Proceedings of the 5th European BSD Conference

November 10-12, 2006
Novotel Milano Linate, Milan, Italy

EuroBSDCon 2006

Proceedings of the fifth European BSD Conference

November 10-12, 2006

Novotel Milano Linate, Milan, Italy

Event organization:

WillyStudios.com
I-20040, Carnate (Milano)
Via Carducci 9

Published by WillyStudios.com di Stucchi Massimiliano Andrea
Via Carducci, 9
I-20040 Carnate (Milano)

Tel. +39 02 44417203
Fax. +39 02 44417204
email. Info@willystudios.com

http://www.willystudios.com

The Program Committee is composed by:

Giuseppe Maxia: Program Chair

Matteo Riondato: FreeBSD

Jan Schaumann: NetBSD

Jeffrey Hsu: DragonFlyBSD

Greg Lehey: Expert at large

Guido Sintoni: Freelance Journalist

Massimiliano Stucchi: Conference Organizer

Add IPv6 support for IPFW2 and DUMMYNET for FreeBSD

Raffaele De Lorenzo, Luigi Rizzo, Mariano Tortoriello

October 14, 2006

Abstract

We illustrate how to build a firewall and a traffic shaper for the FreeBSD system (including Releng 4.x, 5.x, 6.x) with enhanced IPv6 filter capabilities compared to standard IPv4 capabilities. IPv6 will become soon the new standard Internet Protocol, and it differs radically from IPv4 Internet Protocol. New security policies are needed for all systems that currently use (or will use) the IPv6 protocol. As far as compatibility is concerned, the new protocol can coexist with the old one, since they can work independently. Therefore, it will be possible to move gradually from IPv4 to IPv6. The goal of this paper and related codes is the implementation of the IPv6 protocol inside existing firewall/-traffic shaping programs (IPFW2/DUMMYNET) supporting only IPv4. In this way, compatibility is preserved. In the first section we describe in detail the IPFW2 Firewall and DUMMYNET Traffic Shaper, including functionality and rule structure. We also describe the technical implementation and the hook with the IPv4 FreeBSD Kernel stack. In the second section we describe the Internet Protocol Version 6 (IPv6), the main differences with respect to IPv4, and how IPv6 is included in the FreeBSD kernel (IPv6 stack). In the third section we describe our implementation aimed at making IPFW2 and DUMMYNET working with IPv6 rules. We describe in detail the hooking with the FreeBSD IPv6 stack, crucial for a correct implementation. Tests are described in the last section. On April 18th 2005 this code was committed in FreeBSD CURRENT by Brooks Davis (via Luigi Rizzo). See *http://www.freebsd.org/news/status/report-jan-2005-mar-2005.html* for more info.

1 Ipw2 and Dummynet

Starting from version 4.x:ipfw2, FreeBSD has had a very reliable Firewall. The Ipfw2 is an advanced firewall, very powerful and versatile. You can use the ipfw with a dummynet traffic shaper for a nearly absolute security of your systems. However, the security is not absolute because there is no IPv6 support for ipfw and dummynet.

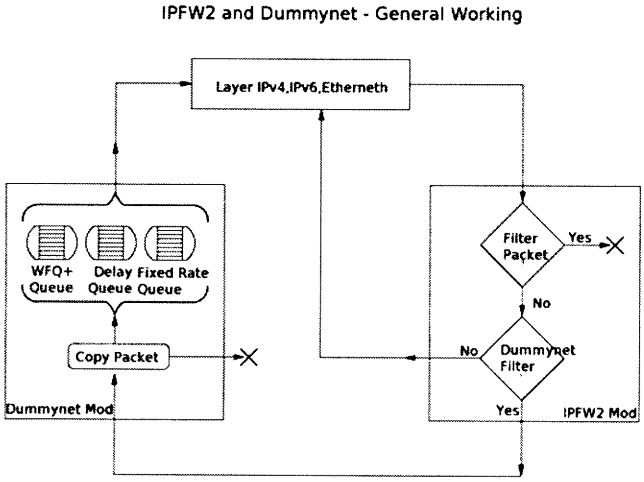


Figure 1: Package Management with Dummynet and Ipfw2

In FreeBSD 4.x the IPv6 entrusts the IPv6 traffic to ip6fw, a firewall based on the old 'ipfw'. The ipfw does not have the advanced features of ipfw2. For example, the ipfw works on static/dynamic rules rather than on traffic shaping. Ipfw2 and Dummynet sources are very nice 'c' codes, more flexible for changes and adds on. We will discuss the following steps that have led us to the realization of the new firewall and traffic shaper:

- Adapting IPFW2 Firewall module
- Adapting DUMMYNET Traffic shaping module
- Adapting FreeBSD's IPv6 stack

The most relevant add-on in Ipfw2 was the introduction of a new pool of rules for IPv6, based on those

already present in IPv4, while for Dummynet we introduced a hook to the IPv6 stacks and some adapting of the structures. In the user module we introduced a new parsing method for the new IPv6 rules. These rules are syntax-based like in the IPv4, and preserve the compatibility with all the existent releases. Finally, we introduced Ipfw2 and Dummynet hooks in the FreeBSD IPv6 stack, in a similar way used for the FreeBSD IPv4 stack. We will first describe the operative function of Ipfw2, Dummynet, and FreeBSD IPv6 stack (developed by the Kame project). Then, we will describe in detail all the add-ons we made and the results of the testing.

2 IPFW2, The Firewall

In the last few years, the Internet has expanded exponentially, and the world networking traffic is continuously increasing. Methods for checking the internet traffic (Firewalls) and for controlling communications flows (Traffic Shapers) are needed. A Firewall is a packet analyzer set, which operates between the packet operative management (like fragmentation, CRC check, ...) and the true packet management. Flow control is practically all dependent on the Firewall policies.

Ipfw2 can be split into two parts: the first half is operative and the second half is for control. The control part dials up with the user (root) through a user interface. The user can also interact with Ipfw2 trough sysctl variables (see manual pages) that control the entire firewall behavior. The operative part is the core of Ipfw2 and it is the routine called for checking-in the rules. This operation is a sequential scan of all the rules in a descent mode, from the first until the last rule which says "default rule". The first rule that matches with the arrived packet is applied. The sequence number of the rules is automatically increased, or you can insert a preferred sequence number when you define the rule. The "default rule" has a sequence number 65535 and it is special, because it cannot be erased/modified! This is policy-compiled and it can be IPFIREWALL_DEFAULT_TO_ACCEPT understanding like to accept all packets from all or IPFIREWALL_DEFAULT_TO_DENY instead. How is Ipfw2 invoked, and who calls it? The firewall is called in some points of the IP/ETHERNET stack (see Fig. 2), and Ipfw2 can be called more than once. You can see from this picture that the Ipfw2 works fine in input and in output and can protect from external attacks. However it can also limit the traffic requests to exterior, and is useless in the flow control of the inside system.

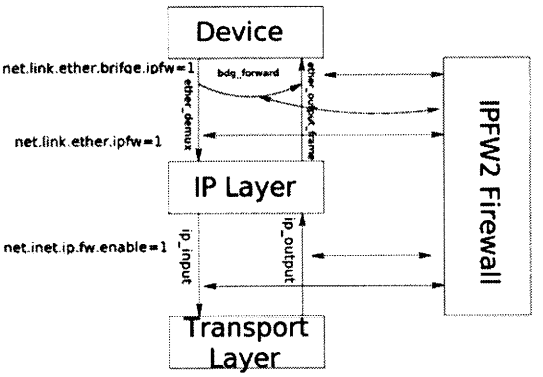


Figure 2: How ipfw2 and Dummynet are invoked

2.1 Structure of rules

Rules are organized in a list called *struct ip_fw*. The relevant part of this structure is the body of rules, which is structured as a set of micro-instruction blocks all related to some IP/TCP packet parts and their action policy. The micro-instruction core is the structure *ipfw_insn* that contains the micro-code and identifies the instruction type, and it has the dimension value of the microcode. This structure is very small and not adapt for complex parameters, but is scalable. If you need some complex parameters, you must include them in the base structure. Their dimension will be added to the dimension parameter value. Some rules were created in this way, in order to have some special type for fast and comprehensive programmer code. Structure of the rule is:

1. action
2. policy filter

Both parts have one or more structures *ipfw_insn* followed by other parameters related to the operation encoded by the microinstruction. The supported opcodes are listed in *enum ipfw_opcodes* and have built-in the same classes discussed above. The filter rule has one or more micro-instructions that are analyzed by the firewall in some blocks for packet-matching. Microistructions can discriminate all parts of a packet, like the IP header (for example source address and source destination), the IP payload, of the high protocol header (TCP ports,UDP ports...). Some frequently used commands have a dedicated type in order to simplify the programmer code, for example *ipfw_insn_sa* is

a structure dedicated to the IPv4 socket while the structure *ipfw_insn_ip* is used for the IPv4 address and the relative netmask. You can see how these structures follow the Ştructure of rulebecause they are built with a structure *ipfw_insn* and the type *struct sockaddr_in* (for the first example) and *struct in_addr* (for the second one). The äctionöf rule describes the real firewall action when the analyzed packet matches with the filtered commands. Typical actions are "DENY","ACCEPT","DIVERT","FORWARDING". The actions that divert the packet to DUMMYNET are (*O_PIPE* and *O_QUEUE*). These microinstructions have the same structure of filtered microinstructions but differ from them slightly because the action is easier and needs only the structure *ipfw_insn*. The structure *ipfw_insn_cmd* in the rule is the first filter command and eventually other commands are allocated contiguous. Their dimension is defined by parameters created by the module. The same method is used for äctions, but the first action offset is a free parameter inside a basic structure for fast use. The strength-point of Ipfw2 is the easier expansion of rules and the easier creation of new rules (for the last operation you must insert a new *opcode* and a new dedicated structure).

2.1.1 Ipfw2 Control Part

The control part is an interface that links a communication socket with the kernel module (pointer *ip_fw_ctl_ptr* that references the function *ipfw_ctl*) and organizes the rules. With this instrument you can obtain:

1. List of free rules
2. Add/Remove rules
3. Groupping rule set
4. Reset or view rule's counter

When you insert a new rule, the control part makes a simple check that consists of measuring the microsinstruction dimension.

2.1.2 Ipfw2 Operative part

The operative part is called by the *ipfw_chk* function through *ip_fw_chk_ptr* and it takes filtered operations. The hooks that call Ipfw2 are posted at the points

1. Ethernet stack (*if_ethersubr.c* and *bridge.c*)
2. IPv4 stack (*ip_input.c* and *ip_output.c*)

The Operative part is invoked by some parameters that are stored in a structure called *args*. This structure contains the packet that will be processed and some parameters like a pointer to the last rules if the Ipfw2 was called before. This is important if the *sysctl* variable *net.inet.ipfw.one_pass* is set, because it makes faster the checking rule process. The operative part, when is invoked, collects some information from the packet that will be used for filter operations. Next, it checks and matches all microinstructions that build the rules. If these operations are true, actions will be runned.

3 Dummynet: the traffic shaper

Dummynet is the module that allows to mould the IP traffic that runs thought the net interfaces. Using the ipfw control part you can configure how to model the traffic throught available policies. It works in a very simple way: every traffic rule can be seen as a tap that can be opened or closed by the same rule. The water flow is the matched bandwidth. Dummynet allows for different kinds of data flow control. For every queue and for every parameter, Dummynet can decide the modality of the traffic. The queues can be of 3 different kinds

1. Fixed rate Queue
2. Delay Queue
3. WF2Q+ Queue

The fixed rate queue is used for setting up the bandwidth permanent to a single rate. The delay queue is used for slowing down the speed of packets. These two kinds are also called *pipe*. A queue WFQ2+ (Worst-case Fair Weighted Fair Queueing) policy is an efficient variant of the WFQ policy. The queue associates a weight and a reference pipe to each flow, and then to all the backlogged (i.e., with packets queued) flow, and finally to all backlogged, proportionally to their weights. Note that weights are not priorities; a flow with a lower weight is still guaranteed; a flow with a lower weight is still guaranteed to higher weight if is permanently backlogged. In practice, pipes can be used to set hard limits to the bandwidth that a flow can use, while queues can be used to determine how different flows share the available bandwidth.

If we add some Dummnynet rules, the relative packet flow is shown in Fig. 4

Similarly to Ipfw2, we can divide the structure of Dummynet in three parts

- Control part
- Operative part
- Interface

3.1 Dummynet Control Part

Like for Ipfw2, the control part is used to create and configure *pipe/queue* throught the function *ip_dn_ctl*.

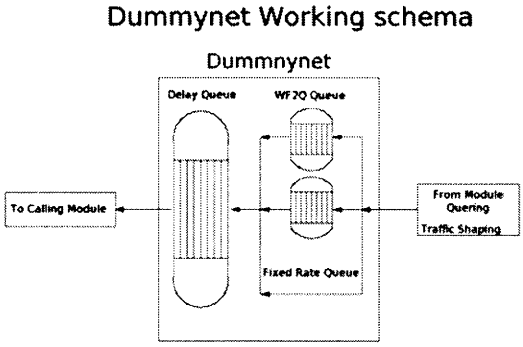


Figure 3: How Dummynet work

More common configuration for Dummynet are

1. **mask**: Used for data flow control throught identification mask that is composed of parameters from TCP/IP/UDP/ICMP protocols.
2. **plr**: Is the packet loss ratio, a probability parameter for packet loss
3. **red/gred**: Are algorithms for traffic flow.

The first flow type (pipe) corresponds to some parameters

1. **bw**: Is the *bandwidth* assigned to the flow
2. **delay**: Is the delay for slowing the packets flow.

To configure a *queue*, you can use these parameters

1. **pipe**: A pipe to redirect the flow to some filters
2. **weight**: A weight used for the *fair queuing* algorithm.

All configurations parameters were stored in the structure *dn_pipe*.

3.2 Dummynet operative part

The committed work by Dummynet operative part is a regular run slice determined by the kernel variable *HZ*. Therefore this variable determines the Dummynet queues wake-up. The operative part runs actions to packets in queues; in fact inside all the queue structures there are some temporal policies. The routine *dummynet* sends get-ready packets (stored in *dn_pkt*) to the interface (stored in flag *dn_dir*). These packets are back grabbed by *dummynet_io*, according to an algorithm for traffic flow (**red/gred**). Next the packet (tagged by DUMMYNET first) is passed to the IP/Ethernet layer and is re-inserted to the Ipfw2 and, according to the *sysctl* variable *one_pass*, is re-filtered.

4 Internet Protocol Version 6

IPv6 (see rfc2460) will replace the actual IPv4 protocol for the new network needs (like multimedia flow traffic, and simply the end of IPv4-addresses). IPv6 use 128 bit for addresses space and *unicast*, *multicast* and *anycast* classes. These address classes are used for services coming out of some hosts. IPSEC has a native support in IPv6. The IPv6 header is illustrated in Fig. 7 (see rfc2460). The IPv4 header is illustrated in Fig. 6. Notice that the IPv6 header is simpler (therefore faster) than IPv4.

IP Version 4 - Header scheme

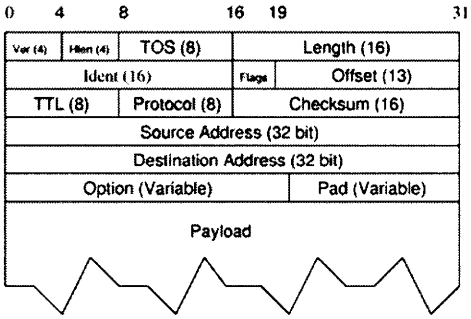


Figure 4: IPv4 Header

These are the main differences between IPv6 and IPv4.

Traffic Class: Used by router for identifying the same traffic class (priority packets). *Flow Label:* Data flow (like the same for ATM Protocol), used for real-time.

Notice that in the IPv6 header the *checksum* field was removed for service reasons. The *header lenght* field was removed because the dimensions of header are fixed to 40 byte. The payload length is 16 bit fixed but this is too small for higher capacity LAN; this is why the *jumbogram* field was used. The *options* field was removed because IPv6 uses a list of extension-headers for options. The extension-headers are six and are inserted next to the IPv6 header. The routers forwarding are very fast because they do not analyze it (payload, see *rfc2460*). The order for the IPv6 extension-header is

- header IPv6;

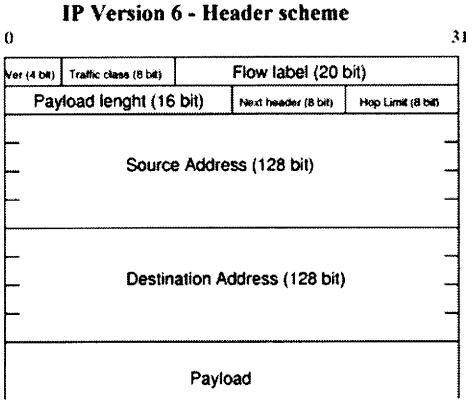


Figure 5: IPv6 Header

- hop by hop option header
- destination option header
- routing header
- fragmentation header
- authentication header
- encrypted security payload header
- destination option header
- upper layer header (es. TCP o UDP).

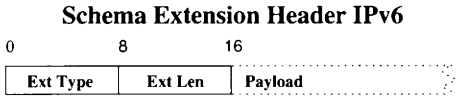


Figure 6: IPv6 extension header

The fields for fragmentation are removed, because this was made by extensions-header. In IPv6, the fragmentation is no more made in routers, and is very fast. The true MTU for transmission is calculated by a *MTU Discovery* procedure. In IPv6, the MTU has a minimum value of 1280 bytes. For compatibility, IPv6 uses an encapsulated *tunneling* procedure for IPv4 packets.

5 IPv6 stack in FreeBSD

FreeBSD IPv6 are made by the Kame group. The I/O part are controlled by *ip6_input* and *ip6_output* functions.

6 Userspace interface

There is a unique interface for both Dummynet and IPFW2, and the main command is *ipfw*. You can see more details in the man page [2]. The main goals of the ipfw command line are

- *rule management*: modify all rules in the ipfw2 ruleset
- *rule statistics*: perform visualization of rule status and statistics for the administrator
- *dummynet management*: add dummynet specific rule in the ruleset

The command interface is designed to create directly the structure needed by ipfw2 and to send it to the ipfw daemon. The syntax used is backward-compatible with the ipfw1 ruleset, at least for the basic rules. Due to the increase of capabilities, some command were added to the interface.

The command line interpreter takes each option written by the user and creates the related micro-op. The rule is created simply by acquiring the commands in writing order. This a very simple way to perform packet filtering and it permits to develop a very complex ruleset that can be interpreted at the start time of the deamon. The ruleset can be considered as a network language that ensures the security of the system. Once the interface has completed the creation of the rule, it opens a socket with the control part of ipfw2 and it performs the operation by sending an appropriate structure of command and rule.

The structure of a command can be summarized as follows

ipfw ***main_command*** *[rule_body]*

the main command encodes the behavior of the interface, some common main commands are

- *add/delete*: ipfw2 rule management
- *list* or *show*: performs visualization of the firewall statistics and ruleset
- *pipe/queue*: dummynet management

the rule body depends on the main command but it reflects the ipfw2 rule structure described above, and can be divided in

action protocol ***from*** *source_address* ***to*** *destination_address [option]*

in the following example we could identify a first part in which we encode the action and a second part in which we store the match criteria for the IPv4 packet

ipfw add deny ip from 192.168.0.1 to 192.168.0.20

some other actions could be, for instance,

- allow/permit/accept/pass
- deny/drop

The introduction of the IPv6 caused some changes in the address interpreter, in the protocol interpreter, and in the statistic output. However, the global structure of the command line was kept. The structure of the rules was maintained in both IPv4 and IPv6 environments so the flexibility and power of ipfw2 ruleset is available for both technologies.

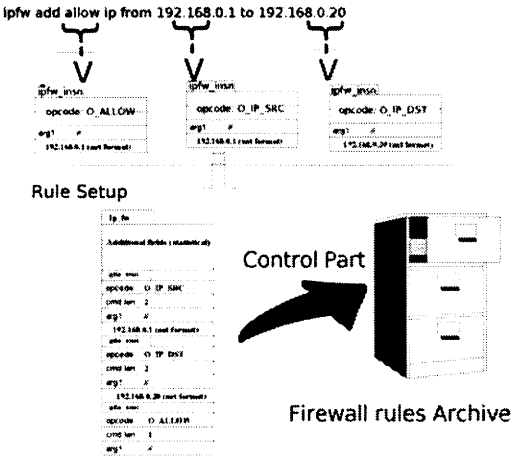


Figure 7: Translation of IPv4 rule

In addition to this simple syntax there are a lot of commands and option by which is possible to modify the behavior of the firewall during the ruleset analysis. For example it is possible to modify the sequence of the scanning, deactivate some rules, support additional statistics, create sono rule on the upper layer protocol i.e. TCP/UDP port. Some of these commands could be the following

- **ipfw enable debug**
- **ipfw show**
- **ipfw pipe show**
- **ipfw flush**

The traffic shaper Dummynet uses the same basic structures and commands, but has some particular feature to configure its own capabilities. Dummynet can be used only with ipfw2 because they share the scanning engine. We can say that dummynet is a particular action that a matching packet must follow. The importance of dummynet is so relevant that some dedicated command were developed in the ipfw2 interface. However, the main program remains the ipfw2 core. The following example illustrate how to limit all the IPv4 bandwidth

- **ipfw add pipe 1 ip from any to any**
- **ipfw pipe 1 config bw 30Kbit/s**

For details please see [2] and [3].

7 Adding IPv6 Support

7.0.1 Add IPv6 support to Ipfw2

For adding IPv6 support to Ipfw2 and Dummynet we made the following interventions :

1. *IPFW2*: Add interception of IPv6 packets and creation of new filtered rules for it.
2. *ipfw*: Adduce user interface parser to new IPv6's opcodes.
3. *Dummynet*: Add support for IPv6 pipe/queue and moreover, support to calling the IPv6 stack.
4. *ip6_input,ip6_output*: Adding Ipfw2 hooks and Dummynet's packets management.

