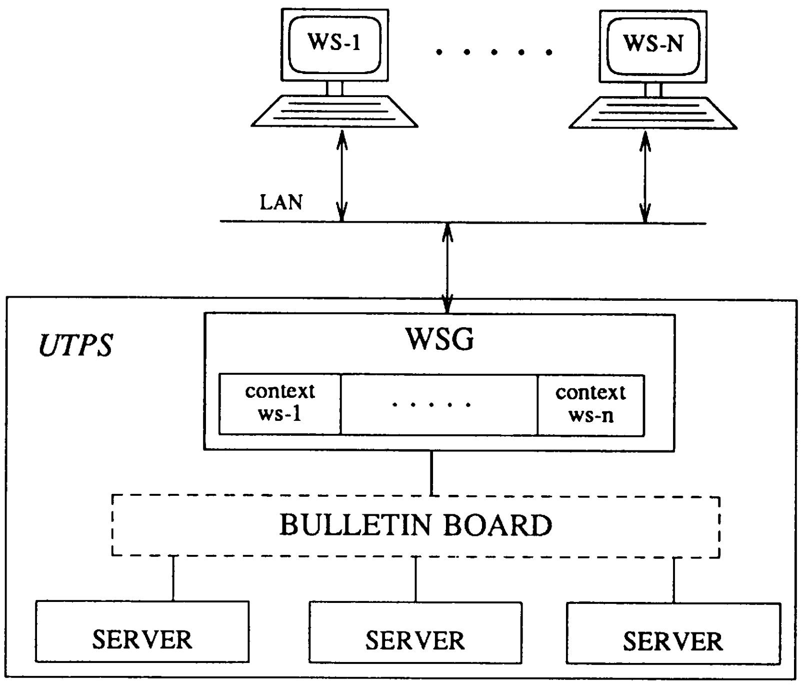


Figure 4. Workstation Gateway (WSG) - Multi-stated Client Surrogate

3.4 Tier-1 Administration

Although workstations themselves are considered to be personally administered, their connection to an ETP system should be subject to the administration of that system. In particular, an ETP administrator should be able to determine activity to/from the workstation, enable/disable its connection to the system, and advise it of abnormal conditions (e.g. imminent ETP system shut-down). A natural way to do this is to have WSG provide surrogate administrative services.

4. TIER 3: INCORPORATING PROPRIETARY TP SYSTEMS

As mentioned in the Section 1, the bulk of commercial TP processing is currently handled by proprietary TP systems. As TP users and TP vendors incorporate UNIX based solutions, there will continue to be a need to access the programs and data on proprietary TP systems. Overall, the approach taken to accommodate Tier-3 machines into an ETP system is to provide gateways from Tier-2 machines to Tier-3 machines. Within a given ETP system it is entirely likely that multiple heterogeneous proprietary systems may need to be incorporated. Such a scenario is depicted in Figure 5. In this figure, two Tier-2 machines (UTPS-1 and UTPS-2) are connected to three Tier-3 machines (no Tier-1 machines are shown). UTPS-1 is connected to two 370-compatible mainframes (PTPS-1 and PTPS2), each running an instance of the CICS Transaction Monitor [IBM-1]. UTPS-2 is connected to PTPS-2, and in addition is connected to PTPS-3, a Tandem computer running the PATHWAY Transaction Monitor [Tandem].

4.1 Gateway to Tier-3

In the ETP model, Tier-3 machines are considered to be servers. The basic paradigm of request/response is extended from Tier-2 to Tier-3 via UTPS-PTPS Gateway processes (UPGWs), depicted in Figure 5. By advertising proprietary services on Tier-2 machines, UPGWs can act as surrogate servers for the "real" servers, which reside on the proprietary system. Requests for proprietary application services, which appear to the system to be processed on Tier-2, are really forwarded by the UPGWs to servers on the proprietary system. To do this, the UPGWs need to have access to mappings of local service request name to proprietary server name, and a method of transforming data to the

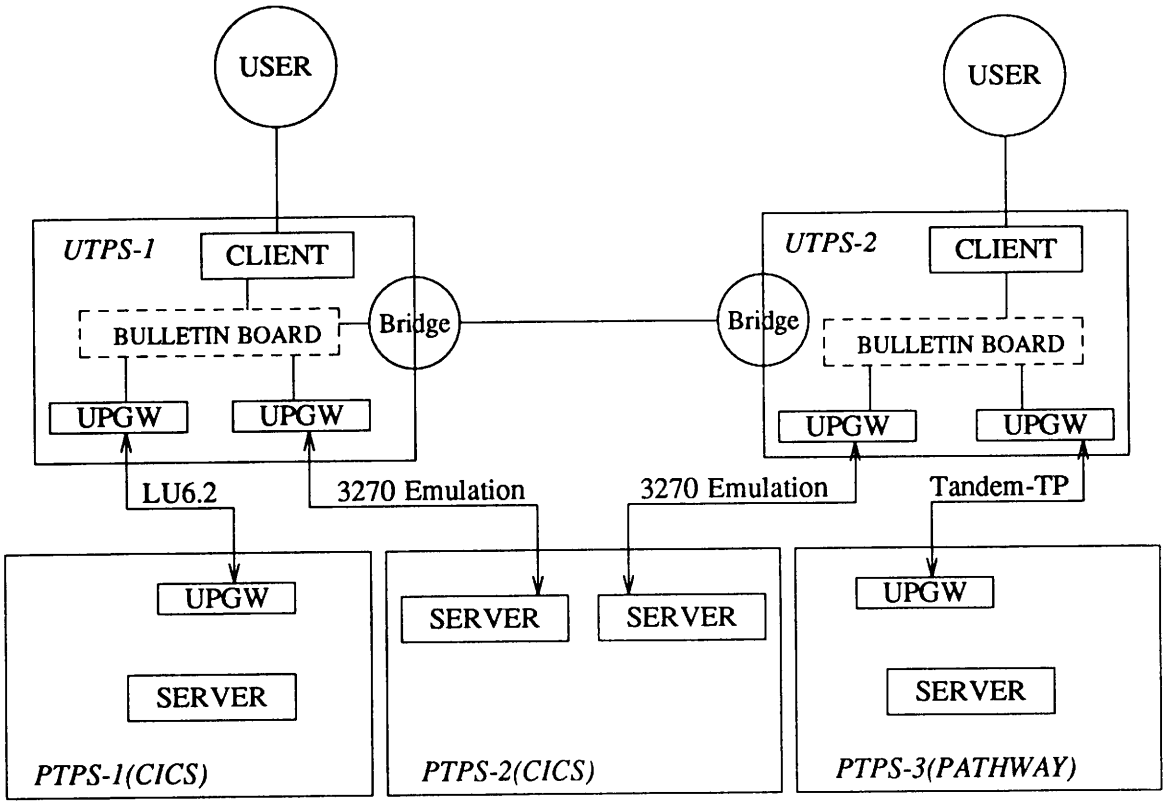


Figure 5. Multiple Heterogeneous Tier 3 Systems

format of the proprietary systems.

The actual method of interface to the proprietary system is encapsulated within each UPGW. For gateways which interface to the proprietary system via terminal emulation, only a Tier-2 side gateway need be written. In this case, servers on the proprietary side are really terminal-bound processes, and the gateway needs to convert inputs and outputs to terminal format via a mapping language. Such a case is shown between the Tier-2 machines (UTPS-1, UTPS-2) and the PTPS-2 machine in Figure 5, where the protocol is 3270 emulation. For gateways which interface to the proprietary system via a program-to-program interface, for example IBM's LU6.2 [IBM-2], it is likely that a peer gateway on the proprietary side needs to be provided to yield the request/response paradigm available on Tier-2. This case is depicted in Figure 5 for the PTPS-1 and PTPS-3 machines, each of which has a gateway partner for the gateways on the connected Tier-2 machines. Of course, it should be possible to mix both program-to-program and terminal emulation encapsulations, even to the same machine, within one application.

4.2 Tier-3 API

The location transparency of requests originating on Tier-1 and Tier-2 machines means that Tier-2 processes should not be aware that their requests are processed on a proprietary mainframe. The service could migrate from Tier-3 to Tier-2, and the requesting module should see no difference.

Several choices for an API on the Tier-3 machines themselves are present:

- Provide the same API as in server modules of Tier-2.
- Accommodate the semantics required of Tier-2 interactions, but in a syntax more natural to the proprietary system, e.g in its native TP API.

Unlike the workstation case, where there are not existing APIs for TP, the proprietary TP systems have APIs for TP. So, the choice here is not an obvious one (i.e. provide the same API as on Tier-2), and may depend on many factors, including the ease of implementation and the acceptance of a new API on the proprietary machine. Whatever the choice, automatic conversion of data formats is highly desirable.

4.3 Tier-3 Transactions

It is also highly desirable to have transaction semantics available across the Tier-2/Tier-3 boundary. For example, this allows Tier-2 and Tier-3 database updates to be bound into an atomic unit of work. The protocol for transaction control with a Tier-3 System will depend on that system, and, in general, will be proprietary. For those proprietary systems whose protocols are compatible with the two-phased commit with presumed abort protocol [Mohan], the interface to transaction control can be provided by the XA interface described in [X/Open-2]. To do this, all or part of the proprietary system could be regarded as a Resource Manager, and XA calls are made on the Tier-2 system by the UPGW (or other administrative processes) to drive the transaction protocol. The implementation of XA for the proprietary system is split between the invoking Tier-2 machine and the associated Tier-3 machine. Note that it is likely the case that transaction semantics would only be supported for gateways whose protocol is program-to-program.