7.0.2 IPv6 support for IPFW2

The first intervention was the interception of packets from the IPv6 layer inside the function *ipfw_chk*. In the same way of IPv4, FreeBSD store network packets in a kernel of fixed arrays of struct **mbuf**:

```
/* Identify ipv6 packets and
 * fill up variables. */
if (pktlen >= sizeof(struct ip6_hdr) &&
 (!args->eh ||
   ntohs(args->eh->ether_type)
==ETHERTYPE_IPV6) &&
   mtod(m, struct ip *)->ip_v == 6)
```

the variable *pktlen* has the IP header dimension grabbed from *mbuf* struct, but the *mtod(m,struct ip*)* grabs the header from mbuf. Notice that the management of Ethernet packets differs from the management of IPv4/IPv6 packets, because in the first case there is also the ethernet header. The flag *is_ipv6* was set if the IPv6 protocol was found, and next it will be used for separating the IPv4 code-flow to the IPv6 code-flow. We must also grab the higher protocol header(ICMPv6,TCP,UDP), for collecting information that will used for matching rules.

```
/* Search extension headers to
   find upper layer protocols
*/
while (ulp == NULL) {
 switch (proto) {
```

```
  case IPPROTO_ICMPV6:
    PULLUP6(hlen, ulp, struct icmp6_hdr);
    args->f_id.flags = ((struct icmp6_hdr *)
   break;
.................................
```

If Ipfw2 finds the fragmentation header, it does not grab the higher protocol header while it will not have all packet information (defragmented packet). One of new the rules inserted in IPv6 was the filtering from extension header. Therefore you can see the flag *ext_hd* (bit vector) used for this intention. The pointer *ulp* was used for pointing the header of the higher protocol. After Ipfw2 stores some information, they are used by the filtered rules and Dummynet:

```
args->f_id.src_ip6 = mtod(m, struct
ip6_hdr *)->ip6_src;
args->f_id.dst_ip6 = mtod(m, struct
ip6_hdr *)->ip6_dst;
args->f_id.src_ip = 0;
args->f_id.dst_ip = 0;
rgs->f_id.flow_id6 = ntohs(mtod
(m, struct ip6_hdr *)->ip6_flow);
```

7.0.3 Add on for static rules

Now Ipfw2 checks the static rules throughout some *opcodes* for matching the information grabbed from the packet; if it has a good match then it sets the flag *match*.

```
switch (cmd->opcode) {
.........................

case O_ICMP6TYPE:
    match = is_ipv6 && offset == 0 &&
    proto==IPPROTO_ICMPV6 &&
    icmp6type_match(
        ((struct icmp6_hdr *)ulp)->icmp6_type,
        (ipfw_insn_u32 *)cmd);
    break;
.............................
```

ICMPv6 packets was matched by this function

```
static __inline int
icmp6type_match (int type, ipfw_insn_u32 *cmd)
{
      return (type <= ICMP6_MAXTYPE &&
      (cmd->d[type/32] & (1<<(type%32)) ) );
}


case O_IP6_SRC:
    match = is_ipv6 &&
    IN6_ARE_ADDR_EQUAL(&args->f_id.src_ip6,
            &((ipfw_insn_ip6 *)cmd)->addr6);
    break;
.................................
```

The last two opcodes are used for filtered packet sent from/to localhost (called "me6" to distinguish it from IPv4 "me"). This operation is made by the function *search_ip6_addr_net*, which gains the local IPv6 address and matches it with the packet address.

```
static int
search_ip6_addr_net (struct in6_addr * ip6_addr)
{
 struct ifnet *mdc;
 struct ifaddr *mdc2;
 struct in6_ifaddr *fdm;
 struct in6_addr copia;

 TAILQ_FOREACH(mdc, &ifnet, if_link)
 for (mdc2 = mdc->if_addrlist.tqh_first; mdc2;
    mdc2 = mdc2->ifa_list.tqe_next) {
    if (!mdc2->ifa_addr)
    continue;
    if (mdc2->ifa_addr->sa_family == AF_INET6) {
    fdm = (struct in6_ifaddr *)mdc2;
    copia = fdm->ia_addr.sin6_addr;
    /* need for leaving scope_id in
     * the sock_addr */
    in6_clearscope(&copia);
    if (IN6_ARE_ADDR_EQUAL(ip6_addr, &copia))
        return 1;
    }
 }
return 0;
}
```

Other OPCODES....

```
case O_FLOW6ID:
    match = is_ipv6 &&
    flow6id_match(args->f_id.flow_id6,
              (ipfw_insn_u32 *) cmd);
    break;
.............................
```

The opcode O_FLOW6ID is used to filter from IPv6 packet flow_id field. This operations is made by *flow6id_match* function.

```
static int
flow6id_match( int curr_flow,
ipfw_insn_u32 *cmd )
{
 int i;
 for (i=0; i <= cmd->o.arg1; ++i )
  if (curr_flow == cmd->d[i] )
   return 1;
  return 0;
}
```

7.0.4 Add on for dynamic rules

Dynamic rules are supported by IPv6 in according to this changes/add on:

- Adapting *lookup_dyn_rule* function used for search the rule and eventually the expiretion time.

```
if (IS_IP6_FLOW_ID(pkt)) {
 if (IN6_ARE_ADDR_EQUAL(&(pkt->src_ip6),
  &(q->id.src_ip6)) &&
  IN6_ARE_ADDR_EQUAL(&(pkt->dst_ip6),
  &(q->id.dst_ip6)) &&
  pkt->src_port == q->id.src_port &&
  pkt->dst_port == q->id.dst_port ) {
  dir = MATCH_FORWARD;
  break;
 }
if (IN6_ARE_ADDR_EQUAL(&(pkt->src_ip6),
  &(q->id.dst_ip6)) &&
  IN6_ARE_ADDR_EQUAL(&(pkt->dst_ip6),
  &(q->id.src_ip6)) &&
  pkt->src_port == q->id.dst_port &&
  pkt->dst_port == q->id.src_port ) {
  dir = MATCH_REVERSE;
  break;
}
```

- Adapting *lookup_dyn_parent* function used for adding new dynamic rule.

```
static ipfw_dyn_rule *
lookup_dyn_parent(struct ipfw_flow_id *pkt,
    struct ip_fw *rule)
{
 ipfw_dyn_rule *q;
 int i;
 ..............
     (is_v6 &&
      IN6_ARE_ADDR_EQUAL(&(pkt->src_ip6),
      &(q->id.src_ip6)) &&
      IN6_ARE_ADDR_EQUAL(&(pkt->dst_ip6),
      &(q->id.dst_ip6))) ||
      (!is_v6 &&
      pkt->src_ip == q->id.src_ip &&
      pkt->dst_ip == q->id.dst_ip)
     )
 ..................
```

- Adapting hash function *hash_packet*, used to store dynamic rules.

```
static __inline int
hash_packet(struct ipfw_flow_id *id)
{
 u_int32_t i;
 i = IS_IP6_FLOW_ID(id) ? hash_packet6(id):
    (id->dst_ip) ^ (id->src_ip) ^
    (id->dst_port) ^ (id->src_port);
 i &= (curr_dyn_buckets - 1);
 return i;
}

static __inline int
hash_packet6(struct ipfw_flow_id *id)
{
 u_int32_t i;
 i= (id->dst_ip6.__u6_addr.__u6_addr32[0]) ^
 (id->dst_ip6.__u6_addr.__u6_addr32[1]) ^
 (id->dst_ip6.__u6_addr.__u6_addr32[2]) ^
 (id->dst_ip6.__u6_addr.__u6_addr32[3]) ^
 (id->dst_port) ^ (id->src_port) ^
(id->flow_id6);
 i &= (curr_dyn_buckets - 1);
 return i;
}
```

7.0.5 Other changes

The *check_ipfw_struct* function is used for checking the validity of opcode. We added some new opcodes for IPv6 and they need to check inside this function.

```
case O_IP6_SRC:
case O_IP6_DST:
    if (cmdlen != F_INSN_SIZE(struct in6_addr)
       + F_INSN_SIZE(ipfw_insn))
    goto bad_size;
    break;
.................................
```

Antispoof algorithm implemented for IPv4 was replicated for IPv6 in the same way.

```
static int
verify_rev_path6(struct in6_addr *src,
struct ifnet *ifp)
{
 static struct route_in6 ro;
 struct sockaddr_in6 *dst;
 dst = (struct sockaddr_in6 * )&(ro.ro_dst);
 if ( !(IN6_ARE_ADDR_EQUAL
 (src, &dst->sin6_addr) )) {
   bzero(dst, sizeof(*dst));
   dst->sin6_family = AF_INET6;
   dst->sin6_len = sizeof(*dst);
   dst->sin6_addr = *src;
   rtalloc_ign((struct route *)&ro,
        RTF_CLONING | RTF_PRCLONING);
 }
 if ((ro.ro_rt == NULL) || (ifp == NULL) ||
    (ro.ro_rt->rt_ifp->if_index !=
ifp->if_index))
  return 0;
return 1;
}
```

7.0.6 Changes for Ipfw2 header

The first change to Ipfw header (ip_fw.h) was adding the new IPv6 opcodes in *ip_fw_opcode* structure

```
O_IP6_SRC,
O_IP6_SRC_ME,
O_IP6_SRC_MASK,
O_IP6_DST,
O_IP6_DST_ME,
O_IP6_DST_MASK,
O_FLOW6ID,
O_ICMP6TYPE,
O_EXT_HDR,
O_IP6,
```

We also defined some codes for Extension Header, used for matching-filtered rules.

```
/*
 * The extension header are filtered only for
 * presence using a bit vector
 * with a flag for each header.
 */
#define EXT_FRAGMENT    0x1
#define EXT_HOPOPTS 0x2
#define EXT_ROUTING 0x4
#define EXT_AH      0x8
#define EXT_ESP     0x10
```

The structure *ipfw_isn_ip6* is used for matching rules for source/destination address.

```
/* Structure for ipv6 */
typedef struct _ipfw_insn_ip6 {
 ipfw_insn o;
 struct in6_addr addr6;
 struct in6_addr mask6;
} ipfw_insn_ip6;
```

In the new ICMPv6 protocol we need a new structure called *ipfw_insn_icmp6*, you can see the new implementation of ICMP in *rfc2542*. The "types" of ICMPv6 codes are many more and are defined in *netinet/icmp6.h*, and now they are 203 types. A bit vector structure permit to filterer multi ICMPv6 types in the same rule.

```
#define IPFW2_ICMP6_MAXV 7
typedef struct _ipfw_insn_icmp6 {
    ipfw_insn o;
    uint32_t d[IPFW2_ICMP6_MAXV];
} ipfw_insn_icmp6;
```

The structure *ip_fw_flow_id* is used to distinguish IPv4 addresses and IPv6 addresses, therefore the structure *ip6_dn_args* is used to store some parameters used by Dummynet (see later).

7.0.7 IPv6 support for Dummynet

The relevant changes for Dummynet are related to the I/O sections. They also build new pipe/queue for IPv6 packet and a new hash table function for IPv6 packets. The I/O section is composed by two functions:

- *trasmit_event*: It is invoked when Dummynet need to insert the packet in a queue. This function is periodically called also by the scheduler, in accordance to policies. We have inserted it in the calling to IPv6 stack through *ip6_input* and *ip6_output* functions.

```
static void
transmit_event(struct dn_pipe *pipe)
{
.....................................
    case DN_TO_IP6_IN:
     ip6_input((struct mbuf *)pkt) ;
     break ;
    case DN_TO_IP6_OUT:
     (void)ip6_output((struct mbuf *)pkt,
          NULL, NULL,0, NULL, NULL, NULL);
     rt_unref (pkt->ip6opt.ro_or.ro_rt) ;
     break ;
.....................................
```

- *dummynet_io*: This function is used to insert or create the pipes for IPv6 packets.

```
static int
dummynet_io(struct mbuf *m, int pipe_nr,
.....................................
 } else if (dir == DN_TO_IP6_OUT) {
  memcpy( &(pkt->ip6opt.ro_or),
    &(fwa->dummypar.ro_or),
   sizeof(fwa->dummypar.ro_or));
   if (fwa->dummypar.ro_or.ro_rt)
   fwa->dummypar.ro_or.ro_rt->rt_refcnt++;
   if (fwa->dummypar.dst_or ==
     (struct sockaddr_in6 *) &
     (fwa->dummypar.ro_or.ro_dst));
     fwa->dummypar.dst_or =
     (struct sockaddr_in6 *) &
     (pkt->ip6opt.ro_or.ro_dst);
    pkt->ip6opt.dst_or =
        fwa->dummypar.dst_or;
    pkt->ip6opt.flags_or =
        fwa->dummypar.flags_or;
 }
 if (q->head == NULL)
.....................................
```

If you have an **output pipe**, then you must store routing parameters, like the entire structure *ro*, the source address, the destination address, and the interface (*ifp*) where the packet is from. This is necessary because Dummynet grabs the packet from stack and store it in internal queue. In according to policies Dummynet re-inserts the packet later but in this delay the routing parameters stored in the routing table can be lost. If we do not save these parameters we will be in trouble because when Dummynet will re-insert them the packet stack will cause a kernel panic!!!

7.0.8 Dummynet - other changes

A new hash packet function was inserted for IPv6 packets, in the same way of the IPv4 function; this change is implemented in the *find_queue* function.

```
.....................................
if (is_v6) {
 APPLY_MASK(&id->dst_ip6,
```

```
&fs->flow_mask.dst_ip6);
 APPLY_MASK(&id->src_ip6,
&fs->flow_mask.src_ip6);
 id->flow_id6 &= fs->flow_mask.flow_id6;
 i = ((id->dst_ip6.__u6_addr.__u6_addr32[0])
& 0xffff)^
 ((id->dst_ip6.__u6_addr.__u6_addr32[1])
& 0xffff)^
 ((id->dst_ip6.__u6_addr.__u6_addr32[2])
& 0xffff)^
 ((id->dst_ip6.__u6_addr.__u6_addr32[3])
& 0xffff)^
.................................
 (id->dst_port << 1) ^ (id->src_port) ^
 (id->proto ) ^
 (id->flow_id6);
.................................
```

7.0.9 Dummynet - Header changes

The changes to Dummynet header was some added parameters in *dn_pkt* structure for IPv6 management.

```
struct dn_pkt {
.................................
#define DN_TO_IP6_IN    6
#define DN_TO_IP6_OUT   7
 dn_key output_time;
.................................
 struct _ip6dn_args ip6opt;
};
```

We added the "direction" to IPv6 stack (*DN_TO_IP6_IN e DN_TO_IP6_OUT*) and to the *ip6opt* structure. This structure contains the routing information saved before inserting the packet in the Dummynet queue.

```
struct  _ip6dn_args {
 struct route_in6 ro_or;
 int flags_or;
 struct ifnet* ifp_or,origifp_or;
 struct sockaddr_in6* dst_or;
};
```

7.1 Hooks to IPv6 stack

We have added in *ip6_input* and *ip6_output* some information that permits compatibility between Ipfw2 and Dummynet. The intention of this work is:

1. Intercept the IPv6 packets in input and output and next, calling Ipfw2 to make they destiny.

2. Send the packets to the Dummynet queue and reinsert them next.

7.1.1 Changes to ip6_input

```
/* now check with the firewall ipfw2 */
if (fw_enable && IPFW_LOADED) {
.....................
  goto pass6;
  if (DUMMYNET_LOADED &&
      (i & IP_FW_PORT_DYNT_FLAG) != 0) {
    /* Send packet to the appropriate pipe */
      ip_dn_io_ptr(m, i & 0xffff,
        DN_TO_IP6_IN, &args);
      return;
  }
.....................
}
.............
```

When you extract the mbuf structure (we remind that the mbuf structure in FreeBSD contains packet information), you copy it in the *args* structure used by Ipfw2 by calling it. The Ipfw call returned code decides if the packet will be accepted or dropped. If Dummynet was loaded and there is a correspondent rule for the packet, then *i* variable has stored the number of pipe/queue for it. When Dummynet will re-insert the packet from the queue, it TAGs the packet.

```
.................
 case PACKET_TAG_DUMMYNET:
    args.rule = ((struct dn_pkt *)m)->rule;
    break;
.............................
*/
 }
}
 KASSERT(m != NULL && (m->m_flags & M_PKTHDR) != 0,
    ("ip6_input: no HDR"));
 if (args.rule) {/* dummynet already filtered us */
 ip6 = mtod(m, struct ip6_hdr *);
 hlen = sizeof (struct ip6_hdr);
 goto send_after_dummynet ;
}
```

When ip6_input receive a packet, it searches the existent "TAG" generated by Dummynet or Ipfw2 and if this is true, it forwards it. You can see that the packet never cycles out of this scheme.

7.1.2 Changes to ip6_output

```
if (fw_enable && IPFW_LOADED && !args.next_hop) {
struct sockaddr_in6 *old = dst;
args.m = m;
args.next_hop = (struct sockaddr_in *) dst;
args.oif = ifp;
off = ip_fw_chk_ptr(&args);
m = args.m;
dst = (struct sockaddr_in6 *) args.next_hop;
.............
if (DUMMYNET_LOADED &&
   (off & IP_FW_PORT_DYNT_FLAG) != 0) {
.............
```

```
 args.dummypar.ro_or = *ro;
 args.dummypar.flags_or = flags;
 args.dummypar.ifp_or = ifp;
 args.dummypar.origifp_or = origifp;
 args.dummypar.dst_or = *dst;
 args.flags = flags;
 error = ip_dn_io_ptr(m, off & 0xffff,
     DN_TO_IP6_OUT, &args);
 goto done;
}
}
pass6:
```

ip6_output saves the mbuf structure in the *args* structure and then calls Ipfw2. The Ipfw2 returned code decides the destiny of packet, and if Dummynet was loaded, the packet flow will be the same of the ip_input. Ip6_ouptut, before calling Dummynet, saves the routing parameters of packet (ro), the network interface parameters(ifp,orig_ifp), and the destination socket (st). These parameters are very important!! When Dummynet will re-insert the packet, it will restore the parameters, otherwise ip6_output can not forward the packet and will cause a kernel panic.

```
.............................
case PACKET_TAG_DUMMYNET:
 opt = NULL;
 ro = &((struct dn_pkt *)m0)->ip6opt.ro_or;
 flags = ((struct dn_pkt *)m0)->
ip6opt.flags_or;
 im6o = NULL;
 origifp = ((struct dn_pkt *)m0)->
ip6opt.origifp_or;
 ifp = ((struct dn_pkt *)m0)->ip6opt.ifp_or;
 dst =  &((struct dn_pkt *)m0)->ip6opt.dst_or;
 args.rule=((struct dn_pkt *)m0)->rule;
 if (args.rule != NULL)
   printf("Collecting parameters\n");
   break;
.............................
if (args.rule ) { /* dummynet already saw us */
 ip6 = mtod(m, struct ip6_hdr *);
 hlen = sizeof (struct ip6_hdr) ;
 if (ro->ro_rt)
  ia = ifatoia6(ro->ro_rt->rt_ifa);
  bzero(&exthdrs, sizeof(exthdrs));
  ro_pmtu = ro;
  goto send_after_dummynet;
}
```

When the ip6_output receive a packet it searches for existent "TAG" generated by Dummynet or Ipfw2. If this is true, first it restores the routing parameters, the interface flag, the socket flag, and saved MTU value, then it forwards them. You can see that the packet never cycles in this scheme.

7.2 Test phase

The tests were performed to verify the new features. We built some rules in a network that support IPv6, and we checked the response from Dummynet and Ipfw2. The tests for the user interface (practically the parser), was simply the insertion of rules from the shell done. The various tests consisted of adding new rules for the corresponding case and verifying it by building network traffic flow in the same way. The good insertion of rule granted the successful test for parser. Now, if the counter of rule increment said that *ip6_input/output* was working fine, this is a successful test for the hooks to IPv6 stack. The tests for static rule was

1. **Test for filtering single address**: We has inserted a rule like this:
   ipfw add deny ipv6 from fe80::250:baff:fe78:5941 to me
   The shell output is:
   00100 deny ipv6 from fe80::250:baff:fe78:5941 to me6

   We have tested the rule taking access from the machine fe80::250:baff:fe78:5941 to localhost, like telnet for TCP,ping for ICMPv6, ssh (TCP with SSL), and traceroute for UDP, and next we verified that the traffic passed from the other address.

2. **Test for filtering multi address**: We inserted a rule like this:
   ipfw add deny ipv6 from fe80::250:baff:fe78:5941, fe78::250::fe78:3342 to me
   You can insert an arbitrary number of addresses for it. The test was the same of the previously rule.

3. **Test for filtering address with some subnet-mask:** We inserted a rule like this:
   ipfw add deny ipv6 from fe80::250:baff:fe78:5941/80 to me
   The shell output is:
   00100 deny ipv6 from fe80::250:baff:fe78:5941/80 to me6
   The test was the same of the previously rule but now we have distinguished the hosts from the subnet mask.

4. **Test for filtering me6 option:** See the previously rules.

5. **Test for filtering higher protocol (TCP/UDP/ICMPv6):** For testing TCP packet filtering we inserted the rule:
   ipfw add deny tcp from fe80::250:baff:fe78:5941/80 to me
   The shell output is:

00100 deny tcp from fe80::250:baff:fe78:5941/80 to me6
The tests was the same of the previous rules but in this case the ping6 (ICMPv6) and traceroute (UDP) packets sent from fe80::250:baff:fe78:5941/80 was passed, but telnet and ssh (TCP) were not.
For testing UDP packet filtering we inserted the rule
ipfw add deny udp from fe80::250:baff:fe78:5941/80 to me
The tests was the same, but in this case only Traceroute (UDP) packets from the host were dropped, and the other packets passed.
For testing ICMPv6 packet filtering we inserted rules like:
ipfw add deny icmp6 from fe80::350:baff:fe78:5941/80 to me
ipfw add deny icmp6 from fe80::250:baff:fe78:5941/80 to me icmp6types 16,127
The tests was the same, but in the first case all ping6 packets from the host were dropped, all the others passed. In the last case only ping6 packet typed 16 and 127 were dropped.

6. **Test for filtering IPv6 flow-id:** For testing the flow-id filter we inserted a rule like this:
ipfw add deny ipv6 from any to any flow-id 20,30,50
This rule blocks all datagram that have a flow id specificated. We developed a little program for testing this functionality. The program opens a UDP socket with the host and sends it to a datagram with the desired flow id.

7. **Test for filtering IPv6 Extensions-header:** For testing extension-header filters we inserted a rule like this:
ipfw add deny ipv6 from any to any ext6hdr frag
This rule drop all datagrams that was fragmented, you can insert some options like flow id.

The other tests are for *dynamic* rules and they are related to the limit in the the number of connections TCP/UDP from/to host. So, to test this we inserted rules like:
ipfw add allow tcp from fe80::250:baff:fe78:5941 to me setup limit src-addr 4
ipfw add allow udp from fe80::250:baff:fe78:5941 to me setup limit src-addr 4
This rule limits the number of connections between TCP and UDP to only 4 from the host. The tests was to take some TCP/UDP access from the machine fe80::250:baff:fe78:5941 to localhost, and then verify that if the number of connections is above to 4 they will be dropped.

The tests for Dummynet verify the functionally for these rules:

- *Testing Pipe rules:* We has inserted a rule like this:
ipfw add pipe 1 ipv6 from me to any
ipfw pipe 1 config bw 30Kbit/s
This rule creates a IPv6 pipe and sets the flow speed to 30Kbit/s. From the delay value of ping6 you must verify the correct flow speed.

- *Testing Queue rules:* We inserted the rule:
ipfw add pipe 1 ipv6 from me to any
ipfw pipe 1 config bw 64Kbit/s queue 10Kbytes
This rule creates a IPv6 pipe and sets a flow speed of 64Kbit/s, then it configures a pipe that limits the traffic flow to 10KBytes.

AUTHORS

RAFFAELE DE LORENZO received the Laurea degree (5 years) in Computer Science Engineering from the University of Pisa (Italy) in 2003. He is currently a System Network and Security Engineer with famous Italian bank trough important society of IT, where he develops network security for the Bank.

LUIGI RIZZO received a Ph.D. degree in Electronic Engineering from the SSSUP S. Anna in Pisa, Italy in 1993. Since 1991 he has been with the Dipartimento of Ingegneria dell'Informazione at the University of Pisa,where he currently is Associate Professor.

MARIANO TORTORIELLO received the Laurea degree (5 years) in Computer Science Engineering from the University of Pisa, Italy, in 2003. He is the executive director of a small mechanical engineering company.

References

[1] Paolo Valente - *Sviluppo di un sistema di Fair Queuing in ambiente Unix*, Tesi di laurea, Ottobre 2000

[2] ipfw2 manual: *man ipfw http://www.FreeBSD.org/cgi/man.cgi?query=ipfw*

[3] dummynet manual:*man dummynethttp://www.FreeBSD.org/cgi/man.cgi?query=dummynet.*

[4] C. Patridge. *Request for Comment 1809: Using the flow label field in IPv6*, Giugno 1995.

[5] IAB, IESG. *Request for Comment 1881: IPv6 Address Allocation Management*, Dicembre 1995.

[6] Y. Rekhter, T. Li. *Request for Comment 1887: An Architecture for IPv6 Unicast Address Allocation*, Dicembre 1995.

[7] R. Elz. *Request for Comment 1887: A Compact Representation of IPv6 Addresses*, Aprile 1996.

[8] R. Callon, D. Haskin. *Request for Comment 2185: Routing Aspects Of IPv6 Transition*, Settembre 1997.

[9] S. Deering, R. Hinden. *Request for Comment 2460: Internet Protocol, Version 6 (IPv6) Specification*, Dicembre 1998.

[10] A. Conta, S. Deering. *Request for Comment 2463: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*, Dicembre 1998.

[11] M. Crawford. *Request for Comment 2464: Transmission of IPv6 Packets over Ethernet Networks*, Dicembre 1998.

[12] D. Haskin, S. Onishi. *Request for Comment 2465: Management Information Base for IP Version 6: Textual Conventions and General Group*, Dicembre 1998.

[13] D. Haskin, S. Onishi. *Request for Comment 2466: Management Information Base for IP Version 6: ICMPv6 Group*, Dicembre 1998.

[14] D. Johnson, S. Deering. *Request for Comment 2526: Reserved IPv6 Subnet Anycast Addresses*, Marzo 1999.

[15] D. Borman, S. Deering, R. Hinden. *Request for Comment 2675: IPv6 Jumbograms*, Agosto 1999.

[16] R. Hinden, B. Carpenter, L. Masinter. *Request for Comment 2732: Format for Literal IPv6 Addresses in URL's*, Dicembre 1999.

[17] R. Gilligan, E. Nordmark. *Request for Comment 2893: Transition Mechanisms for IPv6 Hosts and Routers*, Agosto 2000.