4.4 Administrative Integration

It is also highly desirable to provide the administrator of an ETP system with tools to determine the status of the Tier-3 machines which are incorporated into the ETP system. A natural way to provide this is to have the gateway processes also serve as administrative surrogates for the Tier-3 machines. In this case, each UPGW is responsible for booting any software needed on the Tier-3 system for interactions with the Tier-2 system. Likewise, UPGW is the vehicle by which Tier-2 informs Tier-3 that Tier-2 is shutting down. Finally, UPGW also responds to administrative requests as to the status of the Tier-3 System. In this latter regard, it will generally be impossible to keep transparency, and the Tier-2 System will need a method to allow commands specific to the proprietary system to be passed through via UPGW to the proprietary system.

4.5 UPGW Instantiations

In addition to pairwise instantiations for particular proprietary TP systems, two instantiations of UPGW are of particular interest:

1. ISOTPGW. As ISO/TP [ISO/TP] moves towards reality, vendors will begin to make it available in their proprietary environments. Such a protocol then becomes the preferred method for interacting with proprietary systems. Basically, a generic gateway type, ISOTPGW, will be able to accommodate interactions with all ISO/TP conforming proprietary systems, although some customization may be needed for administrative purposes.
2. UUGW. A particular instance of a Tier-3 System might not be a proprietary TP system at all, but rather another Tier-2 System. Since Tier-2 systems have a well defined input port, the WSG described in section 3.1, an instance of UPGW, called UUGW, can be constructed by using the implementation of WS-ATMI for a UNIX workstation. Its partner is the WSG. A set of such gateways working in the opposite directions can provide complete connectivity between the two ETP "domains". The result is that it is possible to create a very large TP system composed of domains of administratively autonomous ETP systems. The domains are joined at the Tier-2 level. The construction of such large systems begins to require the application of more advanced techniques, including the use

of a standards-based naming service and the incorporation of standards-based administrative services.

5. ETP Status

The work reported in [Andrade] describes a method of building an SMP TP System on the UNIX Operating System. Release 4 of TUXEDO System/T [AT&T], currently available from a variety of hardware and software vendors, extends this work to provide a full, networked Tier-2 TP System. As in the SMP case, no modifications of the UNIX Operating System were necessary to produce the Tier-2 system. The development of a full three-tiered ETP system is currently in progress, and will be available in a forthcoming release of the TUXEDO System. The construction of very large systems based on Tier-2 Gateways (UUGW) is under investigation.

6. SUMMARY

An architecture which integrates multiple levels of computer processing into a complete TP system has been presented in this paper. The architecture provides for the migration of proprietary TP solutions to those centered around a network of computers running the UNIX Operating System, and allows users to take advantage of the trend towards decentralized operations, while protecting their investment in proprietary TP systems. Elements of this architecture are currently available, and others are under development.

7. ACKNOWLEDGEMENTS

I would like to thank the members of the TUXEDO project for their ongoing efforts in the realization of the ETP architecture described in this paper. Mark Carges, Jane Dwyer, Howard Elder and Glenn Rose provided valuable comments on drafts of this paper.

8. REFERENCES

Andrade	J. Andrade, M. Carges, and K. Kovach, "Building a Transaction Processing System on UNIX Systems", 1989 UniForum Conference Proceedings, pp. 167-176.
AT&T	AT&T, "TUXEDO® System Release 4.0 Product Overview", 1990.
Bernstein	P.A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in database Systems", Addison-Wesley, 1987.
IBM-1	IBM, "Customer Information Control System CICS/DOS/VS, Application Programmer's Reference Manual (Command Level)", SC33-0077-5, Sixth Edition, July, 19876.
IBM-2	IBM, "Systems Network Architecture, Format and Protocol Reference Manual, Architecture Logic for LU Type 6.2", SC30-3269-3, Fourth Edition, December, 1985.
INFORMIX	INFORMIX, Inc., "Informix-SQL Users Guide", October, 1986.
ISO-TP	ISO/TC 97/SC 21C N 2274, "Information Processing Systems-- Open Systems Interconnection -- Distributed Transaction processing -- Part 3: Protocol Specification," March 1988.
Landis	Ken Landis, "UNIX Calms Wall Street Chaos", CommUNIXations, August, 1990, p.22.
Mohan	C. Mohan, B.G. Lindsay, R. Obermarck, "Transaction Management in the R* Distributed database Management System," ACM Transactions on Database Systems, December 1986, vol. 11, no. 4, pp. 378-397.
ORACLE	ORACLE Corporation, "SQL Language Reference Manual Version 6.0", February, 1990.
Tandem	Tandem Computers, "Introduction to NonStop SQL™", March, 1987.
X/Open-1	X/Open Company Limited, Transaction Processing Working Group, "Interim Reference Model for Distributed Transaction Processing", July 7, 1989.

X/Open-2 X/Open Company Limited, Preliminary Specification, "Distributed Transaction Processing: The XA Specification", April, 1990.

International Language Support in X11 Release 5: Building a Standard for Internationalized Heterogeneous Network Computing

Glenn Widener
Tektronix, Inc.
P.O. Box 1000 MS 60-850
26600 SW Parkway
Wilsonville, OR 97070
glenn@orca.wv.tek.com

International Language Support in X11 Release 5: Building a Standard for Internationalized Heterogeneous Network Computing

Glenn Widener
Interactive Technologies Division
Tektronix Inc.
glennw@orca.wv.tek.com

ABSTRACT

The X Consortium has recently adopted a new version of the Xlib and Xt C programming interfaces. This paper gives an overview of the X internationalization architecture and the relevant changes to Xlib and the Xt toolkit intrinsics planned for Release 5 of X11. The paper also discusses some of the requirements and design issues that lead to the design for X internationalization, as well as issues that are not addressed in the new X standard. In particular, problems with multilingual support and heterogeneous network environments are discussed.

Xlib now supports the ANSI C internationalization architecture, and is based on the ANSI C setlocale() function which configures the C system library for locale-specific processing. Xlib provides new interfaces for obtaining localized text from the keyboard, drawing localized text with X fonts, obtaining localized resource values, and communicating localized text to other clients of the same X display. The current locale (configured by calling setlocale()) is used by Xlib to determine the required keyboard input methods, fonts, resource files, and codeset conversions to implement the new Xlib functions in a locale-specific manner. The Xlib implementor and system administrators are responsible for supplying locale definitions that are consistent with the locale definitions supplied for the host C system libraries, and for mapping them to locales and codesets used for communication with clients executing on other hosts.

This paper presents the concepts behind the design of X input methods and text drawing. Asian input methods and the model for obtaining the fonts required for a given codeset are described. Issues regarding support of multiple locales and displays and font availability in a heterogeneous environment are discussed.

X INTERNATIONALIZATION GOALS

The existing X Window System Version 11 standard, supported by the MIT X Consortium, provides only minimal support for native-language text processing. In general, localization requirements (that is, adaptation to the language and customs of a customer in a non-U.S. locality) must be addressed by the X11 client developer.

Late in 1989, the X Consortium member companies reached a concensus that the time had come to introduce internationalization technology to X. Internationalization ("i18n" for short) is defined by the X/Open Portability Guide, Issue 3, as "The provision within a computer program of the capability of making itself adaptable to the requirements of different native languages, local customs and coded character sets." In simpler, more practical terms, internationalizing is making a computer program adaptable to different locales without program source modifications or recompilation. In most standards-based implementations, i18n means that the programming interfaces to which an application are written are independent of the locale in which the application runs, and that the libraries implementing those programming interfaces are adaptable at runtime to any locale of interest, typically by loading a database that defines the behavior for the desired locale. This database includes such things as error messages in the native language, character types (e.g., "isalpha"), the format for printing a date or numeric value, and collation sequences. Localization, therefore, is this process of adapting the library to the locale.

The reality of internationalization is that a standard programming interface that covers all of the localities in the world does not yet exist. In defining the goals of X i18n for Release 5, the X Consortium specifically elected not to consider the requirements of certain locales, particularly those that require contextual text rendering and bidirectional text support, such as Hebrew and Arabic, or vertical text rendering in Asian countries. These omissions were based on the perceived complexity of these problems, the desire to obtain a basic i18n architecture as soon as possible, and the lack of a strong market requirement for these locales by most Consortium member organizations. Other locale dependencies, such as displaying cascaded menus right-to-left when the text rendering direction is right-to-left, or using different graphical icons in different locales, are best addressed in user interface toolkits, which are not currently part of the X standard.

DESIGN CONSTRAINTS AND APPROACH

As a change to an existing, widely used system which is in the process of standardization, the design of i18n in X must live within certain constraints. Because the X protocol is undergoing ANSI standardization, it was concluded that no protocol changes were permissible. For example, the protocol specifies the ISO 8859-1 character set ("Latin-1", ASCII plus Western European characters) for many text values such as Atoms, font names, and color names. While changing this to a locale-independent encoding would be useful, it was not allowed. Further, it was desired for the first phase of i18n that no protocol extensions would be required. This contributed to the omission of vertical text support, since the X11 protocol does not support vertical text drawing.

The X i18n design team concluded that the right approach to i18n was to modify and extend the C-Xlib interface so that any Xlib client code could be made locale-independent, and in turn to modify the Xt toolkit intrinsics to use the i18n Xlib interface. The specification for these changes should be completing public review in January 1991, with the objective of implementing the new standard in Release 5, to be available in 1991.

INTERNATIONALIZATION STANDARDS: THE LOCALE MODEL

The best known and most widely implemented standard for i18n is ANSI C. Because of this widespread usage and the lack of any widely used alternative technology, the X Consortium concluded that it was appropriate to base X i18n on ANSI C.

ANSI C introduces to the C language the notion of a "locale", which is the run-time environment of a computer program defining the locale-specific behavior of that program. Each locale is identified by a name, which is passed to the ANSI C setlocale() function. Setlocale() initialize the locale for all locale-dependent C system library calls, globally for the calling process. The name typically incorporates a language, a geographical territory, and a codeset (the coded character set in which text is processed).

A key feature of the locale model is codeset independence: application code should not make any assumptions about how characters are encoded. Even in the same locale, each C library implementor is free to choose a different encoding. Later in this paper, we will discuss the consequences of this freedom for heterogeneous network computing, particularly in the X environment.

The burden for implementing locale-specific processing thus falls on the vendor of the C library, and on system administrators who define locale-specific processing characteristics using the database-driven facilities provided by the C library vendor. Many terminal-based applications have been internationalized for Western Europe and the Far East using this system.

The Consortium decided that X could use the setlocale() function directly as the means of configuring X library functions for a locale. The goal is that the application can freely pass text strings between C library functions and X functions. Direct use of setlocale() was motivated by a desire for simplicity, and to avoid duplicating interfaces that could be expected to be supplied by other system software.

Avoiding duplication may prove difficult in one area — the implementation of the definition of the locales inside Xlib. Xlib functions must have access to the complete definition of the codeset of the locale in order to map characters to font glyphs. However, there are few standards in existence for the tools to define C library locales and codesets or the databases those tools produce. As a result, the sample Xlib implementation must provide an independent mechanism for codeset definition that is portable to any C environment, yet allow host software vendors who are supplying both the C library and the X library to integrate this mechanism into their existing C library locale definition system.

The Release 5 Xlib sample implementation will provide a general, portable mechanism for localization, and a number of sample locale databases. For systems that do not support ANSI C, the sample implementation will provide a minimal setlocale() implementation, sufficient to support X localization.

MORE DESIGN CONSTRAINTS: API COMPATIBILITY

Upward binary and source compatibility of the Xlib and Xt application programming interfaces is required. Several Release 4 Xlib functions pass the Latin-1 protocol strings mentioned above directly to or from the caller. The i18n design has to take into consideration both existing Xlib implementations that strictly obey the specification of Latin-1 for these functions, and existing Xlib implementations on non-ASCII-based systems that convert between the host codeset and Latin-1. The latter implementations, while strictly non-compliant, correctly anticipate the X i18n design! At this writing, this compatibility problem is avoided by

- narrowing the character set specification to a portable subset of the ASCII character set, called the "X Portable Character Set",
- assuming that these characters are encoded the same in all locales on a given host (not necessarily as ASCII), and
- leaving the behavior for other characters implementation-defined.

While this assumption about identical encoding of a basic character set in all locales is not guaranteed by ANSI C, it is true for all implementations of which the Consortium is aware and is proposed for the "Portable Filename Character Set" in a new POSIX draft under review. One can imagine the chaos that would result if ASCII characters commonly used in filenames were encoded differently in different locales! But the fact is, even with the proposed POSIX restriction, this chaos can still exist in a heterogeneous networked file system, if all systems do not use an ASCII-based encoding for filenames.

ARCHITECTURE OF AN INTERNATIONALIZED X CLIENT

Figure 1 summarizes the basic architecture of an X11 client, and identifies the major subsystems in which localization will be implemented and the major localization features managed in those subsystems.

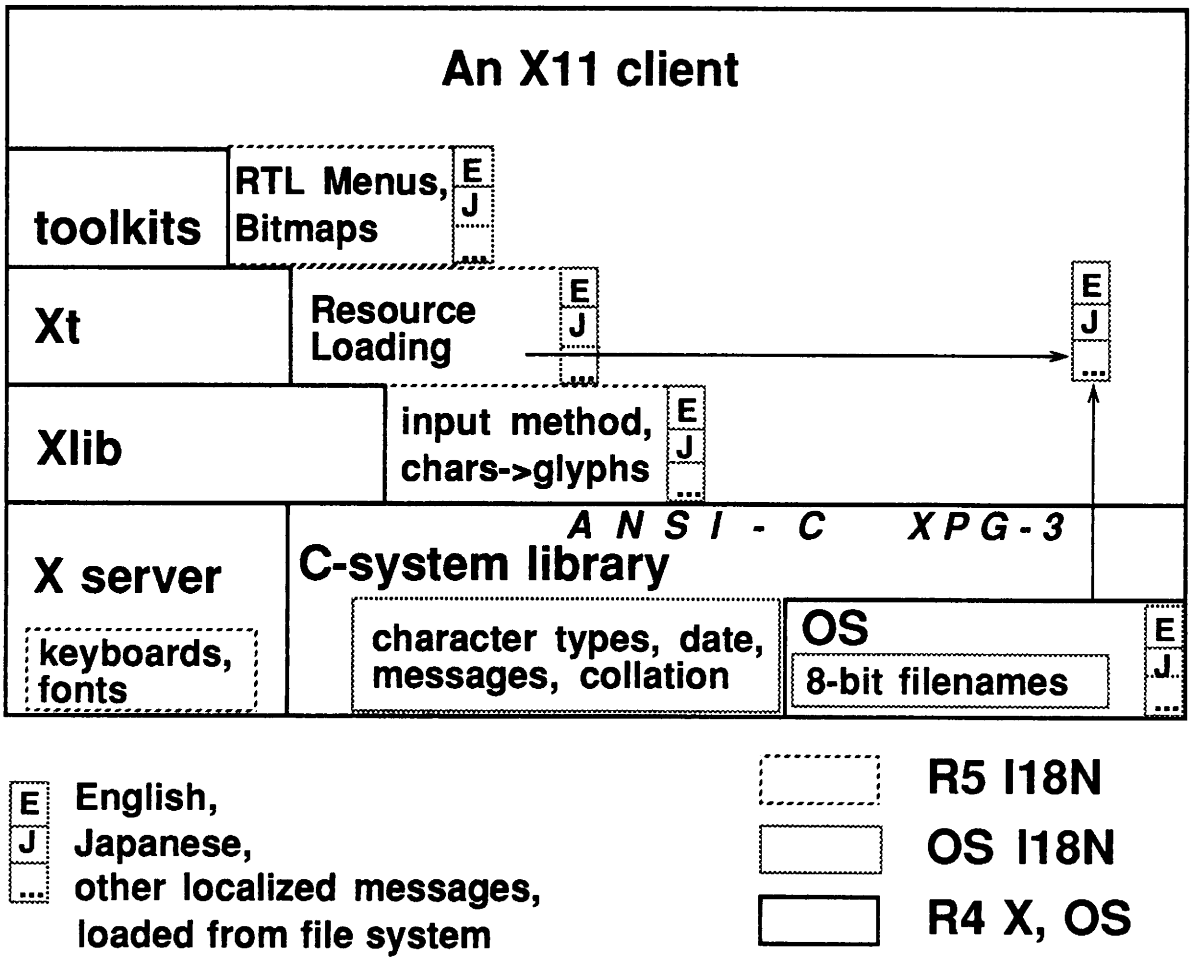


Figure 1 - Internationalized X Client Architecture

ANSI C defines the primary programming interface to the C-system library, although many implementations offer additional interfaces, particularly for localization of error messages and other text that is usually embedded in non-internationalized clients. The X/Open Portability Guide, Issue 3 is an ANSI C based industry standard which offers message localization. Note that an X client will have a choice of using system library or X resource interfaces for localization of messages. As of this writing, the X resource mechanisms are split between Xlib and Xt. At this writing, a proposal from the author is under review within the X Consortium to move the Xt resource loading conventions into Xlib, so that a consistent mechanism is available for all clients, including those not based on the Xt intrinsics. X toolkits, and Xlib itself, will use X resources to obtain localized messages.

X TEXT PROCESSING

Within Xlib, the locale affects only processing of text. To understand how X text processing is internationalized, let's first examine the X Release 4 text processing model, without full i18n, then describe how i18n text is to be processed in X.

Figure 2 shows an X client which starts up, reads some command line options, possibly some text from files, and opens a connection to the display through Xlib. It will typically create an X resource database from textual resource values kept in files and on the X server. Resources include such things as named colors, menu labels, fonts, and bitmaps. In Release 5, these resources will be loaded from a file whose name is constructed from the locale name set by setlocale() and whose contents are in the codeset of the locale.