[18] B. Haberman, D. Thaler. *Request for Comment 3306: Unicast-Prefix-based IPv6 Multicast Addresses*, Agosto 2002.

[19] R. Draves. *Request for Comment 3484: Default Address Selection for Internet Protocol version 6 (IPv6)*, Febbraio 2003.

[20] R. Gilligan, S. Thomson, J. Bound, J. McCann, W. Stevens. *Request for Comment 3493: Basic Socket Interface Extensions for IPv6*, Febbraio 2003.

[21] R. Hinden, S. Deering. *Request for Comment 3513: IPv6 Addressing Architecture*, Aprile 2003.

[22] W. Stevens, M. Thomas, E. Nordmark, T. Jinmei. *Request for Comment 3542: Advanced Sockets Application Program Interface (API) for IPv6*, Maggio 2003.

[23] R. Hinden, S. Deering, E. Nordmark. *Request for Comment 3587: IPv6 Global Unicast Address Format*, Agosto 2003.

[24] B. Wijnen. *Request for Comment 3595: Textual Conventions for IPv6 Flow Label*, Settembre 2003.

[25] J. Rajahalme, A. Conta, B. Carpenter, S. Deering. *Request for Comment 3697: IPv6 Flow Label Specification*, Marzo 2004.

[26] Larry L. Peterson & Bruce S. Davie. *Computer Networks - A System Approach* Second Edition. Morgan Kaufmann Publishers (S. Francisco, California).

[27] The Kame Project - *http://www.kame.org*

What's new in NetBSD in 2006 ?

Emmanuel Dreyfus

October 24, 2006

Abstract

NetBSD is known as a highly portable operating system, but its strengths are not limited to being available on many platforms. NetBSD goals also include security, performance, standards conformance and clean design. Development of innovative features also occurs.

In this paper, we will have a look at the new features that have been integrated into NetBSD this year.

1 NetBSD in the news

1.1 Dead or irrelevant?

Thanks to the numerous and valuable contributions from Slashdot's anonymous coward, we are now all aware that *BSD is dying [1]. While the recurrent Slashdot troll gave us strong warnings about FreeBSD's and OpenBSD's imminent deaths for years, NetBSD was often omitted. Did that mean NetBSD was already dead?

The EuroBSDCon 2005 social event was called "the night of the living dead", in reference to the Slashdot troll. That was an attractive point of view, since it implied that dead projects like the *BSD could be alive and kicking after all. Unfortunately, NetBSD did not show any sign of life that night, as it was even outperformed by DragonFly BSD at the beer drinking contest.

The few people who still remembered an OS called NetBSD were still puzzled about the death of NetBSD: did it occur while nobody was watching? Fortunately, on the 30th of August 2006, one of the NetBSD project founders sent an insightful message to the `netbsd-users@netbsd.org` mailing list [2]. In that message, Charles M. Hannum explained that NetBSD had increasingly become irrelevant. That post was reported by Slashdot, which drove a lot of attention to NetBSD. Charles shortly followed up with an interview at OnLAMP.com [3], entitled "Confessions of a Recovering NetBSD Zealot".

Thanks to Charles, things were clear: the project was not dead, it was just irrelevant, and every Slashdot reader knew about it. That is not very good news, but at least this had the advantage of showing that NetBSD was alive enough to upset someone and make the Slashdot cover page.

As far as I am aware, no news site tried to investigate Charles' claims by interviewing other NetBSD insiders about the affair. I assume I have to make a few comments on the topic.

In my opinion, Charles has various valid points. Indeed NetBSD could be better managed. It could also have more features, fewer bugs, and be more popular. Unless a project is really dead, people always expect more than what they get. There is always room for improvement.

But Charles' judgement as NetBSD being now irrelevant is just a personal opinion, and not everybody shares it. There is still a lot of work done on the project from several dozens of developers (the activity can be monitored through the `source-changes@netbsd.org` mailing list [4]). Obviously that crowd does not consider NetBSD as irrelevant.

The project is also recruiting new developers at a steady rate of few persons per month. That fresh blood shows that we even have newcomers considering NetBSD as an OS relevant enough to start working on it.

And finally, we still have a lot of users, as we will see in the next section.

Charles' detractors will note that his complaints come after years of inactivity as a NetBSD developer, and at the time the board of the NetBSD foundation decided to evict him [5] because he refused to sign the NetBSD developer agreement [6]. Charles' answer to his detractors is available in the press. This is not a nice story, but the only real bad point I will retain is that we have lost a valuable contributor to the project. That is not the first time such a thing has happened, and it will certainly not be the last one: no open source project can retain all its contributors forever. Let us hope we will not do that too often.

1.2 Bugathons

We have received positive press on the two NetBSD bugathons [7]. These are IRC meetings where NetBSD developers and users meet to work on resolving problem reports together [8].

The two first bugathons were organized by Elad Efrat. They occurred on during September 23rd-24th [9], and October 7th-8th week-ends [10].

Both events were huge successes. According to Elad's reports, the first edition gathered 30 developers and 20 users, and resulted in 270 problem reports (PR) being closed. The second edition gathered 3 times more people and resulted in 310 PRs being closed.

Of course, hundreds of PRs being closed do not mean hundreds of bugs being fixed, as many open PRs are duplicates or obsoletes, or even come with a bug fix that just needs to be reviewed and committed. Still, such events are excellent news for the project, as it means we finally found a way to deal with the never-ending accumulation of open PRs [11].

The other very good point about the bugathons is that it clearly shows that despite the claims about NetBSD's irrelevancy, there is still a strong user community around NetBSD.

I hope we will see many more bugathons, and that neither the user community, nor the developers will get tired of them. The open PR database had gotten way too fat, it was high time to make it slim again!

1.3 Google Summer of Code

This year, NetBSD was involved for the second time in Google Summer of Code [12]. This year, 8 projects were started by students [13]:

Support for journaling for FFS : The Berkeley Fast File System (FFS) is NetBSD's preferred file system. Adding a journaling feature to it would remove the need for long file system checks when the system reboots after a power outage. As disks get bigger and cheaper, the time for a file system check has grown far too long for many users.

Support for MIPS64 : NetBSD runs on a large range of hardware, including MIPS based machines. NetBSD also has all the necessary infrastructure to run on 64 bit processors (such as alpha, sparc64, or amd64), but it lacked the machine-dependent bits for running MIPS processors in 64 bit mode.

PowerPC G5 support in NetBSD : This is about adding the machine-dependent bits to get the PowerPC G5 processor supported by NetBSD.

Improved writing to file system using congestion control : In a multiuser environment, several processes can write at once to the same file system, thus causing congestion. The goal of this project was to establish benchmark tools and to research solutions to file system congestion problems.

TCP ECN support : Explicit Congestion Notification (ECN) is a set of congestion control mechanisms described in RFC 3168 [14]. At the TCP level, it works by having the sender adjust the transmission window size to handle congestion notifications sent by routers. Supporting this feature would enable NetBSD to perform better on overloaded networks (provided the routers also support ECN).

FAST_IPSEC and IPv6 : The original IPsec implementations in *BSD kernels was derived from the KAME project [15]. It exhibited poor scalability, and was unable to take

advantage of specialized hardware accelerators to perform the cryptographic computation. The `FAST_IPSEC` [16] kernel option was created to deal with this issue, but it lacked IPv6 support.

`pkg_install` rewrite for pkgsrc : The NetBSD package collection [17] is based on a set of tools that have evolved over the years. It seems the tools have reached the point where a major cleanup is necessary. The goal of this project was to collect the requirements for pkgsrc tools, and re-implement them based on a new clean design.

Improving the `mbuf` API and implementation : `mbuf` [18] is the infrastructure used by kernel networking code to manage memory. The current programming interface features many pitfalls, and it is easy to write buggy code that makes wrong assumptions. The goal of the project was to clean up the programming interface to make it easier to deal with.

Hubert Feyrer's press release [13] gives us this year's result. There has been some successful stories: PowerPC G5 support, ECN implementation, `FAST_IPSEC` and IPv6, and `mbuf` cleanup project were completed. It is also worth noting that the mbuf cleanup project has opened the way to a zero-copy I/O implementation in NetBSD/Xen. Once completed, this should produce a noticable performance win.

pkgsrc infrastructure rewrite and file system congestion control were not fully completed. While not completely done, the file system congestion control still led to interesting performance improvements. On the pkgsrc front, a paper from Joerg Sonnenberger [19] details the recent changes.

And finally, we had two failed projects: the student in charge of the journalised FFS project simply vanished, and the one in charge of MIPS 64 ran into a health problem that prevented him from completing the work in time.

This year again, we have to thank Google for sponsoring our development. Driving students to NetBSD is an excellent thing for the project, as it means new contributors and new features implemented.

1.4 pkgsrcCon 2006

The third pkgsrcCon took place in University of Paris 7 – Denis Diderot [20]. The goal of this technical conference is to gather developers and users of the NetBSD package system, also known as pkgsrc. Here is the conference program:

- Stoned Elipot, System Administration with pkgsrc [21]
- Joerg Sonnenberger, pkgsrc on DragonFly – or Fighting the Windmills [22]
- Roland Illig, `pkglint`: Static Analyzer For Pkgsrc [23]
- Roland Illig, Why Pkgsrc Sucks [24]
- Emile Heitor, `pkg_select` – So Many Packages, So Few Columns [25]
- Thomas Klausner, Roundtable Discussion: Updating Packages [26]
- Adrian Portelli, pkgsrc security one year on... [27]
- Dieter Baron, Thomas Klausner, `pkg_install` Rewrite [28]
- Johnny Lam, Roadmap for Development [29]

2 A few new exciting features

2.1 Xen

Xen [30] is one of the latest hot topics in the world of virtualization. Virtualization is about running multiple OSes at the same time on the same machine. It makes system management easier, as a virtual machine can be easily cloned or migrated to another real machine. Virtualization also offers easier system debugging, and allows hardware resources such as memory and CPU to be easily shared.

For instance, I use Xen virtual machines to run a virtual network with two hosts and a Network Address Translator (NAT) on the same machine. I use that setup to quickly make regression tests on IPsec-tools-based VPN [31] setups.

Virtualization usually works by featuring a host OS, which holds access to the real hardware, and guest OSes, which see virtual hardware. Early virtualization software worked by catching guest OS access to the hardware through exceptions. This enabled running unmodified versions of the guest OS but had a huge cost in performance.

Xen reached unprecedented levels of performance by requiring guest OSes to be modified. The guest OS is now aware it is running on a virtual machine, and accesses the virtual hardware through a well defined API. That approach removed the costly game of generating hardware exceptions for any hardware access such as a reading data from a disk. In Xen terminology, guest OSes are called domU, while the host OS is called dom0. It is worth mentioning, that domU and dom0 all run on the top the Xen kernel. Xen delegates hardware management to the kernel in the dom0 OS. This approach has the advantage of freeing Xen development from writing drivers.

Performance comparison of Xen versus various competitors is available from the Xen web site, and from a third party research group [32].

NetBSD 3.0 already implemented support for Xen 2.0, both as a domU and a dom0. Recently, Christian Limpach and Manuel Bouyer implemented support for Xen 3.0, both as domU and dom0.

Xen 3.0 has a few interesting new features, including:

- Support for up to 32 way SMP guests.
- Hardware-assisted virtualization (Intel VT-x and AMD-V Pacifica), which allows running unmodified guest OS.
- 64 bit support for the AMD64 architecture (not supported by NetBSD yet).

There has also been a lot of code rewriting behind the scenes, but that is not usually considered an interesting feature.

Finally, it is worth mentioning that benchmarks showed superior disk I/O performance of NetBSD as a dom0, compared to Linux [33].

2.2 iSCSI

iSCSI stands for Internet Small Computer System Interface. It is an encapsulation of the SCSI protocol over TCP/IP, documented in RFC 3720 [34], used for Storage Area Network (SAN).

The basic idea of a SAN is to have file servers exporting disk space as a block device, instead of exporting it as a file system, through protocols such as Unix's Network File System (NFS), Windows' Common Internet File System (CIFS), or Apple's Appleshare File Protocol (AFP). It frees the server from the burden of maintaining a file system, and allows easier storage resource sharing and extension.

iSCSI is a hot topic, because it allows building affordable SANs, based on ubiquitous Ethernet and TCP/IP network infrastructure, whereas SAN have traditionally been using specialized hardware, based for instance on fibre channel.

In iSCSI terminology, there is an iSCSI target, which exports selected storage as a block device, and an iSCSI initiator, which accesses the block device exported by the target. Of course, unless you use some kind of a distributed file system which can be mounted by several OSes at once, there can be only one initiator using a target at a time.

Alistair G. Crooks worked on integrating the iSCSI target support developed at Intel, and published a set of HOW-TOs [35], which explain how to set up NetBSD as an iSCSI target, and how to set up MS Windows XP as an iSCSI initiator that uses it. Alistair also presents a paper on iSCSI at EuroBSDCon 2006 [36].

The iSCSI initiator code in NetBSD is still a work in progress, and so is the support for iSCSI authentication mechanisms.

2.3 The build infrastructure now creates ISO images

NetBSD enjoys a unique build infrastructure, which allows extremely easy cross-building from another OS, or from NetBSD itself [37]. That infrastructure made automatic NetBSD builds for all NetBSD ports [38] not only possible, but even affordable.

There has been a long-term missing item in this auto build machinery: the only bootable media it was able to produce were installation floppy disk images. As today's modern hardware more and more often ships without a floppy disk drive, the lack of automatic bootable ISO image was becoming a concern.

That missing feature has been implemented, thanks to the work of various contributors.

First, Daniel Watt, Walter Deignan, Ryan Gabrys, Alan Perez-Rathke, Ram Vedam, and Luke Mewburn, improved NetBSD's `makefs` [39] utility, to support the ISO 9660 format. The purpose of `makefs` is to allow creation of file system images without the need of root privileges. It was initially developed to create FFS images of the RAM disks used in install kernels, so that this operation could be performed during the build process.

The second step was to actually use that feature in the NetBSD build infrastructure. Alan Barrett did the appropriate changes. Thanks to this work, NetBSD is now able to provide bootable ISO images as part of the regular NetBSD-current auto builds.

For now, the ISO images produced through the auto build machinery just contains the installation program, and not the installation sets (i.e.: `base.tgz`, `comp.tgz`, and so on), leaving the generation of a stand-alone installation ISO images as a future work.

2.4 WPA

NetBSD supports various IEEE 802.11 wireless devices. Unfortunately, we did not support the Wi-Fi Protected Access (WPA) protocol [40], which was a shame, since it only left our users with the alternative of using the Wired Equivalent Privacy (WEP) protocol [41], which is well-known for being insecure [42], or VPN-based setups, which are much more complicated to set up.

Thanks to Steve Woodford and Rui Paulo [43], NetBSD now has support for joining a wireless network protected by WPA. Steve and Rui integrated `hostapd` [44] and `wpa_suplicant` [45] from Jouni Malinen's WPA for Linux project [46]. That software includes WPA and WPA2 support both when acting as an access point and as a client. Advanced access point features such as RADIUS are also supported.

2.5 Bluetooth

Bluetooth [47] is a complete stack of wireless protocols standardized by the IEEE 802.15.1 task group for usage in Personal Area Networks (PAN). It is used for communication between various hand-held devices such as cell phones and PDA, or with devices such as hand-free headsets.

Iain Hibbert worked hard on implementing a complete bluetooth stack on NetBSD [48]. Iain started the work on his own, and was later sponsored by Itronix, Inc.

There is a page on the unofficial NetBSD Wiki [49] that shows bluetooth configuration and usage for using bluetooth Human Interface Devices (HID), serial links, audio headsets, and audio hands free devices.

2.6 UDF

Universal Disk Format (UDF) [50] is a file system designed for storing files on optical media. It is developed by the Optical Storage Technology Association (OSTA), and is also known as the ISO 13346 standard.

UDF is seen as the successor to the ISO 9660 format. It is used in DVDs, but can also be used in CD-ROMs or USB flash memories. As more and more optical disks using this format

appear, not being able to read them was a growing annoyance. Thanks to Reinoud Zandijk's work [51], this problem is now solved.

According to Reinoud, the NetBSD UDF implementation is able to read UDF file systems up to version 2.60 that are found on CD-ROM, CD-R, CD-RW, CD-MRW, DVD-ROM, DVD*R, DVD*RW, DVD+MRW disks, and it should be able to read DVD-RAM, HD-DVD, and BluRay disks. Disks do not need to be closed.

Note that support is currently limited to read-only. Read/write support is still a work in progress.

3 Networking

3.1 CARP

The Common Address Redundancy Protocol (CARP) [52] appeared in OpenBSD as a free alternative to Internet Engineering Task Force (IETF) blessed Virtual Router Redundancy Protocol (VRRP) [53] and Hot Standby Router Protocol (HSRP) [54], which are encumbered by Cisco patents.

CARP allows multiple hosts to share an IP address. The main usage for this feature is to build redundant firewalls, but it can also be used for load balancing.

Liam J. Foy imported OpenBSD's CARP to NetBSD [55].

3.2 Link aggregation

YAMAMOTO Takashi committed his implementation of the IEEE 802.3ad Link Aggregation Control Protocol (LACP) [56]. This allows bonding of several Ethernet interfaces into a single virtual `agr(4)` [57] interface.

The current implementation has a few limitations, see the `agr(4)` [57] man page for details.

3.3 NDIS wrapper

Network Driver Interface Specification (NDIS) is a generic programming interface developed by Microsoft and 3com for network interface drivers [58]. Most, if not all, network device vendors will give away NDIS drivers for the products they sell.

The NDIS interface is well defined and NDIS driver are not supposed to access Windows internals without going through the NDIS interface. That means an NDIS driver designed for Windows could work on another OS, provided that a translation layer is set up between the NDIS interface and the native OS.

This is exactly what the NDIS wrapper project is about. By implementing an NDIS compatibility kernel option, it is possible to use binary drivers built for Windows on other OSes. Of course, that is limited to drivers built for the same processor, which usually means i386 only.

NDIS wrapper was first developed for FreeBSD [59], and later adopted by Linux [60]. Thanks to the work done by Alan Ritter during last year's Google Summer of Code [61], NetBSD now also enjoys that feature.

For more information on how to use it, see the `ndiscvt(8)` [62] man page.

4 Storage and file systems

More features beyond iSCSI and UDF:

4.1 tmpfs

`tmpfs` is a new memory-based file system, which was designed by Julio M. Merino Vidal as a 2005 Google Summer of Code project [63].

The goal of `tmpfs` is to replace MFS. The problem with MFS is that it is just FFS hacked to store files in memory instead of on a disk, thus resulting in poor memory usage.

4.2 scan_ffs

Who never erased a partition table by mistake? This error is especially irritating, since your data is still on the disk, but you cannot reach it anymore.

OpenBSD developed a `scan_ffs` utility to solve that problem. Its purpose is to search the disk for a FFS file system, so that you have an opportunity to reconstruct your partition table and recover access to your data.

Thanks to Juan Romero Pardines, NetBSD now also enjoys the `scan_ffs(8)` [64] utility. It is worth noting that Juan also added LFS and FFSv2 support to `scan_ffs`.

4.3 LFS improvements

LFS stands for Log-structured File System [65]. Traditional file systems have been designed with the idea that the hard disk seek time was the bottleneck to I/O performances. This is no longer true if system memory is so big that everything is read from cache, or if the media is not a hard disk.

The idea behind LFS is to write to the disk sequentially, without doing any efforts so that a file can be read sequentially. As files are modified, all changes to the files are saved on disk and mix with each other. LFS write throughput is blazingly fast compared to other file systems. Another interesting feature is the ability to resize the file system while it is mounted.

The LFS implementation used in BSD systems was not maintained enough to remain usable. FreeBSD and OpenBSD eventually removed it. Fortunately, Konrad Schroder stepped in to repair NetBSD LFS and bring it back into a usable state [66].

5 Hardware support

5.1 New ports

NetBSD made a few steps towards total world domination, by adding support for a few more embedded device platforms:

- Atmark Techno Armadillo-9 [67] is a PC/104 form factor embedded device with the same size as a floppy disk. It features an ARM CPU, and a large set of I/O interfaces: USB2, Compact Flash, IDE, Ethernet and VGA. The evbarm port now supports it, thanks to Katsuomi Hamajima's work.
- Also from Atmark Techno, the Armadillo-210 [68] is an extremely small (barely the size of its connectors) ARM based machine, with VGA and Ethernet (supports Power over Ethernet). It has 32 MB of memory and 4 MB of flash. Again, brought to the evbarm port thanks to Katsuomi Hamajima.
- A brand new port, ews4800mips [69]. This brings NetBSD on NEC's EWS4800 workstations. The hard work has been done by UCHIYAMA Yasushi and Izumi Tsutsui.

Garrett d'Amore did a huge amount of work around the evbmips port, to support the following devices:

- the Alchemy Au1550 System-on-a-Chip (SoC) [70] featuring DDR controller, 2 Ethernet interfaces, 4 serial controllers

- Meshcube [71], a tiny cube with wireless Ethernet, RJ45 Ethernet, USB, 32 MB of flash and 64 MB of RAM
- Atheros AR5312, a SoC specialized for Wi-Fi appliances which is found in various wireless devices, such as Linksys WAP55AG 2.0 and WRT55AG, Meraki Mini [78], or Senao/Engenius 5354AP1 Aries2

More work on embedded ports:

- Steve Woodford added support for the Linksys NSLU2 NAS device to the evbarm port. The NSLU2 is an external hard disk with integrated Ethernet and USB.
- Shigeyuki Fukushima worked on the evbmips port to add support for OpenMicroServer 400 [72], yet another tiny server from a Japanese manufacturer who does not seem to have a page in English.
- NONAKA Kimihiro also hacked the evbarm port to add support for a similar Ethernet and USB enabled external hard disk from I-O DATA, the HDL-G400U.

And because embedded is not everything, we also had new desktop and server ports:

- Sanjay Lal imported initial support for Apple Powermac G5. For now it only works in 32-bit mode (using PowerPC 970 bridge mode), and it requires a serial console. It is able to boot to multi-user using a NFS root.
- And finally, Tim Rightnour improved the prep port to support two IBM RS/6000 models: IBM 7024-E20, 7025-F30, and 7025-F40, and the Motorola Powerstack E1.

5.2 AC97 modems

AC97 modems are a standardized set of software modems [73], also known as winmodems. A software modem is in fact a kind of sound card that connects to the phone line. The software has to perform the appropriate modulation of digital data into an analog signal suitable for being sent over telephone lines.

Hardware modems are seen as quite standard devices from the operating system. They are usually attached through a serial line (RS232, serial emulation communication over USB, over bluetooth), and they can be manipulated using the Hayes command set [74]. Software modems, on the other hand, need complex drivers that take care of all the modulation details.

The lack of drivers made winmodems quite unpopular on free OSes. It is interesting to note that they were not popular either in the Windows world, as buggy drivers turned them into unreliable and slow alternatives to hardware modems.

Jared D. McNeill made some work [75] to support AC97 modems. His contribution is split in two parts: First, improve kernel drivers to get access to the AC97 modem. Second, port to NetBSD the Linux `slmodemd` utility, which is the userland program that implements the soft modem.

5.3 VESA support

Jared also worked on VESA [76] support in NetBSD. VESA stands for Video Electronics Standards Association. It is a set of standards for video adapters which is better than plain old VGA. Most video boards implement VESA today.

The point in supporting VESA is that it allows using the console in graphic mode without having to get into the horrible details of how the video board works (that will be left to the X server). The console can therefore be used at higher resolution, and non ASCII character sets can be displayed.

And just for fun and because it was now possible, Jared D. McNeill added splash screen capability to the NetBSD kernel boot sequence [77].

5.4 MIDI

MIDI stands for Musical Instrument Digital Interface [79]. It is a communication protocol used to interface a computer and an electronic musical instrument.

The original NetBSD MIDI support, developed by Lennart Augustsson in 1998, and the USB MIDI support added by Takuya SHIOZAKI in 2001, served also as starting points for the MIDI support currently in FreeBSD and OpenBSD, but then saw little active improvement for a while, during which the code in the other projects diverged in order to address some bugs and functional concerns.

Chapman Flack adopted the NetBSD orphan MIDI code and fixed a lot of problems [80] on many aspects of MIDI support, from the sequencer API to hardware interrupt handling. Among other improvements, USB MIDI throughput problems that resulted in frequent dropped input data and drastically limited output rate have been corrected, with input drops eliminated under test conditions and sustained simultaneous output on multiple ports at the full MIDI 1.0 data rate. A default behavior for MIDI Active Sensing has been added that allows applications to detect communication interruptions with much simpler code than to parse and time out Active Sense messages explicitly, and that leads to reasonable behavior in pipelines of standard tools that have no knowledge of Active Sensing at all.

Chapman Flack also did a lot of code clean-up, and redesigned the MIDI framework to make it more machine-independent. The userland API has been clarified in a more detailed `midi(4)` [81] and an expanded `<sys/midiio.h>` [82] that for the first time supplies and documents a programming interface to the sequencer.

5.5 IEEE 1394

KIYOHARA Takashi imported FreeBSD's implementation of IEEE 1394 (also known as Apple's trade mark FireWire) [83]. That software allows NetBSD to use IEEE 1394 attached hard disks, and to use an IEEE 1394 link for IP communications.

That import also substantially improves the stability of IEEE 1394 on NetBSD.

5.6 Miscellaneous device driver work

Here is a quick summary of the steady work on device driver support:

- Network controllers
  - Damien Bergamini, FUKAUMI Naoki, and Matthias Drochner worked on a driver for Ralink PCI/Cardbus/USB WLAN adapters.
  - Rui Paulo worked on support for ASIX AX88140A and AX88141 Ethernet controllers.
  - Juan Romero Pardines imported OpenBSD's driver for Realtek 8139/8201L Ethernet interfaces.
  - Chuck Silvers imported OpenBSD's driver for NVIDIA nForce Ethernet controller
  - Garrett d'Amore imported HAL 0.9.17.2 from Atheros, to support new SoCs such as the AR531x
  - Rui Paulo added support for RT2661-based wireless interfaces
  - Tohru Nishimura developed a driver for Micrel KSZ8842 and KSZ8841 Ethernet controllers
  - Christos Zoulas imported David Boggs' driver [84] for SBE (previously known as LMC) Wide Area Network (WAN) cards [85]. Now one can build a NetBSD WAN router.
  - David Young also added support for GCT Semiconductor GRF5101 transceiver/synthesizer.
- Audio controllers
  - Juan Romero Pardines upgraded the `auich` audio driver to support ICH7 and Intel 6300ESB audio controllers.

  - TAMURA Kent improved the `azalia` driver to bring S/PDIF [88] support.
  - Chapman Flack upgraded the `eap` audio driver to use `txrdy` interrupts for MIDI, and added the `es1373` register definitions, which could be a start for S/PDIF support
- Disk controllers
  - Manuel Bouyer imported Joerg Sonnenberger's work on the driver for ServerWorks K2 SATA controller from OpenBSD.
- Human Interface Devices (HID)
  - KIYOHARA Takashi added support for the touch-panel and LCD screen of PERSONA SH3 machines.
  - Takeshi Nakayama added support for the the Sharp Telios LCD screen and Battery unit.
  - Christos Zoulas also imported Johan Wallen's driver for Apple's 15" powerbook mouse.
  - Peter Postma adopted the Jornada 720 machine-dependent code and worked on keyboard and power management.
- Serial communication and USB
  - Nick Hudson added a driver for Cypress micro controller based serial devices
  - Lennart Augustsson imported OpenBSD's driver for accessing an iPAQ through USB.
- Power management and hardware monitoring
  - Jared D. McNeill imported support for Intel power management technology SpeedStep PIIX4, from FreeBSD. On the AMD front, Juan Romero Pardines integrated Martin Vegiard's work on AMD PowerNow, and imported the Cool'n'Quiet driver from OpenBSD.
  - Juan Romero Pardines also imported OpenBSD's driver for ITE's IT8705F, IT8712F and SiS' SiS950 hardware monitors (these devices report temperature, fan speed, and various other useful information).
  - David Young added a driver for AMD Geode SC1100 micro controller's watchdog timer.
  - Jeff Rizzo imported OpenBSD's driver for Dallas Semiconductor 1-wire bus [87], General Purpose I/O (GPIO) and temperature sensors using that bus.
- Video devices
  - Steve Woodford developed a driver for the Topfield TF5000PVR range of digital video recorders [86].

6 Binary compatibility

NetBSD has the capability of running binaries from other OSes that are built for the same processor [89]. This works with very little overhead, by emulating system calls. When the foreign binary makes a system call, the NetBSD kernel behaves like the foreign OS kernel would have. The foreign binary gets appropriate answers from the kernel, and it just works.

With the help of Nicolas Joly, who made a lot of testing with Linux binaries, I improved a lot the Linux binary compatibility for machines running NetBSD/amd64. The NetBSD kernel now emulates enough of the Linux Native POSIX Thread Library (NPTL) [90] kernel code to masquerade as a 2.6 series kernel to Linux processes.

Unfortunately, other NetBSD ports lack the machine dependent code for emulating the NPTL, and are stuck at emulating the 2.4 Linux kernel.

I also contributed the 32-bit Linux binary compatibility for NetBSD/amd64, which is also known as the `COMPAT_LINUX32` kernel option. Intensive tests made by Nicolas Joly suggest that it has reached a fair level of usability.

7 System Packages

System Packages, or **syspkg**, is a new infrastructure for packaging the base system in fine-grained packages. Once **syspkg** will be fully integrated in the build and installation processes, an administrator will be able to install a NetBSD system that contains **dhclient** but not **dhcpd**, for instance.

syspkg is still under development. Alan Barrett made some progress, by adding the ability to generate **syspkg** **.tgz** files from the NetBSD build infrastructure. Nothing can be done yet with the generated files. The next step is to give tools such as **pkg_add** [91] the ability to install and upgrade **syspkg**.

For more information on how to generate **syspkg**, see the NetBSD build documentation [92].

8 Security

8.1 News from Security Officers

Since the beginning of 2006 the NetBSD Security Officer team has released 22 security advisories for NetBSD [93]. The advisories cover both issues found in third party software included in the base NetBSD operating system (e.g. BIND, OpenSSL etc.) and issues found in the kernel and user land. Included in this was SA2006-019 [94] which documented an issue discovered by two NetBSD developers that was found to impact all BSDs (NetBSD, OpenBSD, FreeBSD and DragonFly BSD). The severity of issues discovered ranged from denial of service to privilege escalation attacks.

8.2 News from pkgsrc security team

The pkgsrc Security Team monitors vulnerabilities found in software included as a part of pkgsrc. In May 2005 the team started using RT [95] to track issues to ensure that vulnerable packages are identified and updated promptly. In addition to this, the security team also tries to ensure that pull-ups for any security fixes are applied to the pkgsrc stable branches.

The file used for tracking package vulnerabilities [96] currently has 2183 active entries. At the start of 2006 it had 1606 entries, this represents 577 entries that have been added since.

For more information on the pkgsrc Security team see the presentations from Adrian Portelli at pkgsrcCon 2005 [97] and pkgsrcCon 2006 [27].

8.3 Further integration of PaX

The goal of the PaX project [98] is to provide a set of defense mechanisms against attacks that rely on writing in a process address space. Stack buffer overflows are the best example of such attacks. The main PaX idea is to prevent process memory to be writable and executable at the same time. That way, an attacker that uploads executable code through a buffer overflow will have difficulties to execute it. There are also other tricks, such as Address Space Layout Randomization (ASLR), which are designed to make attacks less reliable.

PaX was originally developed for Linux, but as usual, good ideas spread to other OSes. Elad Efrat updated NetBSD's **mprotect(2)** [99] to enforce **W^X** (Write or eXecute, but not both) policies.

There are always odd programs that need to execute code they produce at run-time. The Java Just-In-Time (JIT) compiler is an example. The paxctl [100] tool can be used to enable or disable the **W^X** policy on a per-program basis.

8.4 Kernel authorization framework

Kernel authorization (**kauth** for short) is work behind the scene also done by Elad Efrat. It produces few visible features to users, but it is the foundation for very interesting future work on security.

Authorization mechanisms have always been very simple in traditional Unix systems. Apart from file system permissions, the only kind of authorization checks that existed was "is the process running under UID 0?"

That meant that the root user had all the privileges, while other "unprivileged" users had none. Root's awesome power was loosely delegated through set-UID programs, which exposed a lot of unexpected security bugs for exploitation.

kauth's goal is to make kernel authorization much more flexible, so that fine-grained security policies could be enforced. NetBSD **kauth** is a clean room implementation based on Apple's **kauth** [101] (the original code from Apple could not be copied as its licensing was too restrictive for inclusion in the BSD-licensed NetBSD kernel).

For now, NetBSD **kauth** is just used to re-implement the traditional Unix security model, but it allows future development of alternative security models. Capabilities [102] are an example of probable future work.

Elad also presents an article on NetBSD security improvements [103] at EuroBSDCon 2006, which includes **kauth** coverage.

8.5 File association kernel programming interface

File association kernel programming interface (**fileassoc** for short)[104] is another work behind the scenes done by Elad Efrat and Brett Lymn. **fileassoc** is a Kernel Programming Interface (KPI) used to store meta-data associated to a file. The first usage of **fileassoc** is to store trusted executable signatures for the NetBSD subsystem responsible of verifying executable integrity (also known as **veriexec**) [105].

Fileassoc can also be used to store any meta-data, for instance extended file system attributes such as Access Control Lists (ACL).

8.6 Paper on NetBSD security

It would be too bad to close the chapter of security without a word on Elad Efrat's paper on NetBSD security [106], published at securityfocus.com. This excellent article tells a lot about OS security state of the art and alternative that are available to NetBSD.

9 More work behind the scenes

Jason Thorpe added experimental support for storing extended file system attribute on FFSv1 file system. While FFSv2 has provisions to store file system extended attributes, there was no place to store them on a FFSv1 file system. It is now possible to store them in plain files.

Christos Zoulas merged the duplicated code between **libc** and kernel. There is now a **src/common** directory in the source tree that holds the shared code. That change was the opportunity to use the same **zlib** (compression algorithms used everywhere in the system) between kernel and userland. It was also the opportunity to make **libc** compatibility code build optional, so that people without the need of backward compatibility can build a smaller **libc**. There is a minor side effect: it's not possible anymore to just check out **src/sys** and build a kernel: **src/common** is now required as well.

On February 2005, I switched our IPsec key exchange daemon (known as **racoon**) from the orignal KAME [15] implementation to IPsec-tools [107], a fork made initially for Linux that was more reactive to features addition. IPsec-tools CVS was hosted at SourceForge, and it was decided to move it to the NetBSD CVS server. The reasons for the move were a more

reliable and more secure CVS, automatic builds, and automatic Coverity [108] scans. While IPsec-tools HEAD is merged in NetBSD-current, it still remains as a stand alone package for Linux, FreeBSD and Darwin, so that move changes nothing for IPsec-tools and NetBSD users (except that NetBSD-current will now always include latest IPsec-tools code).

And finally, Darren Reed, Nick Hudson, and Christos Zoulas completed the work required so that `ktrace` and `kdump` get the ability to report Light-Weight Process (LWP) information.

10 Third party software

NetBSD ships with various third party software integrated [109]. Here is the current status as of October 2006.

10.1 Removed software

- Sushi [110], the curses-based administration tool. Obviously, it never really found its users, and it was difficult to maintain.
- Kerberos IV [111] has been removed, in favor of Kerberos V [112]. Both versions were maintained in-tree for some time, but it is now assumed that all Kerberos IV users have migrated to Kerberos V.
- Vinum [113] was removed because nobody was interested enough in it to actually maintain the code. This is probably because NetBSD already had RAIDframe, which provides a similar set of features (except the volume manager part, for which NetBSD is left without any equivalent).
- Sendmail [114] suffered a new security issue, but nobody was ready to maintain it. It was therefore decided to remove it, leaving Postfix [115] as the only mail software in the base system (previous NetBSD releases contained both Sendmail and Postfix). Fortunately for Sendmail fans like me, Sendmail remains quite easy to install through the NetBSD package system.

10.2 Software resurrected from the dead

In January 2002, Caldera released [116] the source for ancient Unix versions up to AT&T UNIX version 7 [117]. All the critical pieces of Unix have been re-implemented as free software in *BSD for a long time, but there were a few nifty utilities left that Perry E. Metzger resurrected from the dead and integrated in NetBSD:

- `deroff` [118], a tool to remove roff constructs from files
- `spell` [119], a spell-checker
- `ching` [120], the Unix oracle, which answers any of your questions (that one stands in `/usr/games`)

10.3 Third party software upgrades

Many contributors did some work on upgrading third party programs bundled with NetBSD. The third party software distributed with NetBSD page [109] gives details about individual programs.

- `pppd` 2.4.4
- NTP 4.2.2p2
- GCC 4.1 and GCC 3.3.6 (Some ports use 4.1, others still use 3.3.6)
- binutils 2.16.1
- BIND 9.3.2
- `file` 4/16
- am-utils 6.1.3
- CVS 1.11.22
- OpenSSL 0.9.8a
- OpenSSH 4.3
- OpenPAM 20050616

- `groff` 1.19.2
- IPFilter: 4.1.13
- PacketFilter (PF): from OpenBSD 3.7
- Postfix 2.3.2
- `zlib` 1.2.3
- `wpa_supplicant` / `hostapd` 0.4.9

And last but not least, the X.org status. NetBSD is committed to switch to X.org [121], since most, if not all, open source OSes did that move. It is now clear that most development will come from X.org and not from the XFree86 project [122].

X.org 7.0 code has been imported in NetBSD CVS by Michael Lorenz. It seems no status has been published, but according to Michael, it it possible to build and run X.org on NetBSD. Performance is a bit disappointing, as `x11perf` shows a 10% drop versus XFree86. On the other hand, X.org memory usage is a bit better.

The next step in X.org migration is to integrate the build in the NetBSD build machinery, and start using it as the default X implementation for ports where it has some interest. It is worth noting that a few NetBSD ports with very odd X servers may have no point into moving from XFree86 to X.org, because the X.org implementation will offer no benefit, and may not be really maintained. It is therefore possible that NetBSD retains both X implementation for some time.

Conclusions

Whatever has been said in the press, NetBSD is still a very active project, and giving an idea of a whole year of activity is not a straightforward task. Of course this paper left some changes unmentioned. Curious readers might want to take a peek at the raw NetBSD change log [123] in order to get an exhaustive list.

One recurrent issue when trying to collate a list of significant changes is that too often, developers tend to neglect telling the world what they did and why it is so cool. The change log is full of lines which are meaningless for the average user, and even sometime for the average NetBSD developer. In my opinion, we obviously have some room for improvement here. Giving more publicity about what is done in NetBSD is one way to stop the rumors that it got irrelevant, dead, or that it is only useful for running toasters.

Acknowledgments

I would like to thank a few fellow NetBSD developers for reviewing this paper, and taking time to tell me about the thing they had been working on: Alan Barrett, Christian Biere, Pavel Cahyna, Elad Efrat, Havard Eidnes, Chapman Flack, M.J. Fleming, Liam J. Foy, Iain Hibbert, Bang Jun-Young, KIYOHARA Takashi, Thomas Klausner Sanjay Lal, Michael Lorenz, Jared D. McNeil, Greg Oster, Adrian Portelli, Jeremy C. Reed, Antoine Reilles, Tim Rightnour, Lubomir Sedlacik, Thor Lancelot Simon, Joerg Sonnenberger, Steve Woodford, and Christos Zoulas.

References

[1] Anonymous coward, *BSD is dying, Slashdot web site
`http://bsd.slashdot.org/comments.pl?sid=189013&cid=15569908`

[2] Charles M. Hannum, The future of NetBSD, `netbsd-users@netbsd.org` mailing list
`http://mail-index.netbsd.org/netbsd-users/2006/08/30/0016.html`

[3] Charles M. Hannum, Confessions of a Recovering NetBSD Zealot, OnLAMP web site
`http://www.onlamp.com/pub/a/bsd/2006/09/14/netbsd_future.html`

[4] The NetBSD project, `source-change@netbsd.org` mailing list
`http://mail-index.netbsd.org/source-changes`

[5] Alistair G. Crooks, Organizational Changes to the NetBSD Project, `netbsd-users@netbsd.org` mailing list
`http://mail-index.netbsd.org/netbsd-users/2006/09/01/0015.html`

[6] The NetBSD foundation, NetBSD Foundation Membership Agreement, NetBSD web site
http://www.netbsd.org/developers/agreement.txt

[7] The NetBSD foundation, The NetBSD Bugathon: Reloaded, NetBSD web site
http://www.netbsd.org/hackathon/

[8] The NetBSD foundation, GNATS Bug Database Summary, NetBSD web site
http://www.netbsd.org/Gnats/

[9] Elad Efrat, NetBSD Bugathon: Not quite dead, netbsd-anounce@netbsd.org mailing list
http://mail-index.netbsd.org/netbsd-announce/2006/09/25/0000.html

[10] Elad Efrat, NetBSD Bugathon #2, netbsd-users@netbsd.org mailing list
http://mail-index.netbsd.org/netbsd-users/2006/10/09/0002.html

[11] Hubert Feyrer, NetBSD open Problem Reports, NetBSD web site
http://www.netbsd.org/~hubertf/open-prs.gif

[12] Google, Google Summer of Code, Google web site
http://code.google.com/soc/

[13] Hubert Feyrer, Announcing NetBSD and the Google "Summer of Code" Projects 2006, NetBSD press releases
http://www.netbsd.org/Foundation/press/soc2006.html

[14] K. Ramakrishnan, S. Floyd, D. Black, The Addition of Explicit Congestion Notification (ECN) to IP, RFC 3168
http://www.rfc-editor.org/rfc/rfc3168.txt

[15] The KAME project
http://www.kame.net/

[16] The NetBSD project, fast_ipsec(4) man page
http://netbsd.gw.com/cgi-bin/man-cgi?fast_ipsec++NetBSD-current

[17] The NetBSD project, pkgsrc: The NetBSD Packages Collection, pkgsrc web site
http://www.pkgsrc.org

[18] The NetBSD project, mbuf(9) man page
http://netbsd.gw.com/cgi-bin/man-cgi?mbuf++NetBSD-current

[19] Jorg Sonnenberger, pkg_install, EuroBSDCon 2006
http://www.eurobsdcon.org/talks-sonnenberger.php

[20] pkgsrcCon web site
http://www.pkgsrccon.org/2006/

[21] Stoned Elipot, System administration with pkgsrc, pkgsrcCon 2006
http://www.pkgsrccon.org/2006/slides/mpkg.pdf

[22] Joerg Sonnenberger, pkgsrc on DragonFly – or Fighting the Windmills, pkgsrcCon 2006
http://www.pkgsrccon.org/2006/slides/pkgsrc-on-df/index.html

[23] Roland Illig, pkglint: Static Analyzer For Pkgsrc, pkgsrcCon 2006
http://www.pkgsrccon.org/2006/slides/presentations/pkglint.html

[24] Roland Illig, Why Pkgsrc Sucks, pkgsrcCon 2006
http://www.pkgsrccon.org/2006/slides/presentations/why-pkgsrc-sucks.html

[25] Emile Heitor, pkg_select – So Many Packages, So Few Columns, pkgsrcCon 2006
http://www.pkgsrccon.org/2006/slides/pkg_select.pdf

[26] Thomas Klausner, Roundtable Discussion: Updating Packages, pkgsrcCon 2006
http://www.pkgsrccon.org/2006/slides/updates.html

[27] Adrian Portelli, pkgsrc security one year on..., pkgsrcCon 2006
http://www.pkgsrccon.org/2006/slides/pkgsrc-Security-06.html

[28] Dieter Baron, Thomas Klausner, pkg_install Rewrite, pkgsrcCon 2006
http://www.pkgsrccon.org/2006/slides/pkg_install.html

[29] Johnny Lam, Roadmap for Development, pkgsrcCon 2006
http://www.pkgsrccon.org/2006/slides/roadmap.html

[30] University of Cambridge computer laboratory, the Xen virtual machine monitor, Xen web site
http://www.cl.cam.ac.uk/research/srg/netos/xen/

[31] Emmanuel Dreyfus, Remote user access VPN with IPsec, EuroBSDCon 2005
http://pubz.hcpnet.net/rasvpn.pdf

[32] University of Cambridge computer laboratory, performances, Xen web site
http://www.cl.cam.ac.uk/research/srg/netos/xen/performance.html

[33] Martti Kuparinen, Xen Disk I/O Benchmarking: NetBSD dom0 vs Linux dom0
http://users.piuha.net/martti/comp/xendom0/xendom0.html

[34] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, E. Zeidner, Internet Small Computer Systems Interface (iSCSI), RFC 3720
http://www.rfc-editor.org/rfc/rfc3720.txt

[35] Alistair G. Crooks, NetBSD iSCSI HOWTOs, current-users@netbsd.org mailing list
http://mail-index.netbsd.org/current-users/2006/02/21/0018.html

[36] Alistair G. Crooks, iSCSI - beyond the hype, EuroBSDCon 2006
http://www.eurobsdcon.org/talks-crooks.php

[37] Luke Mewburn, Matthew Green, build.sh: cross-building NetBSD, BSDCon 2003
http://www.mewburn.net/luke/papers/build.sh.pdf

[38] The NetBSD project, Summary of daily snapshot builds, NetBSD release engineering web site
http://releng.netbsd.org/cgi-bin/builds.cgi

[39] The NetBSD project, makefs(8) man page
http://netbsd.gw.com/cgi-bin/man-cgi?makefs++NetBSD-current

[40] Wikipedia, Wi-Fi Protected Access
http://en.wikipedia.org/wiki/Wi-Fi_Protected_Access

[41] Wikipedia, Wired Equivalent Privacy
http://en.wikipedia.org/wiki/Wired_Equivalent_Privacy

[42] S. Fluhrer, I. Mantin, A. Shamir, Weaknesses in the Key Scheduling Algorithm of RC4, Selected Areas in Cryptography 2001: pp1-24.