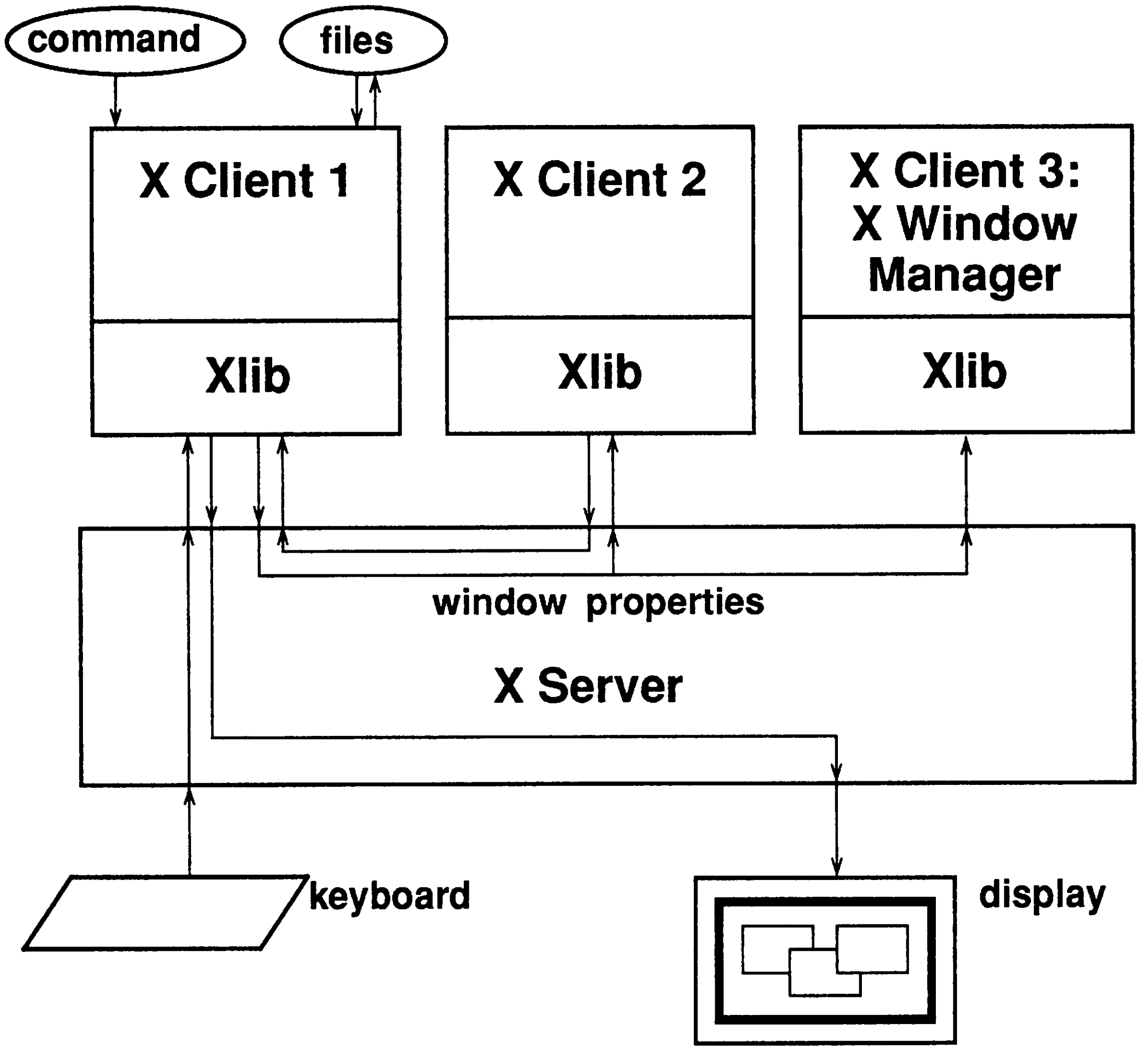


Figure 2 - Text Processing in X

In creating its windows, the client puts a number of textual properties such as window and icon names on the window for the benefit of the window manager.

Once the necessary resources have been processed, the client initialized, and its X windows created, the client can begin accepting text from the keyboard and printing text in windows, using a font provided by the server and selected by a resource value. The next two sections will describe how this is internationalized.

Finally, X supports a general mechanism of selections, that allows data to be passed between clients under the control of graphical selections by the user. This data includes any permanently visible text. Both this text and the window properties should be communicated in the multi-character set encoding "Compound Text", defined in Release 4. Release 5 Xlib will contain utility functions to convert localized text to/from Compound Text.

TEXT INPUT I18N

To obtain text typed at the keyboard, the X client removes each successive hardware-dependent KeyPress event from the Xlib event queue and passes it to the Xlib routine XLookupString. This function returns a 32-bit hardware-independent "KeySym" value that represents the glyph on the keycap, and, if meaningful, a string that corresponds to the pressed key. Depending on context, the client will either use the KeySym to trigger key-driven actions or append the string to a text buffer.

On English keyboards there is a one-to-one correspondence between keystrokes and characters (ignoring case). Unfortunately, it is almost the only language where this is the case! Most European languages require support of diacritical marks, which typically require two or three keystrokes to generate a single character. X11 Release 4 allowed Xlib implementors to support diacritical marks and other "Compose sequences", internally to the XLookupString function.

Asian writing systems based on the Chinese ideographic character set, which has thousands of characters, present far more complex input problems. Typically, users must input a phonetic form, such as Hiragana or Katakana in Japan, then use an interactive, highly sophisticated interface called an "input method" to convert a sequence of phonetic characters into ideographs. The method is interactive because in general the conversion cannot be accomplished without human intervention. Typically, the input method offers the user a choice of several possible equivalent ideographs, and the user selects the desired one, in a "pre-edit" dialog.

A large amount of research is ongoing in Asian countries to improve the efficiency of input methods, applying AI technology to reduce the frequency with which the user is forced to select the desired ideograph. These input methods are large, complex programs, and typically are shared between multiple applications using some form of inter-process communication.

However, in a windowed environment, it is desirable that the overall user interface appear seamless, with the pre-edit dialog being conducted in close proximity to the text entry point. Ideally, the phonetic pre-edited text would appear at the text insertion location, rendered identically to the surrounding text. This is called "on-the-spot" pre-editing. Because of the complexity of having the application render the pre-edit text, many designs have used a separate window or a sub-window of the client's window to display the pre-edit dialog.

A new input routine, XmbLookupString, extends XLookupString to support complex input methods. The client creates an opaque "XIM" object in the current locale to represent the input method, creates an "IC" (Input Context) object to represent a particular text input dialog, and calls XmbLookupString with the IC and each successive KeyPress event. Prior to calling XmbLookupString, the client will interact with the XIM and IC to configure the input method for the desired pre-edit style from among those that the input method supports, and to set various user interface resources so that the input method user interface is consistent with the application user interface. In addition, if the client wishes to support "on-the-spot" pre-edit, it must supply the IC with callback functions that implement the pre-edit text rendering.

The input method API is more complex than can be described here. In some architectures the input method will intercept the key events before they arrive in the client's input queue, or will need to steal certain non-key events prior to client event dispatch. Interested readers should obtain the input method specification listed in the references to learn more about input method architectures supported by X i18n. Figure 3 should give an idea of the potential complexity.