[43] Steve Woodford, WPA support, current-users@netbsd.org mailing list
http://mail-index.netbsd.org/current-users/2005/10/01/0014.html

[44] Jouni Malinen, hostapd(8) man page
http://netbsd.gw.com/cgi-bin/man-cgi?hostapd++NetBSD-current

[45] Jouni Malinen, wpa_supplicant(8) man page
http://netbsd.gw.com/cgi-bin/man-cgi?wpa_supplicant++NetBSD-current

[46] Jouni Malinen, Host AP driver for Intersil Prism2/2.5/3, hostapd, and WPA Supplicant
http://hostap.epitest.fi/

[47] Wikipedia, Bluetooth
http://en.wikipedia.org/wiki/Bluetooth

[48] Iain Hibbert, Bluetooth, tech-net@netbsd.org mailing list
http://mail-index.netbsd.org/tech-net/2006/05/23/0000.html

[49] The NetBSD Wiki, Bluetooth
http://wiki.netbsd.se/index.php/bluetooth

[50] Wikipedia, Universal Disk Format
http://en.wikipedia.org/wiki/Universal_Disk_Format

[51] Reinoud Zandijk, HEADS UP: UDF file system added to NetBSD source tree, tech-kern@netbsd.org mailing list
http://mail-index.netbsd.org/current-users/2006/02/02/0027.html

[52] Wikipedia, Common Address Redundancy Protocol
http://en.wikipedia.org/wiki/Common_Address_Redundancy_Protocol

[53] R. Hinden, Virtual Router Redundancy Protocol (VRRP), RFC 3768
http://www.ietf.org/rfc/rfc3768.txt

[54] T. Li, B. Cole, P. Morton, D. Li, Cisco Hot Standby Router Protocol (HSRP), RFC 2281
http://www.ietf.org/rfc/rfc2281.txt

[55] Liam J. Foy, CARP Committed (correctly presented), current-users@netbsd.org mailing list
http://mail-index.netbsd.org/current-users/2006/05/18/0005.html

[56] Wikipedia, LACP
http://en.wikipedia.org/wiki/LACP

[57] YAMAMOTO Takashi, agr(4) man page
http://netbsd.gw.com/cgi-bin/man-cgi?agr++NetBSD-current

[58] Microsoft corporation, NDIS - Network Driver Interface Specification
http://www.microsoft.com/whdc/device/network/ndis/default.mspx

[59] David Chisnall, Project Evil: Windows network drivers on FreeBSD
http://www.pingwales.co.uk/2005/07/15/Project-Evil.html

[60] NdisWrapper for Linux at SourceForge
http://ndiswrapper.sourceforge.net/

[61] Alan Ritter, NDIS on NetBSD
http://netbsd-soc.sourceforge.net/projects/ndis/

[62] Bill Paul, ndiscvt(8) man page
http://netbsd.gw.com/cgi-bin/man-cgi?ndiscvt++NetBSD-current

[63] Julio M. Merino Vidal, HEADS UP: tmpfs added, tech-kern@netbsd.org mailing list
http://mail-index.netbsd.org/tech-kern/2005/09/10/0004.html

[64] Niklas Hallqvist, Tobias Weingartner, scan_ffs(8) man page
http://netbsd.gw.com/cgi-bin/man-cgi/man?scan_ffs+8+NetBSD-current

[65] Wikipedia, Log-structured File System
http://en.wikipedia.org/wiki/Log-structured_file_system

[66] Konrad Schröder, Log-structured File System for NetBSD
http://www.hhhh.org/perseant/lfs.html

[67] Atmark techno, Armadillo-9
http://www.atmark-techno.com/en/products/armadillo/a9/

[68] Atmark techno, Armadillo-210
http://www.atmark-techno.com/en/products/armadillo/a210/

[69] The NetBSD project, ews4800mips port page, NetBSD web site
http://www.netbsd.org/Ports/ews4800mips/

[70] RMI, Alchemy Au1550 Processor
http://www.razamicroelectronics.com/products_alchemy/au1550_overview.htm

[71] meshcube.org, The meshing computing website
http://www.meshcube.org/index_e.html

[72] Plat'Home, OpenMicroServer 400
http://www.plathome.co.jp/products/oms400/

[73] Wikipedia, softmodem
http://en.wikipedia.org/wiki/Winmodem

[74] Wikipedia, Hayes command set
http://en.wikipedia.org/wiki/Hayes_command_set

[75] Jared D. McNeill, Preliminary AC'97 modem support in auich(4), current-users@netbsd.org mailing list
http://mail-index.netbsd.org/current-users/2005/04/07/0022.html

[76] Wikipedia, VESA
http://en.wikipedia.org/wiki/VESA

[77] Jared D. McNeill, VESA framebuffer console in NetBSD
http://www.invisible.ca/space/vesa-framebuffer-console-in-netbsd

[78] Meraki Networks, Meraki Mini
http://www.meraki.net/mini.html

[79] Wikipedia, MIDI
http://en.wikipedia.org/wiki/MIDI

[80] Chapman Flack, RFC: merge chap-midi branch, tech-kern@netbsd.org mailing list
http://mail-index.netbsd.org/tech-kern/2006/06/19/0003.html

[81] Lennart Augustsson, Chapman Flack, midi(4) man page
http://netbsd.gw.com/cgi-bin/man-cgi?midi+4+NetBSD-current

[82] Lennart Augustsson, Chapman Flack, <sys/midiio.h> header file
http://cvsweb.netbsd.org/bsdweb.cgi/src/sys/sys/midiio.h?rev=1.13.4.1

[83] Wikipedia, firewire
http://en.wikipedia.org/wiki/Firewire

[84] David Boggs, lmc(4) man page
http://netbsd.gw.com/cgi-bin/man-cgi?lmc+4+NetBSD-current

[85] SBE Inc, Products - WAN
http://www.sbei.com/index.php/products/wan/

[86] Topfield web site
http://www.topfield.co.kr/

[87] Wikipedia, 1-Wire
http://en.wikipedia.org/wiki/1-Wire

[88] Wikipedia, S/PDIF
http://en.wikipedia.org/wiki/S/PDIF

[89] The NetBSD project, NetBSD binary emulation
http://www.netbsd.org/Documentation/compat.html

[90] Ulrich Drepper, Ingo Molnar, The Native POSIX Thread Library for Linux
http://people.redhat.com/drepper/nptl-design.pdf

[91] Jordan Hubbard, John Kohl, Hubert Feyrer, Thomas Klausner, pkg_add(1) man page
http://netbsd.gw.com/cgi-bin/man-cgi?pkg_add++NetBSD-current

[92] Luke Mewburn, Todd Vierling, Procedure for building NetBSD from source
ftp://ftp.fr.netbsd.org/pub/NetBSD/NetBSD-current/src/BUILDING

[93] The NetBSD project, Security and NetBSD
http://www.netbsd.org/Security/

[94] NetBSD Security Officer team, NetBSD Security Advisory 2006-019
ftp://ftp.netbsd.org/pub/NetBSD/security/advisories/NetBSD-SA2006-019.txt.asc

[95] Best Pratical Solutions LLC, Request Tracker
http://www.bestpractical.com/rt/

[96] The NetBSD project, packages vulnerability file
ftp://ftp.netbsd.org/pub/NetBSD/packages/distfiles/pkg-vulnerabilities

[97] Adrian Portelli, pkgsrc security, pkgsrcCon 2005
http://www.pkgsrccon.org/2005/slides/adrianp/pkgsrc-Security.html

[98] The PaX project
http://pax.grsecurity.net/docs/pax.txt

[99] The NetBSD project, mprotect(2) man page
http://netbsd.gw.com/cgi-bin/man-cgi?mprotect+2+NetBSD-current

[100] Elad Efrat, paxctl(1) man page
http://netbsd.gw.com/cgi-bin/man-cgi?paxctl+1+NetBSD-current

[101] Apple computers Inc, Technical Note TN2127 Kernel Authorization
http://developer.apple.com/technotes/tn2005/tn2127.html

[102] Wikipedia, capability-based security
http://en.wikipedia.org/wiki/Capabilities

[103] Elad Efrat, NetBSD Security Enhancements, EuroBSDCon 2006
http://www.eurobsdcon.org/talks-efrat.php

[104] Elad Efrat, Brett Lymn, fileassoc(9) man page
http://netbsd.gw.com/cgi-bin/man-cgi?fileassoc++NetBSD-current

[105] Brett Lymn, NetBSD verified executables
http://www.users.on.net/~blymn/veriexec/

[106] Elad Efrat, Recent Security Enhancements in NetBSD
http://www.securityfocus.com/infocus/1878

[107] The IPsec-tools project
http://ipsec-tools.sf.net

[108] Coverity, automated error precention and source code analysis
http://www.coverity.com/

[109] The NetBSD project, Third party software distributed with NetBSD
http://netbsd.org/Documentation/software/3rdparty/

[110] Tim Rightnour, Sushi - an extensible human interface for NetBSD, BSDCon 2002
http://db.usenix.org/events/bsdcon02/full_papers/rightnour/rightnour.pdf

[111] Kungliga Tekniska Högskolan (KTH) Kerberos page
http://www.pdc.kth.se/kth-krb/

[112] Heimdal Kerberos page
http://www.pdc.kth.se/heimdal/

[113] Greg Lehey, The vinum volume manager
http://www.vinumvm.org/

[114] the Sendmail consortium, Sendmail home page
http://www.sendmail.org/

[115] the Postfix project, Postfix home page
http://www.postfix.org

[116] Bill Broderick, Caldera license
http://www.tuhs.org/Archive/Caldera-license.pdf

[117] Unix Archive Sites
http://www.tuhs.org/archive_sites.html

[118] deroff(1) man page
http://netbsd.gw.com/cgi-bin/man-cgi?deroff++NetBSD-current

[119] spell(1) man page
http://netbsd.gw.com/cgi-bin/man-cgi?spell++NetBSD-current

[120] ching(6) man page
http://netbsd.gw.com/cgi-bin/man-cgi?ching++NetBSD-current

[121] The X.org foundation
http://www.x.org/

[122] The XFree86 project
http://www.xfree86.org/

[123] The NetBSD project, Recent Changes and News
http://www.netbsd.org/Changes/

Recent Security Enhancements in NetBSD

Elad Efrat <elad@NetBSD.org>

September 2006

Abstract

Over the years, NetBSD obtained the position of the BSD focusing on portability. While it is true that NetBSD offers an easily portable operating system, care is also given to other areas, such as security. This paper presents the NetBSD philosophy of security, design decisions, and currently offered security features. Finally, some of the current and future research will be revealed.

1. Introduction

Running on almost twenty different architectures, and easily portable to others, NetBSD gained its reputation as the most portable operating system on the planet. While that may indicate high quality code, the ever demanding networked world cares about more than just that. Over the past year, NetBSD evolved quite a bit in various areas; this paper, however, will focus on the aspect relating to security.

This paper was written and structured to present a full overview of the recent security enhancements in NetBSD in an easily readable and balanced form that will satisfy new, intermediate, and experienced users. References were sprinkled across the text to provide more information to those who want the gory details, while preserving the continuity.

Section 2 will present the bigger picture of security in NetBSD: how NetBSD perceives security, the design decisions of NetBSD software in general and the security infrastructure and features more specifically. Section 3 will present a detailed overview of the recent enhancements in the security infrastructure and features of NetBSD including, where relevant, details about the design, implementation, and possible future development. Section 4 will present current security-related research and development in NetBSD, and section 5 will discuss how the described enhancements work together to provide a more secure platform. Section 6 concludes the paper, and summarizes availability of discussed features.

2. The Tao of NetBSD Security

We are all familiar with the mantra that *security is a process, not a product.* When regarding software development, specifically operating systems, it should be part of the design, from the ground up. As the descendent of an operating system over 20 years old, NetBSD carries a security model designed and implemented with different threats in mind; the Internet was smaller, more naive, and less popular.

The following sections will provide background to the approach taken to enhance the security of the NetBSD operating system: the considerations, existing approaches, and case-studies.

2.1 Considerations

When approaching to enhance the security of NetBSD, two of the most important leading principles were maintaining compatibility and interoperability[1]. Presenting changes that would dramatically impact the user-base was out of question, and careful planning had to be done. In addition, any change to underlying back-ends had to be well thought-out so it maintains existing semantics without enforcing them during design stage.

2.2 Security Approaches

Operating system security is nothing new, and NetBSD is not the first to address the issue. In designing software – and security software in particular, it is mandatory to learn from the experience of previous work. Below are some common approaches to security and real-world case-studies.

2.2.1 Code Auditing

Code auditing addresses security issues by looking for programming glitches in the source code of the program, often with the assistant of automated tools[2]. Normally the work of vulnerability researchers, when done proactively by the programmers themselves, has the potential of locating and fixing bugs with security implications before anyone else finds and exploits them.

While some would argue that striving to produce bug-free code is the *one true way* of achieving security, this view is a fallacy for two main reasons. The first is that security issues are not always the result of programming errors; while code auditing tries to ensure no software bugs will be maliciously exploited because said bugs would simply not exist, it alone ignores other important aspects, such as configuration errors and user behavior policies.

The second reason is that it is not possible to write bug-free code[3]. Over the past decade, the awareness to writing secure code rose significantly; automated tools evolved, allowing easy pinpointing of software bugs; open-source software is available for the review of thousands – if not millions – of people; yet, we still see new security vulnerabilities on a daily basis. Some of those, ironically, are of the exact same type that affected us ten or twenty years ago[4].

2.2.2 Exploit Mitigation

The unorthodox approach of exploit mitigation addresses bugs from the opposite direction of code auditing: instead of looking for them in and removing them from software to make it more secure, it *adds bugs* to the exploit code to prevent it from working. While that may be over-simplified, the purpose of exploit mitigation technologies is to interfere with the inner-workings of the exploit, eliminating the – often unusual – conditions that make it work.

On one hand, some would claim that exploit mitigation discourages developers from writing secure code and vendors from quickly responding to security incidents: they

---

[1] Two other leading principles – not impacting the system performance and an easy user interface, will not be discussed in this paper.

[2] Coverity, for example, offered its services to various open-source projects, including NetBSD, for free. See http://scan.coverity.com

[3] http://www.cs.columbia.edu/~smb/papers/acm-predict.pdf

[4] http://www.cert.org/homeusers/buffer_overflow.html

know there's a *safety net* guarding them, and so they pay less attention to security when writing code, or taking their time coming up with fixes for security issues.

On the other hand, however, this is also where exploit mitigation technologies excel: they introduce the concept of preventing the successful exploitation of security vulnerabilities, even before a fix is available. Moreover, they prevent entire classes of bugs, and don't require constant updating.

2.2.3 Architectural Integration

So far, the previous two approaches assume the cause of a security breach is a bug in the code that is being exploited. The first approach tries to eliminate such bugs, and the second one tries to make it next to impossible to successfully exploit them. However, some environments require more than just that – for example, the ability to define detailed usage policies and associate them with entities on the system became a mandatory part of many security policies. In our context, we can relate that to the Unix permissions model; simply put, due to the coarse separation between a normal user and a superuser, it cannot be used to express many security policies as detailed as may be required.

That led to the research of various modern security models, of which most recognized ones are fine-grained[5] discretionary access controls (DACs) and mandatory access controls (MACs). To put things simple, DACs focus on the data owner's ability to specify who can use it and for what; MACs focus on a mandatory policy that affects everyone.

These systems allow an administrator – and where applicable, the users – to specify fine-grained policies; effectively, this means that a user or a program can be made to work with the minimal amount of privileges required for their operation (which, as implied above, cannot be done with the traditional Unix security model), resulting in damage containment in case of compromise or otherwise minimized impact from security vulnerabilities.

2.2.4 Layered security

To itself, layered security[6] is not a single approach. Where any of the previous three took a different route, the layered security approach suggests that maximized security can only be achieved by combining efforts on all fronts: code auditing is important, but does not come in place of useful exploit mitigation technologies; and architectural integration, of course, has little to do with any of them.

Although the above may sound obvious, it is not too often when you see an operating system that puts an emphasis on all three aspects; it will usually be the case that only one of the approaches is fully practiced. Following are some short case-studies that illustrate the importance of each approach by using real-world examples.

2.2.5 Case Studies

Shortly after splitting from NetBSD in 1995, OpenBSD became widely known for its unique – at the time – approach to security: proactive code auditing. Instead of

[5] I emphasize *fine-grained* because DACs already exist on Unix; however, as noted, they are too coarse.

[6] Also known as *Defense in Depth*.

retroactively responding to security issues, OpenBSD developers performed thorough code auditing sessions, *sweeping* for bugs. This act proved itself more than once, after vulnerabilities found in other operating systems were *already fixed*[7] in OpenBSD.

This, however, did not last too long. In 2002, winds of change blew through the OpenBSD mindset: the long standing fort of code auditing fell, adopting exploit mitigation technologies to its lap[8]. While the reasons behind the move were not published, some speculate that it was the release of an exploit allowing full system compromise of OpenBSD's default configuration[9] that finally proved that even a group of dedicated programmers cannot find all bugs; at least not first.

Said exploit mitigation technologies made their public debut around 1996, with the appearance of the Openwall[10] project, and later evolved dramatically by the PaX[11] project in 2000. Research done in both projects formed the basis of today's exploit mitigation technologies. Another commonality of the two was that they offered an implementation based on Linux – which only makes one wonder why it was OpenBSD that was the first to officially adopt these technologies.

Linux, however, took a different direction. First with the addition of POSIX.1e[12] capabilities in 1999, fine-grained discretionary access controls, later with SELinux[13], an implementation of mandatory access controls, and finally with the introduction of the Linux Security Modules framework[14], abstracting the implementation of both, Linux focused mainly on offering means for an administrator to define a detailed security policy, hoping to minimize the effect of a vulnerability.

Not lagging behind too much, though, exploit mitigation technologies also appeared in the official Linux kernel during 2004-2005; in fact, they also made an entrance to the official Windows world with Windows XP SP2[15], and Windows Vista is expected to include even more such technologies[16].

Simply put, all three major approaches have been practiced by widely used operating systems at one point or another. It is clear to see that although initially a single approach was chosen, eventually it was understood that layered security is the key to stronger defense of computer systems.

2.3 The NetBSD Perception of Security

Learning from others' experience, the approach taken by NetBSD employs three main principles:

- Simplicity. There is no point in providing a feature, whether it's a kernel subsystem or a userland tool, if it's not intuitive and easy to use. Furthermore, overly complex code is harder to audit, which may lead to additional bugs.

---

[7] http://www.openbsd.org/security.html#process
[8] http://www.monkey.org/openbsd/archive/misc/0207/msg01977.html
[9] http://www.securityfocus.com/news/493
[10] http://www.openwall.com
[11] http://pax.grsecurity.net
[12] http://wt.xpilot.org/publications/posix.1e/
[13] http://www.nsa.gov/selinux/papers/module/t1.html
[14] http://www.kroah.com/linux/talks/usenix_security_2002_lsm_paper/lsm.pdf
[15] http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx
[16] http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx

- Layered security. It is well understood that there is no single solution to security. NetBSD addresses security from a variety of angles, including code auditing, adequate and extensible security infrastructure, and exploit mitigation technologies.
- Sane defaults. Accepting that security may not be the highest priority for all users, NetBSD provides sane defaults to fit the common case. Detailed supplementary documentation helps enable and configure the various security features.

Using the above guidelines, a variety of security solutions were evaluated to address different threat models. With emphasis on implementing a solution that would fix a real problem and provide an intuitive and easy to use interface (when one is required), a variety of changes – ranging from tiny hooks, through additional kernel subsystems, to architectural modifications, NetBSD has made important first steps in improving its overall security.

3. Overview of Recent NetBSD Security Enhancements

3.1 Kernel Authorization

The introduction of kernel authorization, often referred to as kauth(9), in the NetBSD kernel has been one of the larger-scale changes ever done in NetBSD. The interface is modeled after an interface of the same name developed by Apple for Mac OS X[17], though unfortunately due to licensing issues it was impossible to make use of existing code, and so the NetBSD implementation was written completely from scratch.

Kernel authorization redefines the way credentials are handled by the kernel, and offers a simple and easy to use – yet powerful and extensible – kernel programming interface to enforce security policies. It is important to emphasize that kernel authorization does not provide any additional security by itself, but rather provides an interface on top of which security policies can be easily implemented. The strength of the security directly depends on the strength of the policy used.

The kernel authorization infrastructure is required for supporting fine-grained capabilities, ACLs, and pluggable security models among other things. It will allow NetBSD administrators and users to maintain the existing traditional Unix security model, offer capabilities to replace set-user-id and set-group-id programs, and allow third-party developers and appliance manufacturers to implement a custom security model to either replace or sit on-top of the existing one.

3.1.1 Related Work

Similar infrastructures are Linux's LSM (discussed earlier) and TrustedBSD's (now in FreeBSD) MAC framework[18]. Both have been in use for a couple of years, but like kernel authorization, are still very young to backup with real-world experiences.

3.1.2 Design

Apple did most of the design work for the kernel authorization infrastructure. A large part of the design is available online, and it's merely the implementation that was

[17] http://developer.apple.com/technotes/tn2005/tn2127.html
[18] http://www.trustedbsd.org/trustedbsd-discex3.pdf

unavailable. Therefore, most of the design-related work in doing the native NetBSD port focused on completing the missing parts from the online documentation and taking care of compatibility issues.

Kernel authorization maps the privilege landscape of the kernel to *actions* grouped as *scopes*. For example, the *process scope* groups *actions* such as "can trace", "can see", and "can signal" – which are all operations on processes.

When a request for an operation is made, the *action* is passed to the *authorization wrapper* of the relevant scope, together with related context. The context is variable: it is different for each request. The authorization wrapper dispatches the request and the context to the *listeners* associated with the scope. Each listener can return a decision – either allow, deny, or defer (indicating the decision should be left to the other listeners) – and the authorization wrapper evaluates the responses from all listeners to decide whether to allow or deny the request.

In order for a request to be allowed, no listener may return a deny decision. If all listeners return a defer decision, the request is denied.

3.1.3 Implementation

The implementation of kernel authorization in NetBSD was done in several stages. First, the backend was written. This included the majority of the code that worked behind the scenes to implement the credential memory management and reference counting, locking, and scope and listener management. It was then tested to ensure all parts work as a black-boxes, allowing initial integration in the NetBSD code. Part of that work included merging the contents of the *ucred* and *pcred* structs into a single, opaque (as possible) type called *kauth_cred_t*.

The next step was a series of mechanical kernel programming interface changes. Credentials could no longer be allocated on the stack, and so a lot of code had to be modified to use the kauth(9) memory management routines. Additionally, code that directly referenced members of the *ucred* and *pcred* structures had to be modified to use the accessor and mutator routines provided by the kauth(9) interface. Existing interfaces such as suser(9) and groupmember(9) were deprecated in favor of calls to kernel authorization wrappers, and others such as sys_setgroups(2) and sys_getgroups(2) were modified to use the new interfaces.

The following step consisted of thorough testing – to ensure transparent integration and equivalent semantics – which uncovered some bugs with the kernel authorization code, most of them in the NFS portion of the kernel.

3.1.4 Future Development

While implementing the kernel authorization back-end and making the kernel dispatch its authorization requests to it was an important *ground preparation*, there is more work to be done before declaring this interface useful.

The first step in the integration of kernel authorization was mostly mechanical and transparent to users, intended to preserve existing semantics. The next logical step is to examine the kernel to ensure the interface abstracts the security model used in NetBSD.

Given its heritage, the NetBSD kernel is too tightly coupled with the Unix security model, and the concept of a single super-user with a user-id of zero is often hard-coded. For example, a lot of privileged operations check for an effective user-id of zero directly in the process' credentials structure, not making use of the suser(9) interface.

The next logical step will be to identify these locations, and properly replace these vague effective user-id checks with calls to the kernel authorization interface, describing the privilege required to complete the operation. The same applies to any authorization wrapper calls acting as placeholders, checking for super-user rights.

The above work will result in the complete abstraction of the security model used in the NetBSD kernel, allowing switching easy as a one-line change in the kernel configuration between the Unix security model, a finer-grained capabilities model, or a third-party security model possibly implemented using an LKM.

3.2 Veriexec

Veriexec is NetBSD's file-integrity subsystem, allowing the verification of a file's digital fingerprint before accessing it. Introduced in NetBSD by Brett Lymn in 2002[19] and later integrating work from Vexec of the Stephanie project[20] in 2005, Veriexec provides means to ensure real-time file integrity and monitoring combined with intrusion detection and prevention capabilities.

Initially self-contained, Veriexec's core – the interface for associating meta-data with files regardless of file-system support using in-kernel memory – was recently abstracted[21] to form the Fileassoc interface to satisfy similar needs from other features.

3.2.1 Related Work

Integrity checker implementations have been around for decades. Used for various purposes such as virus protection in DOS and file changes notifications in Unix, the concept itself is not new to the security industry. Dr. Fred Cohen's research was among the first to offer insight about using integrity checkers to protect from malicious software[22]. Tripwire[23], presented by Eugene Spafford and Gene Kim, allowed system administrators to be notified about corrupted or altered files in a timely fashion.

Yet, while there are numerous products for every computing environment, they all share a common set of flaws that prevents them from realizing their potential.

First, none of them integrates with the operating system deep enough to provide real-time protection: most are retroactive tools used to *notify* after changes were detected.

---

[19] http://mail-index.netbsd.org/tech-security/2002/10/30/0000.html
[20] http://ethernet.org/~brian/Stephanie/
[21] http://mail-index.netbsd.org/tech-kern/2006/06/08/0007.html
[22] http://vx.netlux.org/lib/afc03.html. Dr. Fred Cohen also introduced the concept of *integrity shells*, with which Veriexec is sharing some commonalities; no implementation was made available, however, and therefore it is impossible to tell whether the faults mentioned also apply to them.
[23] http://portal.acm.org/citation.cfm?id=191183

This approach does not address potential damage that be caused in the time-window between a file was altered and improperly used since and when the administrator receives notification of the matter and handles it. It also does not guarantee the integrity of the integrity checker itself: a successful compromise has the potential of remaining *under the radar.*

Furthermore, some implementations use weak algorithms[24] to calculate a file's checksum, or rely on a small data-set for checksum calculation. Other implementations rely on a file's attributes rather than data to evaluate integrity. The impact of the above is that a file can be modified in such ways that even if the integrity checker tries to evaluate it after the change, it will not be able to detect it. Whether it's by altering the file in a way to defeat the checksum algorithm, or modify areas of the file that the integrity checker is known to ignore, or even tamper with the file's attributes – these implementation flaws can all be bypassed by an attacker quite easily.

And last, they all leave out an important aspect in today's reality: the network. Our environments become more and more inter-connected; we access files from untrusted locations on a daily basis; some architectures rely on a networked environment for everyday operation: centralized storage, backup, and so on. Existing products may provide a certain level of local protection on a host, but leave an important – and interesting – question unanswered: how do you cope with the compromise of a remote resource?

While we cannot deal with all aspects of compromise of a remote resource we use, and it is certainly not our goal either, it is important to try and address the ones that can be solved by using an integrity checker integrated in the operating system.