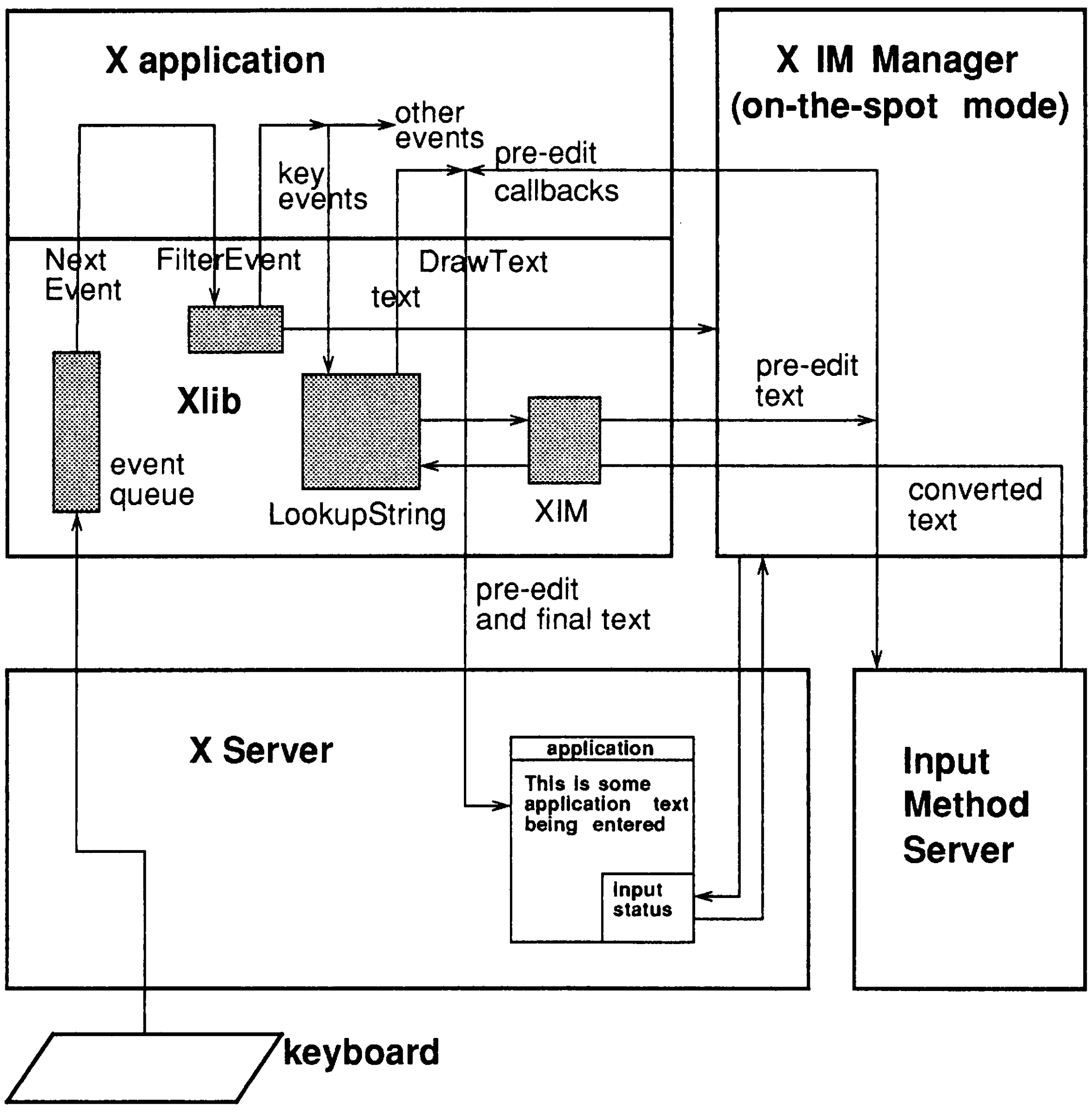


Figure 3 - I18N Text Input

TEXT DRAWING I18N

Release 4 X makes a basic assumption that a codeset (an encoding in which text is processed and stored) is the same as a charset (an encoding in a font). Clients assume that they can open a single X font specified by the user and call Xlib text drawing functions with the font and an arbitrary text string. This assumption is tolerable for Western European languages, but becomes impractical in writing systems with a very large number of characters, such as the ideographic writing systems of the Far East, or in writing systems where there is not a one-to-one correspondence between each "coded character" and a single glyph used to image that character. For many languages in the latter class, such as Arabic or Thai, the glyph used for a character depends on the surrounding text, so that text rendering is context-dependent. Even Western European languages can be context-dependent if ligatures are needed, or if accented characters are represented in the stored text as two separate characters, the accent and the alphabetic character.

To address these languages, Xlib i18n introduces the concept of a "font set", represented by an XFontSet opaque object. A new set of Xlib routines to draw text and obtain text metrics takes an XFontSet instead of a single font. Clients create an XFontSet within the current locale, based on a list of "base font names" specified by the user. Conceptually, a base font name identifies a family of fonts of a similar typeface, each font containing a set of glyphs useful in a particular locale or set of locales. X uses a structured font name format called "X Logical Font Description", which identifies each font property such as typeface, point size, style, or character set by a specific field. A typical base font name specifies all properties except the character set, allowing the locale definition in Xlib to determine which character sets are required.

Ideally, only one base font name would be required to support all locales. However, current practice is to assemble a set of fonts from a variety of sources to cover a given locale. For example, Japanese is frequently supported by one 8-bit font for ASCII, another 8-bit font for Japanese phonetic characters, and a 16-bit font for Kanji. Each might require a separate base font name to be identified. The base font name list is ordered, and Xlib follows a specified algorithm to search the list to obtain a font for each required character sets. A system administrator could specify one base font name list that covered many locales, with Xlib opening only the needed fonts.

By isolating the client from the actual fonts, character sets with greater than $2^{\wedge 16}$ characters, such as Traditional Chinese, can be supported. Also, server memory resources can be conserved by splitting a large character set into multiple fonts, with a font being loaded into the server only if a character from that font is actually rendered.

PROBLEMS WITH APPLYING THE LOCALE MODEL TO X

MONOLINGUAL OR MULTILINGUAL?

Implicit in the locale model are several assumptions about i18n. One of the most important is that a given process handles a single locale throughout its life. True multilingual text processing, in which one can process a single text stream containing text in arbitrary multiple languages, taking all language dependencies into account, is not supported. While in theory, the program could call setlocale() at any time to process text in multiple locales in a single program invocation, for most C library implementations setlocale() is a heavyweight call, on the assumption that applications will call it once at program initialization time. Some limited multilingual support is provided by some vendors, for example, mixed English and Japanese text, by defining a "super-locale" which allows encoding Latin and Kanji characters in a single text stream. However, there is no provision in this model for handling locale dependencies that are independent of the character set, such as the format for printing a date or locale-specific collating sequences. The ANSI C locale model supports multilingual only if the language can be inferred from the character.

A major perennial debate in X i18n is whether X should be emphasizing monolingual support as a clear, achievable near-term objective, or moving ahead toward the more ambitious goal of true multilingualism, Most vendors have felt that at least 90% of their customer needs are addressed by monolingual support. An increasing number of users and vendors, however, are asking for true multilingual support.

Unfortunately, standards groups in other areas than X have not yet begun to address the requirements of multilingualism, and the basic technologies for multilingual text processing are still in their infancy. The X Consortium's charter generally prevents it from standardizing new, unproven technologies. The Consortium concluded that for Release 5, the advantages of following the ANSI C locale model outweighed the existing customer demand for multilingual text processing.

However, consensus is now building that the locale model is inadequate in the long term, and that X should provide an alternative model that can support full multilingualism, at least within X. At the very least, Release 5 i18n should not interfere with clients that wish to implement multilingual text.

At this writing, a multilingual model is not fully defined, and no decision to include one in Release 5 has been made. This author believes that most requirements can be addressed by supporting a "tagged text" format, based on the multi-codeset encoding "Compound Text" currently defined by X for inter-client text interchange. Each individual locale segment would be preceeded by a tag that identifies the locale. This model has the additional advantages that the standard vehicle for inter-client text interchange would support full multilingualism, and that clients would not have to convert text strings communicated from another client to a locale in order to render them at the display, Clients may parse a tagged Compound Text format and maintain the text internally in a "structured text" form. Structured text manages text as a list of objects, each object containing a text segment with attributes (e.g., font, style, point size, and locale). Traversal is efficient, and new attributes are easily added.

LOCALE CONTEXT BINDING

Even if we ignore the needs of true multilingualism, we still have to consider the possibility of differing locales being set simultaneously in a system.

The X programming model is context-rich, with many programmer-visible objects such as displays, windows, graphics contexts, text input contexts, and fonts. One of the fundamental design decisions for X i18n was whether the locale should be strictly global and managed entirely by the client, as in ANSI C, or whether it should be bound to one or more X objects. For example, should one create a FontSet object in the current locale, validate the font availability for the locale and base font names at creation time, and ignore future changes to the global ANSI C locale, or should the validation of font availability be performed at drawing time, based on the locale at drawing time? In general, one can examine a whole hierarchy of possible locale context bindings:

- per-process (global via setlocale())
- per-display (one locale for all clients of a display)
- per-client connection (one locale for each client of a display)
- per-top-level window (or Xt Shell widget)
- per-subwindow (per Xt widget)
- per X text resource (XIM, Input Context, XFontSet)
- true multilingual (mixed locales in a single text stream)

In X, a single process may open connections to multiple displays. An example would be an interactive inter-office message system. Each display represents a different user, and each user should be free to define a locale to match his preferences. Thus, one cannot make the assumption that a given process will operate in a single locale, even if one assumes that there are no multi-lingual users!

The user paradigm in a windowed environment that is most analogous to a process in a shell-based terminal environment is the top-level window. Thus, one can argue that to provide the same level of user-visible locale granularity in X as ANSI C provides in a terminal environment, a locale should be associated with each top-level window. This would support, for example, a user whose native language is Japanese and who is reading mail written in Chinese and in English. All error messages and dialogs would be in Japanese, but the user could instruct the mail reader program to display each message in its native language in a separate top-level window. But then why should the client be forced to show each message in a separate top-level window, instead of in subwindows of a single top-level window?

In ANSI C, if a program needs to operate with more than one locale, it must manage the locales itself, calling setlocale() each time it changes locale context, since the only context in the C library is global to the process. If a multi-locale X client is also multi-threaded, it cannot depend on any global state, and must have some means of binding the locale to objects that are passed explicitly to the lower-level toolkit libraries. Since both the Xlib display object and windows/widgets are passed explicitly throughout X, they are good candidates for locale association.

The compromise reached in the X i18n design was to bind locale to the XIM (Input Method), XFontSet, and resource database objects, but provide no model for binding locale to displays or windows. This approach keeps the programming interface simple, particularly for validating font and input method availability for the locale, while giving multi-threaded multi-display applications a means of managing locales for the most important user data. At the same time, it makes no assumptions about typical locale granularity at the user interface. The Xlib context manager can be used by a client to associate locales or other locale-dependent objects with windows.

HETEROGENEOUS NETWORK ISSUES

In addition to mechanisms to manage locales within the client, there must also be conventions for announcement of the locale by the user. In ANSI C, most clients will allow the C library implementation to select the locale by passing an empty string to setlocale(). Most implementations take the locale name from the environment variable LANG. As we have seen above, even when each user uses only a single locale, that locale needs to be announced at the display for the benefit of multi-display clients. The Release 4 Xt intrinsics have support for localized resources, and provide a convention where the locale name is taken from a program command line option, else a resource value which can be defined at the server, else the C library's environment variable. This announcement hierarchy has been adopted as a recommended convention for Release 5 Xlib, but is not directly supported by the API. The author's aforementioned resource loading mechanism proposal would directly support the Release 4 Xt intrinsics conventions in Xlib.

There is a serious flaw with this design, however. Today there are no standards for locale names; each C library vendor must define their own naming conventions for locales. As a result, in a heterogeneous network, such as the one shown in Figure 4, the only way to announce the locale at the display is if the local system administrator can define a set of matching locale names on all machines in the network. The X Consortium has declined to address this problem, since it affects far more than just X. Unfortunately, it appears that at the present time there is no standards organization which sees such heterogeneous network interoperability as part of its charter.