3.2.2 Design

Veriexec was designed to be a file-system independent integrity subsystem protected from users, including root, by operating solely from the kernel. Recent attacks against various hashing algorithms once thought secure proven the need for interface flexibility – such that can be used both for easy addition of support for new hashing algorithms, as well as future work on digitally signed files.

Careful analysis of the bottlenecks for file access and other file-system semantics (such as rename and remove) resulted in generic hooks, to be called with the required context for decision-making and policy enforcement. At the time of writing, it is impossible to implement the Veriexec policy on top of kauth(9) due to lack of required scopes.

The design process also took into account various environments for Veriexec – from workstations, through servers and critical systems, to embedded task-oriented appliances. Strict levels with varying implications were introduced to support multiple uses, and were named semi-descriptively to hint for said uses: learning mode (level 0), intrusion detection system (IDS) mode (level 1), intrusion prevention system (IPS) mode (level 2), and lockdown mode (level 3).

[24] For example, CRC: https://www.kb.cert.org/vuls/id/25309

3.2.3 Implementation

The most recent version of Veriexec is implemented using the Fileassoc subsystem for management of meta-data and file association, greatly simplifying the Veriexec code, and a device for kernel-userland interaction.

Veriexec is implemented by hooking policy enforcement routines in various parts of the kernel, monitoring execution of normal executables as well as scripts, opening of regular files, and rename and remove operations.

When a file is opened or executed, the evaluation routine, is called with the context of the request (LWP, vnode, filename if any, and access flag indicating how the file was accessed) to make a decision whether the file can be accessed or not. The result is cached to speed-up further evaluations of the same file.

3.2.4 Future Development

During research work on Veriexec, Thor Lancelot Simon pointed out a potential attack[25]. Although Veriexec ensures integrity of files on local file-systems, where all access is done via the kernel, it cannot ensure integrity of files located on remote hosts, imported via NFS, for example.

While Veriexec could be told not to cache the evaluation of such files, the attack vector is when a process, or part of it, is paged-out and later paged in. Because the disk read is done by the VM system, and only of pieces (pages) of the program, Veriexec wasn't aware of it. If the remote host would be compromised, an attacker could write malicious data to the on-disk program, force a memory flush, which would later force a page-in, effectively injecting the malicious data into the address space of the running process on the Veriexec-protected host.

The remedy to this problem is in the form of per-page fingerprints. During fingerprint database generation, the administrator can add the *untrusted* flag to entries located on remote hosts. Veriexec will generate per-page fingerprints for them, and hook the VM system so that when a page-in occurs, the fingerprints of the relevant pages will be evaluated and compared to those calculated previously.

Another natural development for Veriexec would be to introduce support for digital signatures; that is discussed in subsection 4.2.

3.3 Exploit Mitigation

Exploit mitigation techniques are part of the layered security approach of NetBSD, complementing code auditing and more traditional security features, not intending on replacing them.

The purpose of exploit mitigation technologies is to interfere with the exploit code itself, preventing entire classes of exploits from working by short-circuiting common exploitation techniques. One popular example is making sure areas of the memory that are writable, such as the stack and the heap, are non-executable, and vice versa: areas that are executable, such as the where the program's code is, are not writable.

[25] http://mail-index.netbsd.org/tech-security/2002/11/01/0010.html

This prevents exploits that rely on injecting malicious code to a program's memory from working, because said code cannot be executed.

3.3.1 PaX MPROTECT

For a while NetBSD had support for non-executable mappings[26] on hardware platforms that allow it. However, experienced hackers have found a variety of ways to bypass them. Two of these are return-to-lib exploits[27] and trashing arguments to mprotect(2) to change the protection of memory[28].

The PaX MPROTECT[29] feature was developed to address the latter. It enforces a policy where memory that was once writable will not be able to later gain executable permission, and vice versa.

Naturally, this policy may break existing applications that make valid use of writable and executable memory, such as programs that load dynamic modules. For this reason, a tool is provided allowing marking executables as excluded from the PaX MPROTECT policy. It is also possible to revert the policy, applying it only to executables marked with an explicit enable flag.

While it is possible to modify programs that currently violate the PaX MPROTECT policy to continue working correctly without doing so, this would be an unfeasible effort with third-party applications.

3.3.2 SSP (Stack Smashing Protection) Compiler Extensions

Hiroaki Etoh developed SSP (also known as *ProPolice*) in IBM Research[30]. Its purpose is making exploitation of certain buffer overflows harder by placing random *canary values* right before the function return address on the stack, as well as reordering variables on the stack making it harder – if not impossible – to overflow stack buffers in order to overwrite integers or function pointers, preventing exploitation even without altering the return address.

First introduced in the OpenBSD 3.4 release, a similar functionality is now available in the stock gcc 4.1 compiler, recently integrated in NetBSD by Matthew Green.

3.3.3 Future Development

One of the planned features in this area for NetBSD is implementing PaX Address Space Layout Randomization[31]. Also developed by the PaX author, ASLR addresses exploitation via return-to-lib attacks[32] by randomizing the location in memory of shared libraries used by the application, thus making it a lot harder to correctly guess the location of library functions within the application address space.

---

[26] http://netbsd.org/Documentation/kernel/non-exec.html
[27] http://seclists.org/lists/bugtraq/1999/Mar/0004.html
[28] See thread http://seclists.org/dailydave/2004/q2/0045.html.
[29] http://pax.grsecurity.net/docs/mprotect.txt
[30] http://www.trl.ibm.com/projects/security/ssp/
[31] http://pax.grsecurity.net/docs/aslr.txt
[32] PaX ASLR addresses more than that; it also randomizes stack/heap base addresses for both userland and kernel threads.

As expected, hackers found ways to bypass ASLR. The two most commonly used attacks either combine an information leak bug leading to the disclosure of the location of libraries[33], or brute-force exploitation on respawning daemons in an attempt to guess the correct address in one of many attempts[34].

An ASLR implementation would not be complete without a solution to the latter technique. Such a solution, developed by Rafal Wojtczuk, is *Segvguard*[35], employing the basic concept of monitoring the rate of SIGSEGV signals sent to an application in a given time-frame, in an attempt to detect when a brute-force exploitation attack is taking place and prevent it by denying execution of the offending application.

A similar monitor will be introduced in NetBSD once ASLR is implemented.

3.4 Misc. Features

3.4.1 Information Filtering

One of the most common requirements from multi-user systems (such as public shell providers) is that users will not be able to tell what other users are doing – such as running programs, active network connections, login/logout times, etc.

NetBSD implements the above using the kernel authorization interface, and presents the administrator with a single knob that can be either enabled or disabled. When enabled, the authorization wrappers will match credentials of the two objects (the *looker* and the *lookee*) and return the decision.

This abstraction makes it easier to change the behavior of this feature in the future.

3.4.2 Strong Digital Checksum Support

Support for SHA2 checksums has been available in the NetBSD kernel for a while, mainly for the use of the IPSec network stack. Userland, however, was largely neglected. Tools such as cksum(1) and mtree(8) were able to make use only of hashes that were proven weak[36]. Given mtree(8) can be used to evaluate file-system integrity, this was rather dangerous.

The recent improvements to Veriexec, allowing it to support SHA2 hashes, amplified the need for userland support for SHA2 hashes, and were the trigger to adding SHA2 hash routines to libc, as well as support in cksum(1) and mtree(8).

3.4.3 Fileassoc

Fileassoc is one of the latest additions to the NetBSD kernel. It allows associating custom meta-data with files, independent of file-system support (such as extended attributes) using in-kernel memory. The interface is the result of research of other security features that stressed the need for an abstraction of code previously used exclusively by Veriexec.

3.4.3.1 Design

[33] http://artofhacking.com/files/phrack/phrack59/P59-0X09.TXT (mirror)
[34] http://artofhacking.com/files/phrack/phrack58/P58-0X04.TXT (mirror)
[35] Ibid.
[36] http://www.schneier.com/essay-074.html

The Fileassoc interface extends an already-existing design used by Veriexec. The requirements for the design were performance – so that using it in performance-critical code would not cause a notable impact on system performance – and ease of use. The interface was extended, allowing more than one *hook* to add its own file meta-data.

Designed with simplicity in mind, the interface allows multiple subsystems to hook private data on a per-file and/or per-device basis.

3.4.3.2 Implementation

To achieve the desired goal of near-zero performance impact of entry lookup, the Fileassoc subsystem makes use of hash tables and linked-lists to resolve collisions. The interface operates on *struct mount* * and *struct vnode* * to identify file-system mounts and files, respectively. While the internal implementation identifies a file as a pair of *struct mount* * and a file-id – the contents of *va_fileid* after a successful VOP_GETATTR() call – this is planned to change in the near future (see subsection 3.4.3.3).

In the current implementation, Fileassoc allows four hooks (which can be modified with a kernel option) to add private data to each file. This is transparent to the users of the interface, allowing changing in the future, if such is required.

3.4.3.3 Future Development

As previously mentioned, Fileassoc still relies on the *va_fileid* field as the unique identifier for files. This is an internal implementation detail, and expected to replaced in the future with file-handles by using calls to the file-system specific *vptofh()* routines.

3.4.4 Password Policy

Administrators often need to enforce a password policy on the system – either a system-global policy, per-application policy, or even a network-global policy. To address that issue, the password policy, or *pw_policy(3)*, interface was developed.

With flexibility and simplicity in mind, the *pw_policy(3)* interface was designed to allow an administrator to specify password policies via a collection of keywords, and applying them to named entities.

The interface is part of *libutil* and is small enough to be used from within any existing application. It was designed in a modular way, allowing future support for more keywords and evaluation routines.

4. Current NetBSD Security Research and Development

Discussed so far are solutions already implemented and available in NetBSD. Below you will find the current goals of the security research done in NetBSD, some of which are planned to be introduced as soon as the NetBSD 5.0 release.

4.1 Deprecating The Kernel Virtual Memory Interface kmem(4)

The kmem(4) device allows raw reading of kernel memory. It was introduced to allow programs that needed information from the kernel a way to extract it by reading the symbol list from the live kernel's on-disk image and seeking to it.

Several issues were raised regarding this device[37], and with 4.4BSD a new interface meant to replace kmem(4) was introduced, named *sysctl*. Sysctl allowed structured and controlled access to kernel information via syscalls carrying a *management information base* (MIB). The kernel held a tree-like hierarchy of information it can provide, and the MIB described what information is looked up.

From a security point of view, the kmem(4) device allows malicious processes running with *kmem* or *root* privileges to directly read or write kernel memory[38]. The attack vector here is widely abused[39] [40] mainly to introduce stealth rootkits into compromised systems.

Currently, NetBSD is doing loose usage of the kmem(4) interface, using it for more than a few userland utilities. There is an on-going effort to gradually convert programs using kmem(4) to sysctl with proper kernel support, allowing us to deprecate daily use of kmem(4) and maintain the interface for debugging needs only, if required.

4.2 Digitally Signed Files

At the moment, the Veriexec subsystem provides integrity based entirely on data. While it is strong enough to maintain file-system integrity on servers and critical systems, it lacks two important features: ability to securely modify the *baseline* during runtime, and ability to associate an identity with a file-system object.

Securely modifying the baseline during runtime is forbidden, even for the super-user, for security reasons: a possible scenario is that the host can be fully compromised and trojanned by an attacker; preventing the super-user from modifying critical programs can prevent that.

Associating a digital signature with a file-system object, regardless of implementation, could solve the above two by allowing an administrator to specify *trusted entities*. These could run any programs – as long as they are signed by them. That would mean that introducing a new program on the system required digital signing by a trusted entity, rather than a super-user adding its digital checksum to a database and rebooting.

It is planned to extend the Veriexec subsystem with this capability, in either one of two possible directions for the implementation; either delegating the digital signature processing to a user-space daemon, or making use of the BSD-licensed BPG inside the kernel.

4.3 Access Control Lists

Perhaps one of the longest remaining Unix relics in NetBSD is the file-system security model. Proven weak over time, modern operating systems implemented file-system access control lists, or *ACLs*.

---

[37] "The Design and Implementation of the 4.4BSD OS", pages 509-510.

[38] The use of raw access to bypass a security guard isn't limited to kernel memory: on-disk inodes could be modified using raw disk access, for example.

[39] http://artofhacking.com/files/phrack/phrack58/P58-0X07.TXT (mirror)

[40] "Rootkits: Subverting the Windows Kernel", chapter 7.

An ACL allows finer-grained file access, extending the *owner-group-other* scheme currently used.

There are two main issues when approaching file-system ACLs. The first is where to store them, and how to associate a potentially variable sized data-structure with a file. The second is what ACL model to use, which may dictate interoperability with other operating systems.

For the former, NetBSD provides both the UFS2 file-system[41], where extended attributes were introduced especially to address this issue, as well as the Fileassoc kernel interface, allowing file-system independent association of meta-data, after such data has been loaded via a driver.

Given recent standardization in ACL structure between Windows NT, Mac OS X, and NFSv4, it was decided to go with the same model for the latter, allowing NetBSD to properly operate in a heterogeneous environment.

4.4 Capabilities

Part of Unix's long-standing weaknesses is the use of set-id programs to elevate privileges of a normal user, either temporarily or permanently, required to complete an operation restricted to the super-user – for example, open a raw socket, bind to a reserved port, and so on.

The above lead to the absurdity that bugs in often trivial and non-critical programs could result in privilege abuse or even full system compromise.

Introducing capabilities, implemented as a set of kernel authorization listeners, will replace the role of the set-id bit in today's systems. Providing a fine-grained privilege model, each program will run with the minimal set of capabilities required for its operation. Furthermore, associating capabilities with users will allow us to define *user roles*, dividing the work-load of the super-user – possibly eliminating it entirely!

While a design for NetBSD capabilities hasn't been laid out yet, it is expected that support for capabilities will be provided on the file-system layer, allowing the association of capabilities with a program using extended attributes, as well as an API a la OpenSolaris ppriv(3) for dropping unneeded capabilities during runtime, and a mechanism for associating capabilities with users on the system.

5. Component Interaction

So far the focus was on introducing the new infrastructure and features in NetBSD, as well as some on-going development. However, no emphasis was put on the interaction between the various components, and how they all cooperate and contribute to NetBSD's layered security model.

Throughout this section we'll examine the role of each feature in the layered security model.

---

[41] http://www.usenix.org/events/bsdcon03/tech/mckusick.html

5.1 Attack Vectors

Attacks can be conducted on various parts of the system, most commonly exploiting bugs in services (remote and local), misconfigurations, general program misuse, and user actions monitoring. Furthermore, post-compromise attacks include implanting trojan horses, backdoors, and rootkits.

Being a multipurpose operating system, NetBSD's security was designed to also be flexible and without a single point of failure: acknowledging different needs in different environments, the various security features are fully customizable, and the system is configured with sane defaults to ease administration.

5.2 Layer One: Exploit Mitigation and Privacy

In attempt to render an exploitation attempt itself as useless, the exploit mitigation features in NetBSD provide the first layer of security. The *curtain* hooks help protect the privacy of users in a multi-user environment, minimizing the potential of pre-attack information gathering and reconnaissance.

5.3 Layer Two: Capabilities

As discussed in subsection 4.4, capabilities are planned to replace the set-id bit. This effectively reduces the amount of privilege each program is running with. Successful exploitation of programs that today could result in pivoting[42] or super-user account compromise will result in a less critical privilege elevation in the worst case, limiting the impact of vulnerabilities on the overall security of the host.

5.4 Layer Three: Signed Files

Mentioned in subsection 4.2, signed files are the natural evolution of Veriexec, basically associating a signing entity with a file in addition to its digital fingerprint. The immediate benefit is obviously in introducing trust in networked environments, where files can be safely exchanged without fear of attacks such as man-in-the-middle[43].

Accessing files – in particular, running programs – that are signed by "trusted" entities in the default configuration could help reduce the possibility of running manipulated binaries even in face of attacks on the digital checksum algorithm. Doing so in the event of a compromise, combined with Veriexec's *lockdown mode*, will allow real-time investigation and remedy.

5.5 Layer Four: File-System Integrity

Interesting uses for Veriexec (presented in subsection 3.2) are its IDS and IPS modes. With functionality somewhat resembling a *fly-trap*, Veriexec in IDS mode can be used to silently monitor operations on critical system files (services, configuration files) in real-time, preventing any access to them once changed. This can make post-mortem analysis an easier task. IPS mode can be used to prevent access to these files altogether and generate proper log-files to help identify the source of the attack.

These two modes of operation can ensure file-system integrity even in the face of a super-user compromise, making it easier for an administrator to handle an attack

[42] Transition from one user to another.
[43] Assuming, of course, that the kernel itself cannot be manipulated.

without fear of trojanned, backdoored, or otherwise modified (via configuration files) services.

5.6 Layer Five: Protected Kernel Memory

Aimed at preserving kernel memory integrity, the work-in-progress for deprecating kmem(4) usage should result in the ability to remove the interface altogether[44], preventing the possibility of kernel memory manipulation by a malicious superuser on a compromised host. The benefit is obvious: no sophisticated rootkits or kernel-level backdoors can be implemented[45].

6. Conclusion

Throughout this paper I've outlined the recent enhancements in NetBSD security in terms of infrastructure and features, and how they conform to NetBSD's perception of security. Finally, I've exposed some on-going research and development, and showed how it all works together to create a more secure platform

While it is true that a lot of work is still ahead of us, this paper exposed the lot of work that is behind us. Over the past year NetBSD improved a lot on the security front, and it is expected that these efforts will pay off – if not already – within the next major release.

6.1 Availability

NetBSD 4.0 will include kernel authorization[46], PaX MPROTECT[47], GCC 4.1 with ProPolice, the information filtering hooks[48], fileassoc(9) [49], and pw_policy(3) [50].

Complete abstraction of the security model using kernel authorization is being considered for NetBSD 5.0, as well as PaX ASLR and a SegvGuard, Veriexec support for per-page fingerprints and digital signatures, file-system ACLs, and capabilities.

7. Credits

Thanks to the folks who reviewed this paper, either in part or in whole, helping improve its accuracy, readability, and quality. Jason V. Miller, Brian Mitchell and the guys at ISS, the PaX author, and Sean Trifero, Johnny Zackrisson, and Christos Zoulas.

Thanks to Brett Lymn, the PaX author, Bill Studenmund, YAMAMOTO Takashi, Matt Thomas, Jason R. Thorpe, and Christos Zoulas for helping with implementing the features discussed in this paper.

---

[44] From most systems. X would still require it without the use of an aperture driver.
[45] Unless, of course, a kernel vulnerability is successfully exploited.
[46] http://netbsd.gw.com/cgi-bin/man-cgi?kauth++NetBSD-current
[47] http://netbsd.gw.com/cgi-bin/man-cgi?paxctl++NetBSD-current
[48] See the *security.curtain* knob.
[49] http://netbsd.gw.com/cgi-bin/man-cgi?fileassoc++NetBSD-current
[50] http://netbsd.gw.com/cgi-bin/man-cgi?pw_policy++NetBSD-current. No programs were made aware of the interface yet, though.

`nnpfs` File-systems: an Introduction

Kristaps Dzonsons

EuroBSD Conference, 2006

Abstract

Writing a file-system is tricky business. Every kernel has a different way of operating upon file-system data; every kernel has a different set of interfaces with which to accomplish these operations. The `nnpfs` kernel driver exports file-system operations for a given mount-point to a character device with a byte-layout specified by an interface. This allows one to write generic user-land file-systems for platforms supporting the `nnpfs` driver, including the BSD family of operating systems. In this document, we present `xfsskel`, which demonstrates a simple "null"-like file-system mounting a loopback file-system sub-tree. The purpose of `xfsskel` is not, however, to provide production file-system; it is, rather, a tool to provide thorough documentation on writing file-systems for the `nnpfs` driver. With such documentation, we believe that many pseudo-file-systems missing across the BSD operating systems may be made available in a clean, cross-platform manner.

1 nnpfs

The `nnpfs` driver is provided as part of the arla [1] project. arla is a free AFS implementation available for a variety of platforms. The `nnpfs` driver allows arla to operate on diverse systems by providing the user-space with a uniform interface to file-system changes. Although writing an in-kernel full-system driver would provide superior operating performance (as demonstrated, for example, with `nfs`), the complexity in maintaining in-kernel drivers for multiple platforms is significant. arla thus produced a generic interface which it set atop a system-specific driver `xfs` (since renamed to `nnpfs` to avoid name collision with other projects). With `nnpfs`, file-system operations are provided to a character device conforming to a set of interface definitions; thus, the `arlad` user-space daemon needn't know about operating system pecularities to function.

Note well that the `nnpfs` driver is by no means a panacea for writing cross-platform file-systems. The driver creates significant overhead in exporting file-system operations, a penalty not taken by in-kernel file-systems. When speaking of `nnpfs`'s utility below, it should be noted that this document biases "file-system" towards "pseudo-file-system", which may be loosely defined as providing a service layer above lower-level file-systems. A "file-system", in this parlance, would be `ffs` or `ext3`, while a "pseudo-file-system" would be a remote file-store accessed via `ssh(1)` or `ftp(1)`.

1.0.1 Supported Platforms

To date, the arla `nnpfs` driver has been ported to FreeBSD, NetBSD, Linux, OpenBSD, Mac OS X[2] and others. It's an ideal candidate for writing file-systems in the user-land that span many architectures. This lecture will focus on the BSD family of operating systems: OpenBSD, FreeBSD and NetBSD[3].

1.0.2 Installation

The `nnpfs` driver itself is part of the base kernel provided by OpenBSD. On NetBSD and FreeBSD, it must be downloaded and compiled separately, then the kernel driver introduced by means of LKM routines. arla may be downloaded through the third-party *ports* collection; in both systems, found under the `net/arla` namespace. The arla package is required for both `mount_nnpfs(8)` system utility and the kernel module. Consult your operating system's manual on how to install third-party utilities, should it be required. Once installed, you'll have to introduce the kernel module appropriately.

Note that OpenBSD may also be treated as mentioned- by downloading the arla sources and compiling the most current module. Although this would

[1] `http://www.stacken.kth.se/project/arla/`

[2] See the arla web-site for current systems.

[3] At the time of writing, OpenBSD is at 3.9; NetBSD at 3.0; and FreeBSD at 6.1. Since this document is heavily platform-specific, upgrades to systems may result in interface and naming changes.

bring the OpenBSD installation up-to-date (see the "Caveats" section, below), we concentrate on the available package in this lecture.

1.0.3 Caveats

The OpenBSD distribution of arla is out-of-date. OpenBSD still refers to the arla interface with "xfs", which has since been replaced by "nnpfs". This document will refer to `nnpfs` exclusively. We'll take special care to mention when these naming discrepancies occur, and they'll occur often. For example, while NetBSD's utility for mounting `nnpfs` points is `mount_nnpfs(8)`, OpenBSD's remains `mount_xfs(8)`. Interface references will use "nnpfs" in naming.

2 xfsskel

The `xfsskel` project[4] came about while researching the various tools for exporting file-system operations from the kernel, ostensibly to write a replicating object file-system operational across the BSD family of free operating systems. The original requirements were:

1. cross-platform (at least OpenBSD, NetBSD, FreeBSD, and Linux)
2. stable (must not require complex administration)
3. well-documented (low overhead in implementation)
4. free (permissible license for maximum re-distribution)

The immediate candidate was `FUSE`[5], File-system in Userspace. However, `FUSE` is limited in its portability. As the time of writing, this is limited to Linux and FreeBSD. arla's implementation was found during research for other freely-available replication agents.

Although `nnpfs` met most requirements, it had very poor documentation. By sifting through the arla code and other implementations of the system, we were able to extract the necessary protocols required to interface with the kernel's file-system export routines.

This document will focus on the original research's target – a replicating file-system `rfs` – to describe potential strategies in implementing a user-space file-system. The `rfs` has not yet reached a state where demonstrable code is available; however, since this is used only to posit strategy, source code is not necessary.

Note that `xfsskel` refers both to a body of source code and considerable on-line documentation describing the `nnpfs` interface. In generally, we'll refer to `xfsskel` as a body of source code.

[4] http://sysjail.bsd.lv/xfsskel/
[5] `http://fuse.sourceforge.net/`

2.1 Source Code

The `xfsskel` system is a simple file-system that mounts another sub-tree within the operating system at the `nnpfs` mount-point. For example, one may the sub-tree `/usr/local/` under `/mnt/nnpfs/`. This is similar to NetBSD's "null" file-system (`mount_null(8)`). The intention of `xfsskel` is not to reproduce this function, nor to provide a production-level file-system implementation, but to provide a source-code (and on-line) reference for interfacing with the `nnpfs` driver. To this effort, the code itself is designed with stepping-through in mind: the path from receiving events on the driver to operating upon them is very simple to follow.