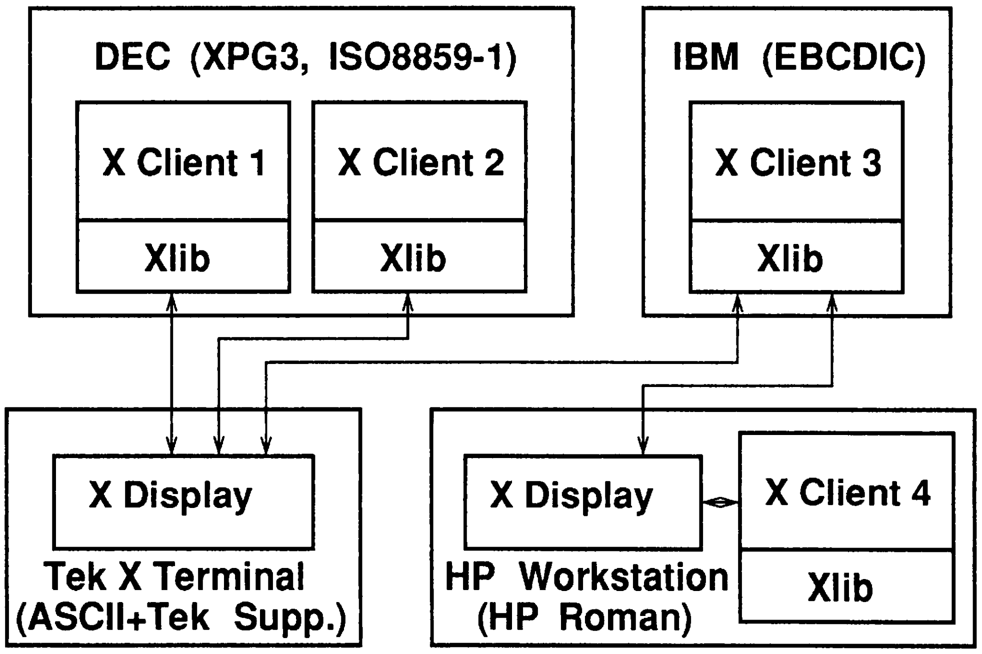


Figure 4 - A Heterogeneous X Network

TEXT COMMUNICATION IN THE X PROTOCOL

Even if the user manages to announce matching locales to clients running on different systems, the implementations of the locale may still not match; in particular, the codesets may be completely different. For text processing within the client, this is no problem, and CompoundText addresses the needs of inter-client text communication across a heterogeneous network. But when the text must be rendered at the display using the fonts provided by the display vendor, problems again arise.

Figure 4 shows an X client running on an IBM mainframe, whose locales are based on EBCDIC instead of ASCII, talking to a Hewlett-Packard display, which provides fonts encoded in proprietary HP charsets. For the client to be able to use these fonts, the Xlib locale definition must be able to map the codeset to the font charsets. However, it is impractical for all Xlib implementations to have knowledge of a variety of proprietary charsets. A solution would be to define a database of mappings from "X standard charsets" (as defined by the Compound Text specification) to proprietary charsets, stored at the display and read by the Xlib implementation. As of this writing, no such protocol has been defined by the Consortium.

CONCLUSIONS

With X11 Release 5, it will become practical to develop internationalized X applications that can meet the local requirements of a large percentage of the world population. The development of the new X11 i18n standard has revealed a number of interesting and important problems which must be solved to meet the processing requirements of the remaining major world languages, to address the needs presented by the increasing amount of inter-cultural communication, to support the growing percentage of the population with multilingual skills, and to realize all of these capabilities in a heterogeneous networked environment.

ACKNOWLEDGEMENTS

The author is indebted to each contributor to the X Consortium "mltalk" discussion group for their knowledge of and insights into internationalization and the X Window System, and for their tireless efforts to achieve consensus on a workable standard for X internationalization.

REFERENCES

Public Review Draft X11 R5 Internationalization Specifications (available from the X Consortium):

Xlib Changes for Internationalization: X Locale Management and Localized Text Drawing, Glenn Widener, Tektronix, Inc.

Input Method Specifications, Vania Joloboff, Open Software Foundation, and Bill McMahon, Hewlett-Packard Company.

X11 R4 Specifications:

X Window System C Library and Protocol Reference, Robert W. Scheifler, James Gettys, Ron Newman, Digital Press, 1988.

X Toolkit Intrinsics - C Language Interface, Joel McCormack, Paul Asente, Ralph Swick, Digital Equipment Corp.

Inter-Client Communication Conventions Manual, David S. H. Rosenthal, Sun Microsystems.

Multilingual Word Processing, Joseph D. Becker, Scientific American, July 1984.

ANSI C X3J11/88-159, ANSI C X3J11 committee (December 7, 1988).

X/Open Portability Guide, Issue 3, December 1988 (XPG3), X/Open Company, Ltd, xpg3@xopen.co.uk, Prentice-Hall, Inc. 1989. ISBN 0-13-685835-8. (See especially Volume 3: XSI Supplementary Definitions)

IEEE Standard Portable Operating System Interface for Computer Environments (POSIX), IEEE Std. 1003.1-1988, New York, New York:IEEE.

Author Index

Keyword Index

Plenary and Panel Speaker Index

UniForum 1991
Plenary and Panel Session Descriptions

The following plenary and panel sessions are organized in the order presented at the UniForum Conference, with the panel sessions grouped under five tracks: Commercial, Futures, MIS, Networking and Portability.

PLENARY SESSIONS

Open Systems — Impact and Analysis

Paul Cubbage (Chair)
Dataquest
San Jose, CA

Brian Grossi
Alpha Partners
Menlo Park, CA

Dr. Michael Pliner
Verity, Inc.
Mountain View, CA

David Tory
Open Software Foundation
Cambridge, MA

Jay Wettlaufer
Visix Software, Inc.
Reston, VA

The panel represents the views of systems manufacturers and applications software vendors, as well as the investment community. The panelists will examine open systems from these aspects: What are open systems? How and why are they evolving? Who benefits and how (users, ISVs and systems vendors)? What is the impact on the software industry? What is the impact on the venture capital community? What are likely future directions? What should be a standard (and what should not)?

Creating a Mass Market for UNIX

Nina Lytton (Chair)
Open Systems Advisor
Boston, MA

Kevin Compton
Businessland, Inc.
San Jose, CA

Alan Hald
MicroAge, Inc.
Tempe, AZ

Steve Malisewski
Compaq Computer Corp.
Houston, TX

Doug Michels
The Santa Cruz Operation
Santa Cruz, CA

Kris Rogers
Merisel, Inc.
Inglewood, CA

This plenary session is targeted at those interested in increasing the success of the UNIX system at the low end. It answers the following questions: How does UNIX stack up against DOS, Windows, OS/2 and PC LANS? What is needed for greater market penetration? What will the benefits be? How and when will these benefits be realized?

Global Computing in the 1990s

William Bonin (Chair)
Hewlett-Packard Co.
Cupertino, CA

T. Michael Nevens
McKinsey & Co.
San Jose, CA

Marc Schulman
UBS Securities, Inc.
New York, NY

George Shaffner
X/Open Co. Ltd.
Reading, Berkshire, U.K.

Geoff Unwin
Hoskyns Group Plc
London, England, U.K.

The age of global computing is here. As businesses expand worldwide operations, information management is an increasingly critical factor in global competitiveness. And the global market is changing rapidly. Political changes in Eastern Europe, the accelerated integration of the European community and the economic emergence of Asia are all factors with which today's IS professional must contend. This panel of experts will explore these issues and provide practical advice to IS professionals who must meet the challenges presented.

UNIX in Commercial Markets

Roger Sippl (Chair)
Informix Software, Inc.
Menlo Park, CA

J. Shirley Henry
Tandem Computers, Inc.
Cupertino, CA

Maggie Konner
International Data Corp.
Framingham, MA

Ray Pena
Bank of America
Concord, CA

Doug Wilson
Kodak Legal Systems
Billerica, MA

Today's powerful hardware platforms, combined with high-performance database engines and application tools, are enabling users to build and run their most critical OLTP applications on UNIX. But commercial users must consider a variety of issues before they switch from proprietary and mainframe-based systems to open systems applications. This session will explore the challenges and benefits of UNIX for commercial computing from a variety of perspectives, including those of an industry analyst, manufacturer, value-added reseller and end user.

Enterprise Solutions

John Ozsvath (Chair)
McDonald's Corp.
Oak Brook, IL

Larry Airaghi
Proctor & Gamble Co.
Cincinnati, OH

Don Boron
Timken Co.
Canton, OH

Cheryl Currid
Coca-Cola Foods
Houston, TX

Jean-Pierre Dejean
Nielsen Advanced Information
Technology Center
Northbrook, IL

User executives will discuss their companies' enterprise-wide needs and priorities for open systems solutions. Each will address the following issues: What is your vision of open systems and what are the critical dimensions of an "enterprise solution"? Where and how do UNIX and derivative operating systems fit in? Where is migration realistic? Where is coexistence mandated? What are your most urgent priorities for action by the computer industry? The session is designed for users who wonder if UNIX and open systems can meet the needs of a global enterprise.

Future Trends

David Card (Chair)
International Data Corp.
Framingham, MA

Bill Rash
Intel Corp.
Santa Clara, CA

Donna Van Fleet
IBM Corp.
Austin, TX

Cheryl Vedoe
Sun Microsystems, Inc.
Mountain View, CA

Peter Weinberger
AT&T UNIX System Laboratories, Inc.
Summit, NJ

Where is UNIX technology headed? Which standards will be important? Where is academia headed? What does the evolution of microprocessors, networks, image processing and multimedia mean to users, developers, vendors and DP/MIS managers? This session should be of benefit to both users and those involved in developing products for open systems.