3 Getting Started

Using `nnpfs` is fairly simple on all systems. On OpenBSD, the kernel driver and associated user-space utilities come bundled with the system. On NetBSD and FreeBSD, one must download the user-space tools separately from the appropriate package management tools. You'll also need to download the `xfsskel` package if you wish to examine the operation of the system.

3.1 Kernel Options

`nnpfs` is a kernel option on the OpenBSD system, where it's enabled by default; on FreeBSD and NetBSD, the kernel module must be dynamically loaded. Consult your operating system's kernel compilation manual for instructions on how to do this.

Once the kernel has been booted (or module installed) and the user-space utilities appropriately installed, you'll have to instruct the kernel to direct all requests for the mount-point to the `nnpfs` driver. On OpenBSD, you'll have to address with the old naming scheme.

 # /sbin/mount_xfs /dev/xfs0 /mnt/xfs0
 # /sbin/mount_nnpfs /dev/nnpfs0 /mnt/xfs0

Consult the appropriate manuals (`mount_xfs(8)` or `mount_nnpfs(8)` before running anything. Once executed, the kernel will export messages to this mount-point to the `nnpfs` character device as listed. If the device is not opened and accessed by an appropriate executable, requests for data will immediately fail.

To examine `xfsskel`, first compile the sources, then execute the foreground process as follows:

 # xfsskel

By default, `xfsskel` will listen on the `/dev/xfs0` device and map the `/` file-system. To change these values, you'll have to edit portions of the source code.

4 nnpfs internals

The `nnpfs` driver exports data to the user-space by a well-defined set of interfaces. On OpenBSD, these are exposed in header files at `/usr/include/xfs/`. On NetBSD and FreeBSD, you'll have to copy the files manually or point your Makefiles to the appropriate subdirectories (for compilation). Any system handling the `nnpfs` driver's device data will need these interfaces. The most significant is `xfs/xfs_message.h` (`nnpfs/nnpfs_message.h`) (hereafter referenced with "nnpfs"). In this file are the structures that define the structure of data events raised through the device.

4.1 Basics

To process requests from the `nnpfs` driver, you'll first need to receive events. This may be accomplished through a standard blocking `read(2)` on the appropriate device (`/dev/xfs0` on OpenBSD, `/dev/nnpfs0` on FreeBSD and NetBSD) or, in the event of more complex systems, usage of `select(2)` or `poll(2)`. `xfsskel` uses a simple blocking `read(2)` call, as it has no other function beyond servicing file-system requests.

A system processing events must offer two services. The first is to read appropriate messages from the kernel driver, and the second is to acknowledge events and action taken on the events. These may be serviced in any particular fashion: since service requests contain an acknowledgement key and are asynchronous, in theory, one could have multiple listening daemons that communicate among one another. For example, one can have a daemon that does nothing but receive messages and modify a shared work queue. These events could be serviced by another process, in turn modifying another work queue. Finally, a third process would be responsible for pushing acknowledgements back into the kernel. Whether this would offer any performance advantage is arguable – certainly, were the system able to process events in parallel, requests could be handled in a different processing context. In most file-system implementations, however, processing is I/O bound and extra processing elements would add little beyond complexity. `xfsskel` has a simple, sequential work-flow so as to keep details as obvious as possible. Unless operations on file-system data are CPU-bound (hashing, compressing, and so on), the benefits of simplicitly outweigh negligable increases in speed.

4.2 Interfacing

The main interface to `nnpfs` lies in `nnpfs/nnpfs_message.h`. This header file contains the structure of messages as they're passed from the `nnpfs` device. The following code fragments demonstrate the process of reading requests from the `nnpfs` device `/dev/xfs0`. Assume the following variables throughout:

```
struct nnpfs_message_wakeup wake;
struct nnpfs_message_header *hdr;
char buf[NNPFS_MAX_MSG_SIZE];
```

```
int fd;
```

First, one must initialise the device.

```
fd = open("/dev/xfs0", O_RDWR);
```

Next, messages may be read from the device as they appear. This fragment uses a simple blocking read; more complex approaches may be necessary depending on usage.

```
read(fd, (void *)buf, NNPFS_MAX_MSG_SIZE);
hdr = (struct nnpfs_message_header *)buf;
```

Messages are passed from the device in a byte-sequence defined by C structures. These are created when inheriting structures encapsulate a global header (left-aligned appropriately), so one may read out a generic header, switch on its type, cast to the appropriate subtype, and descend appropriately. Note well that messages must be read with a size of `NNPFS_MAX_MSG_SIZE`; since the underlying device does not buffer, requests to read smaller requests will yield errors.

```
switch(hdr->opcode) {
case(NNPFS_MSG_VERSION):
case(NNPFS_MSG_WAKEUP):
        /* Process these requests, etc. */
}
```

Assuming that the **struct** nnpfs_message_wakeup wake variable has been appropriately initialised, one may now write a response back to the device.

```
write(fd, (const void *)&wake, wake.header.size);
```

The above fragments are all one needs to get started processing messages. At this point, one must understand the contents of messages and act on their requests.

4.3 Protocol

The above section defines, in code, the sequence of receiving and acknowleging kernel messages What if the kernel requests specific information from the user-space system? If the kernel makes a request for data, it's the user-space system's responsibility to provide the device with necessary structures before acknowledging the request. When an acknowledgement is passed to the kernel, it is assumed that the appropriate data has already been supplied. In this light, the structures defined in `nnpfs/nnpfs_message.h` are divided into requests from the kernel and corresponding responses. Note that not all *all* kernel-user space requests are symmetric. A general sequence follows:

1. receive request1 from kernel
2. receive request2 from kernel

3. response request1-a to kernel
4. response request1-b to kernel
5. response request1-c to kernel
6. ack request2 to kernel
7. ack request1 to kernel

This serves to demonstrate how requests from the kernel are out-of-order and asynchronous: one may respond to any number of messages before responding. The dependence is one of context. When one acknowledges a kernel request, it is assumed that the requested data has been provided. If it has not, the kernel will likely re-fire. Internally, the device maintains a queue of I/O requests and sleeps until an acknowledgement is received. At that point the I/O operation will begin running. It's the programmer's job to ensure that the kernel has all appropriate data before processing.

5 Concepts

To write a system interfacing with `nnpfs`, one must understand some `nnpfs`-specific file-system concepts.

1. How does the driver reference file-system objects?
2. How does the driver signal `read(2)` and `write(2)` data?
3. How are access controls handled by `nnpfs`?

The concepts described below will be explored primarily in words. For "Object referencing", for instance, there are significant supporting in-code structures that exceed the scope of this document. For a complete reference of relevant structures and values, consult the `nnpfs/nnpfs_message.h` header file itself or the `xfsskel` web-site, which has a complete listing of structures and their functions.

5.1 Object referencing

Both the kernel and the user-space implementation must agree upon a convention for addressing file-system objects. For example, the kernel must know how to query a particular file. To accomplish this, the interface defines a 128-bit handle for each object as defined in **struct** nnpfs_nnpfs_handle. When referencing an object (practically, a file; but in theory, any file-system object), the kernel uses this field. It must be installed only once. In the following paragraphs, the term "node" refers to an object in the file-system. A node may be any resident object on a file-system; in the relevant Unix systems, a node may be a named pipe, a directory, file, block device or so on.

`xfsskel` has a simple scheme for node handles. It allocates a node object on the heap (by a data structure containing per-node information, like that maintained by an inode) and uses the pointer as the object reference. Thus, when requests arrive from the kernel, `xfsskel` simply dereferences the appropriate object.

It's of significant note that `nnpfs` may request multiple nodes for the same file-system object. For example, the file `foo/bar` may have several nodes open on it at once. Caching nodes becomes a series issue when balancing many simultaneous node requests.

Once a node has been used, the user-space implementation may request that references to the node are deleted. This is known as *garbage collecting* a node. `xfsskel` does not implement garbage collection at this time, thus in time it will inevitably run out of memory (assuming one continues to operate the file-system).

5.2 Data transfer

Data transfer to and from the user-space system is accomplished by *file-handles*. File-handles provide a means to access file data while bypassing directory permissions: one provides a file-handle to a file path and subsequent operations (`fhopen(2)` et al) need not a direct reference to the file. This allows `nnpfs` systems to be freed from a particular underlying directory structure. Since not all systems provide the file-handle utility, `nnpfs` provides its own version by means of direct `syscall(2)`. The kernel manages open file-handle objects, so the user-space implementation needn't keep track of open file descriptors and so on.

5.3 Access control

Since `nnpfs` does not assume a Unix-like supporting environment, it offers its own flexible means of access control. As with object nodes, it's the user-space's responsibility to provide and enforce access controls (or, as with `xfsskel`, provide none whatsoever).

However, in keeping with the Unix rights-management philosophy, `nnpfs` does have a notion of read, write and execute privilege: "rights". When the user-space system installs a node, it installs with it an array of credentials and rights by mapping identities ("PAG"s) to their rights. A special field is reserved for anonymous rights, should no relevant identities by provided.

When a process accesses the device, it's by default associated with a credential containing only its corresponding UID. The PAG is also this UID – although if the UID changes (say, with `setuid(2)`), the PAG stays the same. One may map PAGs through `syscall(2)` to the `nnpfs` driver so that, for example, a user on one system may have the same PAG as a user on another system. The driver will traffic new PAGs through the system, but it's the user-space system's responsibility to manage and administer PAGs via the system call interface.

6 Implementations

We've discussed `xfsskel`, a simple user-space file-system atop the `nnpfs` driver. Unfortunately, `xfsskel` is woefully inadequate as a production system. In fact, its design specifically allows it to exhaust memory. This section discusses some strategies for designing robust, production-grade file-systems atop the `nnpfs` driver.

6.1 Introduction

In these strategies, we'll examine the replicating file-system as mentioned above: `rfs`[6]. `rfs` should accept reads and writes to its mount-point and propagate the operations among participating nodes in a replicating clique with a well-defined coherence protocol. The system must also accept incoming connections from other nodes in its clique, and respond to their data service requests as well. At any time, the system may have enqueued dozens of simultaneous requests (scaling with the number of participating hosts and the number of local pending I/O requests). The response time of these requests must be as absolutely low as possible. We can make no assumptions as to the underlying hardware in the host. Our only assumption about the underlying host is that it will present a common set of system interface functions to the executable (as defined in the Single Unix Specification and available, among others, on OpenBSD, NetBSD, and FreeBSD). In this document, we'll consider the practical elements of the file-system: how it obtains and distributes data. Note that `rfs` is not designed to be fast: its foremost priority is reliable replication of data. That said, to have a too-slow file-system would be impractical.

6.1.1 Theory

This document describes the *practical* elements of this file-system. That said, the relevant underlying theories are that the sharing algorithm is *pessimistic*, meaning that the coherence protocol assumes failure until all nodes respond accordingly. Second, the directionality of reads and writes shall be unidirectional: a single host maintains a file-lock until changes complete. Following a request for an object, a receiving `rfs` server either forwards the request to the appropriate holding server or initiates a voting session to claim its own lock. This becomes significant when examining the schema for object handles.

The term "node" may become confused when discussing "clique node" (as in member of a replicating clique) or "object node" (as in data structure for a file-system object). We'll be careful to differentiate between the two.

6.2 Process Layout

Although it's tempting to build a complex, multi-threaded user-space system to efficiently manage the possible threads of execution, most of `rfs`'s execution is

[6]Note again that `rfs` is still being researched.

I/O bound. In fact, there's very little processing power required. Most time in execution is spent waiting on data buffers to fill and drain. With this in mind, it's simpler and more effective to design the application to be as asynchronous as possible, and to run within a single thread of execution. While managing fifty threads is a complex and costly task in terms of resources, it's relatively simple to maintain queues of active participants, and serve the I/O requests made available on an edge-triggered basis. Thus, we'll settle with a single process waiting on a dynamic vector of I/O events.

6.3 Servicing events

Events have no priority in our system: events from the `nnpfs` device are similarly prioritised as requests from other nodes in the clique (in practice, this is not necessarily true, but theory details are not relevant to this particular paper). Our system must respond to events from the `nnpfs` device and from network events. Since we'll describe our system as asynchronously as possible, we'll use the UDP protocol for network communication and maintain the status of all pending connections for retransmission.

Once opened, the file descriptor for the `nnpfs` device will be added to a **struct** pollfd vector for use in `poll(2)`. Our application will be driven centrally by a polling call to all readable interfaces (`nnpfs` and listening sockets). We'll instruct `poll(2)` to timeout in order to service potential timeouts from pending, asynchronous waits for acknowledgements in requests for data to other nodes. These pending events will be described in persistent journals (areas memory-mapped to files, preferably on a device that buffers as little as possible), allowing the file-system state to be fully recoverable in the event of system crashes.

6.4 Object References

To identify nodes on a local system (like with `xfsskel`) is fairly easy. Distributed systems are somewhat more difficult: one must know *where* a particular object resides on the network of participating hosts. As mentioned above, changes to an object (following a lookup, wherein the node challenges existing locks) correspond to a unidirectional broadcast from the servicing node within the clique. Thus, we needn't maintain a complex address mapping between object nodes and their content; we can, like in `xfsskel`, maintain a simple lookup. We'll delegate most of this complexity to a node cache, which will manage the cache of available nodes and trigger garbage collection, if necessary.

6.5 Node cache

A node cache must manage the map of node handles to underlying objects. It must trigger garbage collection on an as-needed basis and provide the correct data sets for collection. First, we'll provide a splay tree of object nodes to benefit from request locality, on the assumption that most accesses will be localised. As incoming requests for new nodes are serviced, we'll lookup the node in the

tree with a hash function on the file's provided name and directory parent. If found, the node reference count will increase and a pointer will be returned; if not, a new node will be created and initialised.

The balancing properties of splay trees will ensure that the tree is biased toward locality. Our garbage collection scheme will base itself on both the tree size and depth. Garbage collection will be level-triggered when pre-set limits are exceeded. To provide maximum flexibility, we'll allow these limits to re-size given statistical usage (this shouldn't require more than simple statistical analysis of growth patterns). For example, setting a maximum number of available nodes on a file-system that continuously triggers garbage collection by hovering around the threshold is suboptimal (limits for absolute maxima, of course, will exist).

6.6 Access control

We want `rfs` to provide Unix-like semantics to the calling file-system interface. In deference to simplicity, we make the assumption that each participating node maintains identical UID databases (or common subset of databases that will access the system). This frees us up from complex mappings of users and PAGs, and allows more concentration on the fundamental system underpinnings. In time, this may extend to a more powerful means of authentication, such as LDAP.

6.7 Data transfer

As already mentioned, a host servicing a node request is guaranteed to have a lock on that file. It maintains a file cache on-disc. Although distributing the file in blocks (as does the most currently version of arla) is attractive and relatively simple to implement, it does not show an immediate benefit to the initial implementation of `rfs`. A more advanced system that must account for a wider domain of hosts may want to have more flexibility in striping blocks across hosts. In this scheme, we'll intercept the `nnpfs` driver's notification of written data (`NNPFS_MSG_PUTDATA`) and distribute the blocks as appropriate.

7 Conclusion

By using the `nnpfs` file-system export driver, we've managed to design the concepts for a file-system implementation portable across all systems supported by the device. By adhering to system interface standards, we're guaranteed that code will port to any conforming architecture within the set of `nnpfs`-aware systems. If `rfs` complies with all standards, it stands to be ported immediately to all BSD operating systems, Linux, Mac OS X, and even Microsoft Windows 2000. This provides significant strength to the availability and utility of pseudo-file-systems.

Implementing Lightweight Routing for BSD TCP/IP

Antti Kantee <pooka@cs.hut.fi>
Helsinki University of Technology

Johannes Helander <jvh@microsoft.com>
Microsoft Research

ABSTRACT

The BSD TCP/IP stack is the de facto standard for TCP/IP interoperability. Leveraging as much as possible of the proven code is beneficial to all parties when considering new operating platforms. However, the implementation of packet routing within the stack is targeted for core routers. The code size is huge and presents no benefit for network leaf nodes, especially embedded systems. Additionally, making matters worse, the implementation is convoluted and provides no clear interface for alternate implementations.

This paper discusses a lightweight routing implementation for the BSD networking stack. By replacing the existing routing code by a much simpler implementation the size of the networking stack was reduced by 20%; this is more than the size of the TCP module. For leaf nodes, the functionality and performance of lightweight routing is equivalent to the historic BSD routing code.

1. Introduction

Despite ever-increasing hardware resources, there are still places where it pays to be as small as possible. One of these places is on an embedded device, where kilobytes of additional memory will cost some cents per unit manufactured. Once this cost is multiplied by the number of millions units produced, the business motivations in investing to smaller code become very clear. From a software engineering point-of-view, however, smaller code can either have advantages or disadvantages. It depends on if the reduction was accomplished by architectural structurization or micro-optimization of the existing architecture and their respective proportions. However, this paper will not discuss the merits of either way any further.

While it is relatively easy to write software to match a specification and a certain size target, the challenge is that the real world is very real. It is far from uncommon to see quirks and workarounds in code which has to interoperate with counterparts from other vendors. This is due to specifications being written in spoken language and always containing too much room for interpretation. Hence, it reasons to say that writing software interoperable with real world implementations is difficult.

The BSD TCP/IP implementation has a proven track record in interoperating with the real world. In fact, it can be said to define TCP/IP interoperability. Therefore, when aiming for TCP/IP interoperability, using the BSD code base would be a good approach. However, the BSD code does present some challenges for use. In our case this is the fairly large code size. Some of it can be shaved off using trivial modifications, but taking it beyond a certain point gets hard.

In this paper we present a novel idea for implementing routing in leaf nodes. We show how it was implemented for the 4.4BSDLite2 BSD networking stack and discuss why it was

difficult to do. We also attempt to outline future improvements in the area for a more modular TCP/IP stack.

The rest of the paper is structured as follows: Chapter 2 gives a short overview of our platform, Microsoft Invisible Computing. Chapter 3 presents the current networking stack implementation and our trivial improvements and describes the situation with the routing code. Chapter 4 presents our solution for a major size improvement in the routing code and discusses its implementation. Chapter 5 presents the results and experimental figures for the implementation and finally Chapter 6 presents our conclusions and lines out the future work in the area.

2. Microsoft Invisible Computing

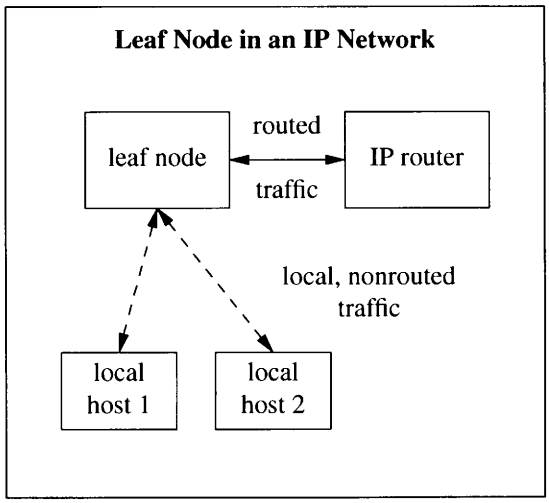
Microsoft Invisible Computing [1] is a research prototype for making small devices part of the seamless computing world. It provides a compact middleware for constructing embedded web services applications and a small component based Real-Time Operating System with TCP/IP networking to make middleware run straight on the metal on several embedded processors. The goal is to make it easy to build custom smart devices and consumer electronics, especially battery operated; and to support research in invisible computing, operating systems, networking, ubiquitous computing, sensor nets, distributed systems, object-oriented design, and wireless communication.

Due to the deployement scenarios, small size for the system is an absolute requirement. Otherwise several interesting cases would be ruled out because of size limitations.

2.1. Requirements

Our requirements for the networking code are dictated by our operating environment. The current target environment is an IPv4-only network. Microsoft Invisible Computing is never being used in the capacity of an IP router, but rather as a data processor positioned as a leaf in the IP network. The system may have multiple interfaces, but only one of them will be connected a smart router.

The main requirement for this work was the reduction of compiled object code size to better be in line with the target environment hardware constraints.



2.2. Size Analysis

For the purpose of keeping the system small, its size is regularly monitored after a build to catch any "creeping featurism" early on. This information is also useful for doing size analysis on the system.

Microsoft Invisible Computing Module Sizes

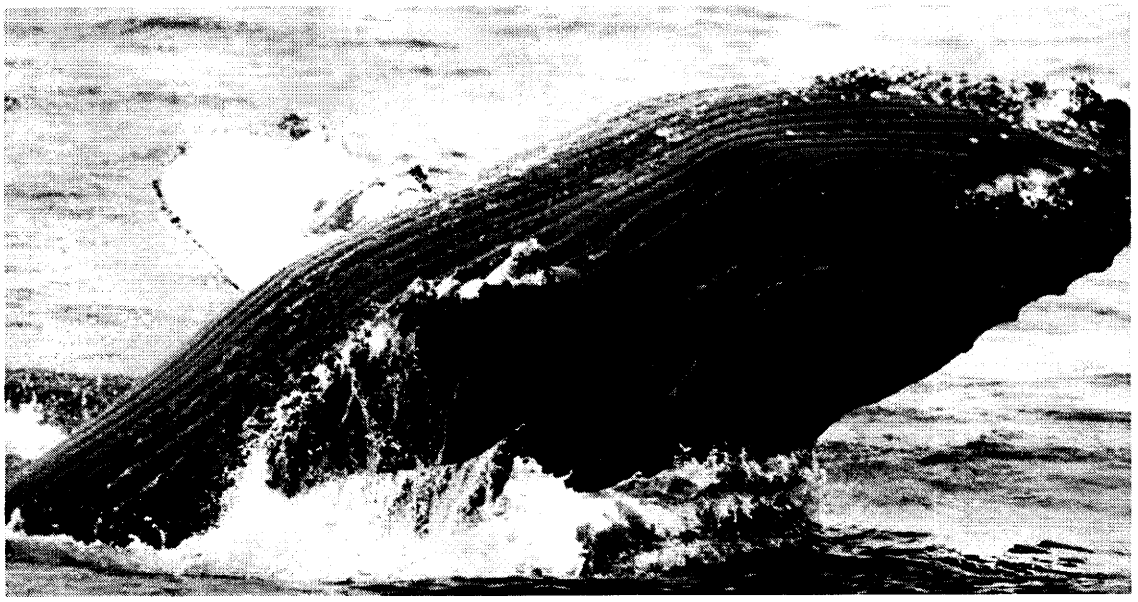
Module	ROM	RAM
BASE	21856	272
MACHDEP	5248	1680
NET	77164	???
TCP	13276	???
XML	12628	16
SOAP	52668	436
HTTP	22692	0
DNS	11052	344
DHCP	5508	96
WSMAN	6740	0
SOAPMETA	17248	20
CRT	14036	32
DRIVERS	11360	308

Clearly, the largest individual module in the system is the network stack. Therefore it is the best candidate for reduction. The SOAP module is also large, but it was already written from ground up with minimalism in mind.

3. Networking Stack

As was identified in the previous chapter, the largest individual component of Microsoft Invisible Computing is the BSD networking stack and therefore was the ideal candidate for size reduction.

BSD Routing Code
(photo Patty Geary)



Our target for the project was to make the networking stack smaller. First off, some trivial enhancements to the networking stack were made. These included mostly replacing macros with function calls[1] and `#ifdef`'ing out all unnecessary bookkeeping code not required for the release build.

For example, the memory buffers (*mbufs*) used by the network code are written so, that most operations handling them are macros. This reduces function call overhead when dealing with them, but inline expansion plays havok on memory footprint and cache behaviour. With modern CPU architectures and their CPU/memory speed ratio, smaller code is usually faster. Additionally, macro-style programming usually does very little to encourage the separation of an interface from the implementation. It also makes run-time loadable components an impossibility due to including the internal representation already in the "calling" code.

Also, another easy way to micro-optimize the routing code size is to replace toggles with inline-`#defined` values. On RISC architectures, this avoids a separate load instruction every time the toggle is referenced. The downside is of course losing the ability to toggle the values runtime.

Using simple tricks like these it is possible to shave several kilobytes off of the networking stack.

After the abovementioned improvements, the compiled network stack size was analyzed and the routing module was found to be the largest individual component in the networking stack. It warranted closer attention to decide its usefulness.