COMMERCIAL

Open Systems Challenges in the OLTP Market

Steve Levich (Chair)
Sequent Computer Systems
Beaverton, OR

Carl Chilley
X/Open Co. Ltd.
Reading, Berkshire, U.K.

Keith Hospers
Independent Technologies, Inc.
Fremont, CA

Tony Story
IBM Corp.
Austin, TX

How do open systems compare to proprietary systems in terms of available high-performance OLTP capabilities? This panel will explore the required functionality for open systems OLTP, what's available now and what's coming in the near future. Views of end users, ISVs and vendors will be represented, along with an update from X/Open on the progress of transaction processing standards.

UNIX System Security

Craig Rubin (Chair)
AT&T Bell Laboratories
Summit, NJ

Kevin Brady
AT&T UNIX System Laboratories, Inc.
Summit, NJ

Trisha Jordan
Sun Microsystems, Inc.
Mountain View, CA

Mike Ressler
Bellcore
Piscataway, NJ

Mark Schaffer
Secure Computing Technology Corp.
Arden Hills, MN

This panel will address the motivation for additional UNIX system security, discuss implications security places upon the end user and describe work in progress by both standards committees and R&D organizations. In addition, topics such as network security and other security features required by end users will be addressed.

UNIX System Management in a Commercial Environment

Brian Gibson (Chair)
Sequent Computer Systems
Beaverton, OR

Martin Kirk
X/Open Co. Ltd.
Reading, Berkshire, U.K.

Larry Kluger
Sun Microsystems, Inc.
Mountain View, CA

Bob Lyon
Legato Systems, Inc.
Palo Alto, CA

E. Scott Menter
Lehman Brothers
New York, NY

UNIX system management tools, including software installation and distribution, on-line backup and restore, batch job control and user account management, must all work in a heterogeneous environment and within a consistent management framework. What technology meets these requirements today? Should standards be set now or would standardization prevent desired growth and development?

High Availability and Fault Tolerance: How Much Protection is Enough?

Jeff Erramouspe (Chair)
NCR Corporation
West Columbia, SC

Al Dei Maggi
Sequent Computer Systems
Beaverton, OR

J. Shirley Henry
Tandem Computers, Inc.
Cupertino, CA

Raanan Peleg
Hewlett-Packard Co.
Cupertino, CA

The introduction of fault-tolerant and highly available UNIX systems in the past years has increased their credibility for mission-critical commercial applications. This panel will discuss the issues surrounding the use of these systems, focusing on the cost of fault tolerance and high availability, and its benefits to the commercial enterprise.

Building and Realizing Value for Your Company

Paul Deninger (Chair)
Broadview Associates
Fort Lee, NJ

Steve Clearman
Geocapital Partners
Fort Lee, NJ

Ron Fisher
Phoenix Technology
Norwood, MA

Ronald Lachman
Interactive Systems, Inc.
Naperville, IL

Martin Waters
Locus Computing Corporation
Inglewood, CA

The goal of this panel will be to present entrepreneurs with timely, real-world insights into the business considerations of competing in the open systems/UNIX marketplace. The question of how best to build the future value of an entrepreneur's shareholdings will be addressed by CEOs of UNIX software or service companies and a leading venture capitalist.

UNIX in the Microcomputer Sales Channel

Jerry Trimm (Chair)
TBS Services, Inc.
Santa Clara, CA

Joseph Biniskiewicz
Ingram Micro D
Santa Ana, CA

Curt Fisher
Sun Microsystems, Inc.
Mountain View, CA

Andy Green
The Santa Cruz Operation
Santa Cruz, CA

Pete Rourke
MicroAge
Tempe, AZ

This panel will discuss the changes that must be made in order to be successful selling UNIX in today's market. Several viewpoints will be presented that emphasize the changes in thinking and infrastructure required by the retail computer resellers when moving to UNIX-based solutions.

FUTURES

Future Evolution of X Window Displays

Greg Blatnik (Chair)
Dataquest
San Jose, CA

Judy Estrin
Network Computing Devices
Mountain View, CA

Peter Shaw
Advanced Graphics Engineering
San Diego, CA

Lynn Thorsen
Evans & Sutherland Computer Co.
Salt Lake City, UT

Rusty Williams
IBM Corp.
Austin, TX

This panel will examine future X Window Display enhancements and evolution, including X Window terminals, PCs and workstations. Topics will include video extensions to the X Window System (vex), 3D (PEX), blending, internationalization, font handling and the use of alternate input devices.

GUI Development Productivity Tools

Ed Lee (Chair)
Hewlett-Packard Co.
Corvallis, OR

Ross Faneuf
Digital Equipment Corp.
Nashua, NH

Bob Watson
Sun Microsystems, Inc.
Mountain View, CA

Ted Wilson
Hewlett-Packard Co.
Corvallis, OR

The recent success of several graphical user environments for UNIX (the X Window System, OSF/Motif and Open Look) would appear to have solved end users' major objection to the system: ease of use. However, applications that utilize these technologies have been slow to appear. This panel will discuss various approaches to accelerating the development of applications with GUIs for UNIX.

Multimedia and UNIX

David Marshak (Chair)
Patricia Seybold's Office Computing Group
Boston, MA

Gilbert Wai
Informix Software
Menlo Park, CA

Karl Wolf
Sun Microsystems, Inc.
Mountain View, CA

This panel will focus on the opportunities for multimedia applications in the UNIX environment, particularly on the potential business uses of audio and video technologies. Presentations will include demonstrations of current multimedia applications and a sneak preview of what the next generation of applications will look like.

Commercial Requirements for Imaging

Georgia McCabe (Chair)
Eastman Kodak Co.
Rochester, NY

John Carpovich
U.S. Navy Publishing & Printing
Washington, D.C.

Cynthia Dai
Sun Microsystems, Inc.
Mountain View, CA

Jef Graham
Hewlett-Packard Co.
Wokingham, Berkshire, U.K.

Historically, imaging-based products have emerged from a number of distinct high-end markets: high-volume document management, medical diagnosis, engineering, CAD/CAM and publishing/printing. Today, with the emergence of open, low-cost, image-capable systems, these distinctions tend to blur. Given this state of affairs, just what is the imaging market?

File Server Architecture for the '90s

Rick Bohdanowicz (Co-Chair)
Novell, Inc.
Sunnyvale, CA

Bruce Nelson (Co-Chair)
Auspex, Inc.
Santa Clara, CA

Doug Kaewert
Sun Microsystems, Inc.
Mountain View, CA

Dr. Philip Lehman
Transarc Corp.
Pittsburgh, PA

Gary Stearns
Hewlett-Packard Co.
Fort Collins, CO

This panel presents and discusses four different approaches to file server architecture from the perspective of the leading network computing technologies: NetWare, NFS, AFS and LanManager. The common thread is how each approach incorporates high-performance file-level UNIX interoperability.

Document Image Processing — An Emerging UNIX Market

Mike Florio (Chair)
Document Technologies, Inc.
Mountain View, CA

Cynthia Dai
Sun Microsystems, Inc.
Mountain View, CA

Steve Davis
Digital Equipment Corp.
Marlboro, MA

Dr. Angela Hey
Areva International
Belmont, CA

Scott McCready
International Data Corp.
Framingham, MA

In many respects, the document image processing segment parallels the development of the UNIX market, with important decisions regarding standards, platforms and performance issues being discussed and proposed. X Windows may also play an important role in this emerging market.

MIS

The Politics of Implementing Open Systems

Judith Hurwitz (Chair)
Patricia Seybold's Office Computing Group
Boston, MA

Helene Csvany
Roadway Express
Akron, OH

Pamela Gray
Marosi Ltd.
Ascot, Berks, U.K.

David Sherr
Shearson Lehman Hutton
New York, NY

This session is geared to data processing managers who are considering implementing open systems or UNIX in a traditionally proprietary environment. Issues ranging from convincing top management of the safety of open systems and UNIX to finding ways to integrate these systems with proprietary systems will be addressed. See how others have successfully implemented open systems in their organizations.

Distributed Databases

Sherri Osaka (Chair)
Informix Software, Inc.
Menlo Park, CA

Dan Bailey
Rust International Corp.
Birmingham, AL

Berl Hartman
Sybase, Inc.
Alameda, CA

Eric Wasiolek
Ingres Corp.
Emeryville, CA

Distributed data base applications are one of the key technologies for the 1990s to manage corporate data efficiently. This panel will explore issues critical to the success of distributed database applications, including implementing distributed databases in heterogeneous environments, the need for distributed transaction monitors, security considerations and requirements, and database usability and administration.

OS/2 vs. UNIX...Which Will I Choose?

Carol Realini (Chair)
Legato Systems, Inc.
Palo Alto, CA

Dan Lynch
Interop
Mountain View, CA

John McCarthy
Forrester Research
Cambridge, MA

Thomas Wheeler
American Express
Phoenix, AZ

What issues do MIS managers face when making the decision between OS/2 and UNIX, or deciding to use both? Each panel member will review the strengths and weaknesses of both platforms. Is there an obvious winner? What are the key success factors in different application and business environments? Each panel member will take a firm stand.

UNIX as an MS-DOS Server

John Harker (Chair)
The Santa Cruz Operation
Santa Cruz, CA

Laura Howard
Sun Microsystems, Inc.
Billerica, MA

Ron Simon
Microsoft Corp.
Redmond, WA

Peter Uhlir
Locus Computing Corp.
Inglewood, CA

The increasing power of UNIX on microcomputer platforms has caused numerous network vendors to create UNIX-based MS-DOS network solutions. How have vendors taken advantage of the UNIX platform in creating their solutions? Did they write their own file systems? Do users know it's UNIX? Do they care? And what are the benefits from the portable nature of UNIX application across platforms?

UNIX System Capacity Planning: Where's the Data?

Dr. Sivaram Chelluri (Chair)
AT&T
Lisle, IL

David Chadwick
Performance Awareness Corp.
Naperville, IL

Terry Flynn
Amdahl Corp.
Sunnyvale, CA

Tony Gaseor
AT&T Bell Labs
Naperville, IL

Dave Glover
Hewlett-Packard Co.
Roseville, CA

Currently no UNIX system vendor provides enough performance management functionality and certainly no two vendors provide equivalent functionality. Panel members will discuss the implementation of performance data gathering facilities in several different versions of the UNIX operating system.

International Commercial Usage of UNIX

Roger Hicks (Chair)
Open Systems Consultant
Auckland, New Zealand

Noboru Akima
Information Technology
Promotion Agency
Tokyo, Japan

Kim Biel-Nielsen
Uniware danmark a/s
Vedbaek, Denmark

Greg Rose
Software Pty. Ltd.
Chippendale, New South Wales,
Australia

In larger, high-profile marketplaces, UNIX has been seen strictly as a technical operating system. However, in many other countries computing has evolved in a different environment and UNIX has become widely used for commercial applications. This panel will look at this commercial UNIX usage in international marketplaces.

NETWORKING

UNIX Multiuser Systems and PC LAN Integration: A Flexible Network Solution

Ron Conway (Chair)
Altos Computer Systems
San Jose, CA

Bob Davis
Novell, Inc.
Sunnyvale, CA

Laura Howard
Sun Microsystems, Inc.
Billerica, MA

Dan Ladermann
The Wollongong Group
Palo Alto, CA

Michael Smith
3Com Corp.
Santa Clara, CA

This panel will look at the business benefits of integrating UNIX multiuser systems and PC LANs. Leading UNIX and DOS networking experts will discuss how UNIX/DOS network integration leads to a flexible network computing environment. Additional speakers will discuss the implementation of UNIX/DOS network solutions in their environments.

Client/Server Networking Alternatives

John Chisholm (Chair)
John Chisholm Co.
Menlo Park, CA

Dick Bush
Auspex, Inc.
Santa Clara, CA

Jeff Hudson
Netframe, Inc.
Milpitas, CA

Dave Langlais
The Wollongong Group
Palo Alto, CA

Heinz Lycklama
Interactive Systems Corporation
Santa Monica, CA

One of the more significant impacts of the client/server computing architecture is the emergence of new niche markets for computing elements. This includes X Windows terminals, file servers, database servers and image servers. This panel will discuss the implementation of Network File Systems, SQL servers, Remote Procedure Calls, X Window support and other alternatives.

Experiences in Managing TCP/IP Networks with SNMP-Based Tools

Joe Bonner (Chair)
Hewlett-Packard Co.
Fort Collins, CO

Ed Alcoff
The Wollongong Group
Palo Alto, CA

Randy Fardal
Retix
Santa Monica, CA

Bill Lanfri
Synoptics Communications, Inc.
Mountain View, CA

This panel, consisting of network managers for large multivendor networks that have been using SNMP-based tools, will share their overall objective in managing multivendor networks, how SNMP tools have assisted in achieving their goals, and what advantages and shortcomings exist with SNMP tools.

The UNIX System — The Missing Link: What We Have in Common is Heterogeneous Networks & UNIX

Ben Salama (Chair)
Interactive Systems Corp.
Naperville, IL

John Krakauer
HealthCare Compare Corp.
Downers Grove, IL

William Wellman
Rush-Presbyterian-St. Luke's
Medical Center
Chicago, IL

UNIX provides the key part of the solution when faced with the challenge of interconnectivity. See how UNIX is the glue for networks of dissimilar computing equipment and why UNIX is the common platform for critical applications that enable companies to communicate with existing proprietary systems.

UNIX System Management in a Distributed Computing Environment

Geraldine M. Vitovitch (Chair)
AT&T UNIX System Laboratories, Inc.
Summit, NJ

Carl Cirillo
AT&T
Lincroft, NJ

Keith Ensroth
K-Mart Corporation
Troy, MI

Robert Fabbio
Tivoli Systems Inc.
Austin, TX

Susan Knapp
Sun Microsystems, Inc.
Billerica, MA

Dale Shipley
Veritas Software Co.
Santa Clara, CA

Panelists will cover a broad range of distributed UNIX system computing environment topics, including configuration requirements, operating costs, data integrity, system and network reliability, administrator/user training and user interface needs, and distributed application support.

TCP/IP to OSI: Handling the Transition

John Harker (Chair)
The Santa Cruz Operation
Santa Cruz, CA

Doug Ambort
The Wollongong Group
Palo Alto, CA

Bob Cooney
NARDAC
Washington, D.C.

Lloyd Spencer
Sun Microsystems, Inc.
Mountain View, CA

Sayuri Tung
Retix
Santa Monica, CA

All U.S. government agencies and departments are required to conform to the Government Open Systems Interconnection Protocol (GOSIP). What is still needed are OSI network packages with simplified end-user administration, good third-party application support, and the interoperability and price performance value that TCP/IP network alternatives currently offer.

PORTABILITY

Open Systems: Interoperability Solutions

Heinz Lycklama (Chair)
Interactive Systems Corporation
Santa Monica, CA

Peter Cunningham
UNIX International
Parsippany, NJ

Walter De Backer
Commission of the
European Communities
Luxembourg, Luxembourg

Ira Goldstein
Open Software Foundation
Cambridge, MA

Mike Lambert
X/Open Co. Ltd.
Reading, Berkshire, U.K.

The 1990s are being called the decade of the open systems environment. Interoperability is a key issue. In this panel session, two vendors currently providing interoperability solutions will describe them and how they plan to coexist. Two end users will describe how they use or plan to use these interoperability technologies to provide distributed applications.

An Insider's View of Open Systems Standards

Carl Chilley (Chair)
X/Open Co., Ltd.
Reading, Berkshire, U.K.

Rikki Kirzner
Dataquest
San Jose, CA

Doug Michels
The Santa Cruz Operation
Santa Cruz, CA

Jeff O'Neil
Arco Oil and Gas Co.
Plano, TX

The open systems movement has spawned a seemingly endless list of standards-setting bodies, including NIST, OSI, POSIX, COS, MAP/TOP, ANSI and X/Open. Do we really need all of these groups? Given the number of groups, can't they work a little faster? This panel will provide an overview of current projects of the various standards-setting bodies and tackle the issue of whether all these activities are useful.

Transparent Interoperability for Software

Wayne Sennett (Chair)
Motorola, Inc.
Universal City, CA

John Alleborn
Merrill Lynch & Co., Inc.
New York, NY

Stanley Cohen
The Atrium Report
Cambridge, MA

Thomas Mace
88open Consortium, Ltd.
San Jose, CA

Cheryl Vedoe
Sun Microsystems, Inc.
Mountain View, CA

Users today talk about networking their entire enterprise transparently, regardless of systems architecture or operating system. How close are we to achieving total transparent interoperability? This panel will explore various solutions, the feasibility of each and how transparent interoperability will affect the industry.

Application Portability — Is it Real or is it the Holy Grail?

Bruce Weiner (Chair)
Mindcraft, Inc.
Palo Alto, CA

Dr. Lin Brown
Sun Microsystems, Inc.
Mountain View, CA

Dominic Dunlop
Migration Software System
Cholsey, Wallingford, U.K.

Alex Morrow
Lotus Development Corp.
Cambridge, MA

Open systems promises application portability. This panel will discuss how well open systems can deliver on that promise and what programmers must do themselves in order to make applications portable. Topics covered include operating systems, programming languages, graphical user interfaces and tools to help verify application portability.

Application Environments: Are Six Better Than One?

Wendy Rauch (Chair)
Emerging Technologies Group, Inc.
Dix Hills, NY

Jim Isaak
IEEE Technical Committee (POSIX)
Digital Equipment Corporation
Nashua, NH

John Williams
General Motors
Troy, MI

Percy Young
Burlington Coat Factory
Lebanon, NH

This panel will discuss many of the similarities/differences between application environments and whether the differences are significant. It will also discuss the method used to combine standards for particular application areas, as well as user techniques for planning and evaluating open systems architecture and handling existing nonstandard software.

Shrink-Wrapped Software: Is ANDF the Answer?

Peter Griffiths (Chair)
The Instruction Set, Ltd.
London, England, U.K.

James de Raeve
X/Open Co. Ltd.
Reading, Berkshire, U.K.

Pat Riemitis
Open Software Foundation
Cambridge, MA

Michael Tilson
The Santa Cruz Operation Canada
Toronto, Ontario, Canada

Simon Walden
Uniplex
Hemel Hampstead, Hertfordshire, U.K.

One solution to the problem of UNIX system binary compatibility is the architecture-neutral distribution format (ANDF). ANDF provides for distribution of semi-compiled software which is finally compiled on a target machine at runtime. Will ANDF be the solution to the problem of UNIX software distribution?

UniForum

The International Association of UNIX Systems Users

2901 Tasman Drive • Suite 201
Santa Clara, California 95054