3.1. BSD Routing Code

The current "off-the-shelf" BSD routing code has its roots in 4.4BSD [2]. That code was written with several targets in mind:

- nodes connected to the network through a single interface
- nodes connected to the network through multiple interfaces
- routers involved in packet forwarding
- support for multiple address families
- good performance of computer architectures of that era

While it works for all of the above, it contains extra payload for implementations not needing all the generic routing capabilities. Also, while the

[1] VAX is not one of our target platforms.

code is well described in literature [3], it is difficult to modify because it presents no clear interfaces but rather chooses to optimize itself for performance. Even though this was understandably a noble goal for the original implementation, hardware developments have made most of the employed tricks unnecessary and sometimes even counterproductive.

The kernel routing code itself can be thought of to be divided into three different modules: routing database, routing socket and in-kernel routing interface. Additionally, link layer addressing is joint at the hip to the routing code.

The Routing Database

The routing table information within the kernel is stored within the routing database. This is implemented as a radix tree in 4.4BSD and is classically found under `src/net/radix.c` in the source tree. It tries to optimize the amount of bit comparisons required to find the most specific match[2] for the given search key from the database.

The interface to this code is optimized more towards performance than to provide a modular database interface. For example, the `rn_search()` function returns the subtree in which the key resides - a clear bias toward a tree-shaped implementation. Also, the address and netmask arguments passed to the radix code are `void *`'s, which conveniently happen to contain an 8-bit integer representing their length right at where the pointer points to - a striking resemblance to `struct sockaddr`. And since the address in the `struct sockaddr` layout does not begin right after this, there is per-family offset into the `void *` argument in `struct domain`'s field `dom_rtoffset`. This argument is given to the radix tree when it is initialized.

The routing database algorithm [4] is based on a modified version of the radix search trie [5]. However, several drawbacks involving modern hardware architectures have been identified [6], and clinging on the historic code is not particularly relevant. Even so, it must be kept in mind that the existing implementation is reasonably efficient and, above all, it has proven to work in the real world.

[2] The one where the netmask stored in the database has least amount of 0-bits.

Route Request Interface

The interface for requesting routing information is implemented in `net/route.c`. It essentially supports the following features:

- route query: `rtalloc()` and `rtalloc1()`
- freeing the allocated route: `rtfree()` (note: this does not remove the route from the database)
- rtdirection handling: `rtredirect()`
- interface for use by the routing socket: `rtrequest()`

The route asked for will be provided in `struct route` or `struct rtentry`. The difference between these two is that the former contains a pointer to the latter and a `struct sockaddr` describing the destination; we will see why this is necessary later. `struct rtentry` itself contains a lot of information, for example the data storage elements used by the radix tree, the gateway, some statistics on the route, the interface used for the route, and so forth.

The Routing Socket

The 4.4BSD kernel does not implement a routing policy, it merely forwards packets according to a set of rules. Routing policies, i.e. decision on what the forwarding rules should look like, are made in userspace[3]. This means that the routing policy implementation in userspace must be able to communicate its decisions to the kernel and equally the kernel must be able to communicate any routing information it receives to userspace.

The method for communication is called a routing socket, i.e. a socket opened using the protocol family `PF_ROUTE`. Messages are then exchanged through it back and forth using `struct rt_msghdr` to describe each exchange. It is for example possible to set and change a route.

The routing messages used by the routing socket are spread also elsewhere into the kernel, although the exact message format is contained in the routing socket code. Other components involved must be able to receive information about routes going up or down and must be able

[3] Technically it would be possible to do it in the kernel also. Userspace programming is just easier most of the time (unless you happen to be a kernel hacker who has an attitude toward writing programs in userspace ;-).

to provide information if their state changes. For example, if an interface is detached, routing packets through it is no longer possible and the entire routing chain must be made to know about this.

Another purpose for routing sockets is to be a mechanism to communicate over the user-kernel barrier present in most modern operating systems. This is something which is not required for systems operating on machines without any form of memory protection, as is commonly the case with microcontrollers and, in this special case, Microsoft Invisible Computing itself.

Link-layer Routing

In addition to doing network level routing, the routing code also handles link-layer routing. This means that for example the ARP cache for IPv4 Ethernet is tightly coupled to the routing code.

For all local networks in the system, the routing table contains a route with the approate address/netmask as the key and the correct interface as the gateway. These route entries also have a cloning flag, `RTF_CLONING`, set. This means that if the radix lookup produces a route with the cloning flag set, a route entry for the queried address should be created and another lookup performed[4]. After the radix match finds a cloning route, a link layer resolution is done. As a special case, the ARP code has additional knowledge about this resolution process, since the ARP entries for the whole local network are not cached, but rather pulled in on a per-demand by doing an arpwhohas and interpreting the response (if any).

4. Lightweight Routing for BSD

The first attempt to make the routing code smaller was to replace the radix tree with a less complex structure, which would hopefully lead to reduced code size with equal performance. This was implemented as a Microsoft Invisible Computing Component Object Binary, COB, so the original radix code or the new lightweight code could be preserved.

However, this facile approach produced very little in the form of results. The new code was only around 2kB smaller than the original. This was mostly due to the convoluted interfaces, which still required handling the complex data structures passed to and from the code. A more radical stratagem was therefore required.

Our target was to support only end nodes on the network. This changed the rules for the routing implementation. Routing on leaf nodes is actually an oxymoron. A leaf, per definition[5], cannot involve routing, since it is connected to the rest of the graph only from one point. The rest of the graph is either accessed through that point or not accessed at all.

The slight exception to the analysis above is that our node is only a leaf in the sense of the networking layer. As discussed in Chapter 3, the routing subsystem is also involved in making decisions about the link layer routing. Our "leaf" node will still be connected to an Ethernet, so support for link layer routing must be taken care of.

Based on the observation presented above, the conclusion was made: if routing is not done, code for it is not needed either. Size savings for non-existent code are quite substantial.

4.1. Lighweight Routing Algorithm

The old BSD routing code uses the routing table to make several decisions about the packet's final destination. This information is encoded in the radix table and the decision of how to handle the packet is automatically done during the radix tree lookup. For example, if the lookup produces a link layer address, the code knows that the packet should be sent to a host on the local network, be it the default gateway or just some other host on the local net. Since we plan to have no radix tree, this information must otherwise be encoded into the system.

One important concept to keep in mind to avoid frequent confusion is the concept of the packet target in the routing code. While an IP packet header contains the final destination for the packet, the routing code is interested only in where the packet should be sent next. Therefore, when we are talking about the target, we mean the IP address of the next hop, not the final destination. Once the next hop is discovered, the packet is sent there and it is that host's problem to look at the IP header to discover the final destination and again decide where it should be sent next.

[4] Remember, the radix tree returns the most specific entry, so if a cloned route was already created for the address in question, the already-cloned entry will be returned and no further cloning done.

[5] We mean the computer science definition. This does not necessarily hold for e.g. bay leaves.

Keeping the existing BSD semantics was a priority, so we modeled the new routing algorithm to what the network stack used to do. The analysis lead to the following algorithm for routing a packet:

- If the target is a multicast address, send to the destination using a previously configured multicast IP address as the source.
- If we have an interface configured for the target address, send the packet to the loopback interface.
- If we have a point-to-point interface with destination address the same as the target address, send through the point-to-point interface to the target address.
- If we have an interface with the target address on the local network, send through that interface to the target address.
- Else, target the packet to the default gateway if one is configured.
- Otherwise, the target packet is discarded as being undelivarable.

Extracting from the above description, we need some configuration information to be able to operate:

- The configuration information for network interfaces
- The default gateway IP address
- Multicast interface information

4.2. Route Caching and Packet Forwarding

The old code does routing already on the transport layer and caches a `struct route` in `struct inpcb`. This is done so that lookup could be done for a certain connection once and then used thereafter. The other reason is source address selection: the packet destination must be known so that a source address for the packet can be selected.

This leads to some complications. First, for packet forwarding, the information available from the PCB is unavailable since the packet is not going through the transport layer at all. This means that the IP output routine must check if it has a valid route or not and do routing if it find it was passed a NULL route. Note, that source address selection will not be a problem for routers, since the source address is already always present in the IP packet.

Second, the cached information is not always correct. It is possible to send packets to multiple different addresses from UDP sockets. Therefore, the cached information in the route structure much be verified each time cached route use is attempted and a relookup done if the cached target did not match the target at hand.

Since we do not concern ourselves with packet forwarding, the problem becomes slightly simpler: we can always pass a valid route information to the `ip_output()` routine and do not have to do routing there any longer. However, we cannot easily move routing completely to the network layer, because we need the local address to select the right `inpcb`. It would clarify the structure greatly, though.

4.3. Multicast Addresses

Using multicast addresses has some special treatment within the BSD networking stack. For receiving packets, it must be possible to join and part multicast groups and to check if we belong to a multicast group a packet was received for.

We already mentioned earlier that the routing algorithm explicitly checks for a multicast destination and instead of using the default gateway uses the multicast address as the nexthop destination. The ARP resolution routine then translates the multicast address to the respective ethermulti address and it the packet is handled on a multicast capable router on the target network. This is what the old BSD code also did, including manually checking for multicast destination and possibly overriding the routing code decision to send the packet to the default gateway.

For selecting the source interface we use the same method as the BSD code. It is possible to set a system-wide default multicast output interface, although we do not currently provide a mechanism to set it on a per-socket basis.

4.4. ARP

ARP, on the fundamental level, is a translation service. An address of some format goes in and an address of some other format comes out. In our case these are the IPv4 address of the packet nexthop destination and the Ethernet address of the destination. Some caching is also necessary to avoid doing lookups every time a packet is sent.

We chose to implement ARP just as is described above: a very simple lookup database. Due to the removal of the radix tree code, we

could not use the old solution, where the ARP cache was kept in the radix tree as well. Instead, we store all the addresses in a linked list. This was mostly an accident, as any other data structure would have worked much better[6]. The difference is that the ARP module no longer has any knowledge about our network-level "routing" module. Its operation is entirely driven by the ethernet output routine, which tries to resolve the link layer addresses of packets as they are being sent.

The main interface for making ARP queries, `arpresolve()`, still exists in our new implementation. It either returns the translated address queried for or fires off a query for the address and creates a new embryonic ARP table entry. If a reply arrives, the embryonic ARP table entry is filled out. Since the old ARP code used the route expiration functionality, we had to implement similar timers in the new code. The `arp_rtrequest()` function, which had a clear routing socket bias, was replaced with `arp_addentry()` and `arp_delentry()` implementing functionality evident from the designation.

5. Results & Analysis

To recall, our target was to have equal functionality for leaf nodes with similar performance and significantly reduced code size.

5.1. Features

Our routing algorithm described in Chapter 4 is able to route packets from a leaf node to the local network(s) and the default gateway. It supports sending and receiving multicast packets and can join and depart multicast groups. This retains all of the features necessary for us from the old BSD routing code.

5.2. Code size

We analyze the code size savings by comparing the before and after size for the networking stack (without TCP) on our target platform.

The compiled ROM code size is almost 14kB smaller. Represented as a percentage, the old code is over 22% larger than the new code. And as a comparison, the compiled size for the TCP code is 13276 bytes. Therefore, using naive logic, the savings from the routing code rework enable the inclusion of the TCP code where it was not earlier possible due to size constraints. Of course the application using networking would produce extra overhead here, so this is not a completely accurate conclusion.

[6] The problem with lists is that one tends to write code using them even without thinking ...

Before/After Compiled Size

segment	before (bytes)	after (bytes)
text	77164	63160
data	1316	???

5.3. Performance

To measure the performance differences between the old and new code, a program which sends a UDP packet to several consecutive IP addresses was devised. Every UDP packet sent will cause a routing lookup because the target address does not not match the cached address.

We tested two cases: sending packets to the local network and sending packet to outside of the local network, i.e. sent to the gateway. All tests were executed while running Microsoft Invisible Computing as a regular process on top of Windows XP.

Performance Measurements

test	old (seconds)	new (seconds)
local 1	8.98	13.84
local 2	21.38	42.14
remote	238.36	245.73

The test "local 1" measured sending a packet to 128 addresses on the local network 16000 times, while "local 2" sent a packet to 256 different addresses 16000 times. The test "remote" sent a packet to 67108864 (0x4000000) addresses outside the local network. The stack was modified to only do routing, not actually send the packets, for the duration of these tests.

For the local network case, performance difference and be attributed to the difference between the computation complexity of radix tree and the O(n) performance of the linked lists of the new ARP code. If the latter was replaced with an O(log n) structure, performance would be equivalent.

For the remote case, performance appears to be the same and the difference in measurements can be attributed to noise because of running Microsoft Invisible Computing as a process. This is not surpising, since for both the new and old versions routing on a leaf node is pretty much a NOP.

5.4. Implementation

The current implementation is extremely intrusive and it is not possible to support the old and new routing code decided by a compile-time option[7]. This is mostly due to the implementation of the original routing code, its involvement in many different places in the networking stack, and it leaking too much of its implementation out. While it is not a tempting idea to be restrained only to leaf nodes, it is on level with the current requirements for the networking part of the system.

6. Conclusions and Future Work

We presented a method for implementing routing on IP leaf nodes without implementing an IP routing table at all. The key was to explicitly teach the code what kind of routing we want instead of using routing table entries to express the same information. The benefit was huge code savings due to not having to express rules on an abstract level when they could be expressed on a concrete level.

The routing code presented works for leaf nodes, but left a more general approach to be desired. Support for general-purpose routers should be investigated as an alternate implementation. It may not be worthwhile to carry the old routing code along at all, but rather to start from a clean slate or the work presented here.

We identified two places within the networking stack for a use of a database: IP routing and ARP translation. These were previously implemented using the same database, the radix tree. However, there are many more places within a kernel that could benefit from a more general database type of component instead of relying on ad-hoc data retrieval structures, which more often than not happen to be linked lists[8]. Even a simple, lightweight data storage algorithm, such as A-trees [7], would provide better options for generic data storage and retrieval when readily available in all BSD operating systems.

Finally, routing could be considered a more general-purpose component; there is no need to artificially separate OS and application routing.

[7] Of course, it would be possible to write a script that would wrap all the "-" lines from the diff behind `#ifdef OLDROUTING` and the "+" ones behind `#ifndef OLDROUTING`, but that would be an utter maintenance nightmare and therefore practically impossible.

[8] We experienced this shameful phenomenon with our ARP table implementation.

Acknowledgements

We wish to thank Alessandro Forin for inspirational and insightful conversations and Sasha Nosov for motivation.

Further Information

Microsoft Invisible Computing, including the routing implementation described here, is available as source code under the Microsoft Shared Source License from the website: `http://research.microsoft.com/invisible/`

References

1. Johannes Helander and Alessandro Forin, *MMLite: A Highly Componentized System Architecture,* pp. 96 -- 103, Eight ACM SIGOPS European Workshop (1998).
2. Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *Design and Implementation of the 4.4BSD Operating System,* Addison-Wesley (1996).
3. Gary R. Wright and W. Richard Stevens, *TCP/IP Illustrated, Vol2.,* Addison-Wesley (1995).
4. Keith Sklower, *A Tree-Based Packet Routing Table for Berkeley UNIX,* pp. 93 -- 99, USENIX Association Conference Proceedings (1991).
5. Robert Sedgewick, *Algorithms in C,* Addison-Wesley (1990).
6. André Oppermann, *Optimizing the FreeBSD IP and TCP Stack,* Fourth European BSD Conference (not in proceedings) (2005).
7. Alistair Crooks, *The A-Tree - a Simpler, More Efficient B-Tree,* pp. 185 -- 201, Proceedings of the 3rd European BSD Conference (2004).

How The FreeBSD Ports Collection Works

Mark Linimon

How New Ports Are Created

Whenever a FreeBSD user finds an application out on the Internet that they find interesting, often the first thing that they do is to try to create a 'port' for it. (In FreeBSD terminology, a 'port' is the files that allow the application to be downloaded and built directly from the Internet. These files include a Makefile, which includes a one-line summary of the port's function, location of the application sources on the Internet, and installation, deinstallation, and dependency information); a 'distinfo' file that contains meta-information about the distribution file name and security checksums; a 'pkg-plist' that is a list of files that will be installed; a 'pkg-descr' that gives a longer description of the port's function; and, optionally, other files that include patches to make the application install and run correctly on FreeBSD, and startup and shutdown scripts to help automate any daemons that are installed. Furthermore, in FreeBSD terminology, a 'package' is the resulting pre-built binary, if such a thing is allowed by the author's license.)

Not only is a port the easiest way for an individual user to install an application, it is also the most convenient way for any user to install the application. The only real difference between a one-time use and a completed port is that extra work needs to be done to make sure the port completely deinstalls itself when requested. The goal is that users should be able to feel free to test individual ports to see if they are useful for them, without being concerned that they will clutter up the system with stale files on deinstall.

Users generally create ports on the i386 architecture, although FreeBSD is seeing increased interest from users in running amd64 in native mode. (For purposes of the Ports Collection, the Intel 64-bit chips that are compatible with the amd64 architecture extensions, are also considered to be 'amd64'.) Users tend to have the latest mainstream FreeBSD release (this is currently 6.1-RELEASE, although 6.2-RELEASE is imminent as of this writing). Support for older releases (in particular, at this time, 5.5-RELEASE) is desireable but a secondary priority.

How New Ports Are Submitted

New ports are submitted via the same mechanism as that used to report errors, updates, or other matters: the Problem Report (PR) database. The database that FreeBSD currently uses, GNATS, is fairly primitive, so some add-on tools have been created to provide other functions. The FreeBSD Ports Monitoring System ("portsmon", at http://portsmon.FreeBSD.org) watches the PR database entries for variations of the string "new port" in the Synopsis, and keeps track of those on a special page. Users who don't use portsmon can still see the PRs coming in as they are echoed to the freebsd-ports-bugs@FreeBSD.org mailing list, or browse them via a web interface.

Procedures For Handling Problem Reports About New Ports

Any FreeBSD ports committer can take an interest in the new port and assign it to themselves in GNATS. (A 'committer' is the FreeBSD term for an individual who has privileges to commit to the source repository.) It is the committer's responsibility to ensure that the port fetches, compiles, installs, and deinstalls properly, and also has no security problems. (While it is not possible for FreeBSD, as a group of volunteers, to vouch for the security of any application, it is a goal that we try to reach; security considerations are discussed further below).

Our priorities for ports to function correctly are (roughly):

FreeBSD 6.X on i386
FreeBSD 5.X on i386
FreeBSD 6.X on amd64
FreeBSD 7.X (development version) on i386
FreeBSD 4.X on i386
other combinations of major release and architecture (including sparc64)

However, since everyone is a volunteer, these are only guidelines.

If, during testing, a committer finds a problem, he or she should reply to the original PR with a question for the submitter, and change the state of the PR to 'feedback'. In some cases, it takes multiple rounds of feedback to get the port ready for committing.

Once the commit is done, the committer is also responsible for hooking the port up to the build and adding other metadata that the Ports Collection needs to know about (e.g. the 'modules' file). In addition, he or she is also responsible for making sure that the build process for the entire port dependency generation is not broken. This command, 'make index', is used to generate a flat-file of metadata (including the dependency information) used to speed up the automated installation/deinstallation tools.

A periodic process on one of the main FreeBSD machines will catch errors in the index build, and post them to the freebsd-ports-bugs@ mailing list (together with a list of the most recent committers).

There is one further note to make about new ports. A guideline has been introduced within the past year that all new ports must have a listed maintainer to be accepted into the Ports Collection. What had happened in the past, when this was not the case, is that the number of unmaintained ports was growing far too quickly. (Note that an "unmaintained" port can still be updated by any interested contributor; but, the intention is that as much as possible, there should be at least one person willing to take a sufficient interest in the port to try to help keep it useful.)

Any committer who agrees to handle a 'new port' PR is expected to either handle it in a timely fashion, or return it to the general, unassigned, pool so that someone else may take it up. While there is no fixed period for this, it should be within a few weeks.

A few ports also install scripts to run daemons at system startup time. These scripts need to be installed into a localized etc/ directory, so the port must also handle installing these files.

The philosophy of the build tools is to not enable the daemon on install or disable the daemon on deinstall; the user is warned that these steps need to be taken manually. (The current belief is that there are too many edge conditions to correctly test for, such as, updating a port in place and possibly having the update fail, and so forth).

Some ports are different versions of existing ports, and these may require special handling. For instance, if version N+1 is incompatible with version N, and a great deal of other software depends on version N, then it may be best to maintain two separate ports. In this case, to preserve the checkin history, FreeBSD performs what we call a "CVS repo-copy" for "repository copy". Although FreeBSD is currently evaluating alternatives to CVS, this is a longer-term project. With CVS, copying all the files in the repository to a new location is the only way to preserve the history.

The preference is to do a repo-copy, rather than an entirely new port with no history, if version N+1 will eventually replace version N. However, if version N+1 is a complete rewrite, it may not make sense to preserve the history. This is evaluated on a case-by-case basis.

After a repo-copy, the committer handling the PR is expected to do a forced checkin to note the original location, and the reason for the split. There are other technical changes that should done at this time (portname, distfile information, and so forth) before hooking it up to the builds.

In general two different versions of ports cannot be installed simultaneously, since they usually use the same names for files that they install. A mechanism in the ports framework entitled CONFLICTS tells the installation tools not to allow this collision.

In some cases version N+1 is a "development" version of a piece of software which is intended for developer debugging only. These ports are similar to the above but by convention take on a "-devel" suffix in the portname.

Maintainance of Ports

Once a port is added, the easiest part of the work has been done. Applications are constantly in flux, and updates to the various files need to be made, tested for their ability to install, tested for their ability to actually run, and tested for their ability to deinstall cleanly. In addition, changes to the base system in -CURRENT can affect the ability of ports to install and run correctly. Finally, there are several processor architectures on which we want the ports to run.

We can divide these tasks into:

- Build Failures and Problem Reports
- Port Updates
- Procedures To Mark Ports "Broken" And Procedures To Mark Ports That Have Reached Their End-Of-Life

Build Failures and Problem Reports

For each build environment (processor architecture * major OS release), test builds are continually run on a set of dedicated machines known as the "pointyhat cluster" (after the name of the machine that dispatches all the jobs. The mythical "pointy hat" is the virtual dunce cap awarded to FreeBSD committers who commit someting to the source tree that doesn't work, especially if it doesn't build in the first place.)

Whenever the build cluster notes that a port has failed to fetch, compile, install, or deinstall correctly, some kind of action needs to be taken. If the port is maintained, email is sent to the maintainer including either the entire build log, or a pointer to it. (This process is not completely automated). If the maintainer is able to duplicate, and then fix, the problem, they are expected to file a Problem Report (PR) containing the fix. If not, they can ask the community for help (either via mailing lists or IRC).

If the problem cannot be isolated, or the port is unmaintained, the port is marked BROKEN. This will both prevent the build cluster from attempting to build the port over and over again, and prevent users from being able to install the port (a warning is issued, and the install exits). A make(1) variable, TRYBROKEN, exists in the ports framework to override this setting.
For instance:

```
make TRYBROKEN=yes install
```

Users may also file Problem Reports against ports that pass the automated tests, but fail to run properly, asking that they be marked BROKEN. Although some ports do indeed contain their own regression tests, in general these are fairly rudimentary. We rely on our users to advise us whether the ports actually work and are generally useful.

There is a policy that we give maintainers 2 weeks to respond to bug reports, requests for updates, and so forth. After that time, any committer may step in and make the change at their own discretion by invoking a "maintainer-timeout". That duration has been chosen to try to strike a balance between not infringing on a maintainers' prerogative to maintain a port as he sees fit, and the needs of users to have their problems addressed in a timely fashion. Maintainers who do not respond to PRs within 3 months may lose the maintainership of their ports.

In general, we intend that PRs address only the problems involved with running a particular application on FreeBSD (rather than a problem with the application itself). On occasion we may recommend that a user needs to open a ticket with the author directly. However, if a maintainer finds a bug that affects the application everywhere, we encourage sending the patch back upstream to the author to try to help the larger user community. (This often happens when a new version of gcc is imported to the base system, for instance). In general, this produces good results, but not always.

We also encourge maintainers to send the patches that are needed to adapt the application to FreeBSD upstream to the authors; in some cases they will accept these patches, but in others they will not, and the patches must remain local to the FreeBSD ports tree.

Port Updates

Software authors are continually updating their software, for reasons including:

- new features
- fixes for security problems
- fixes for other bugs
- changes necessitated by updates to other software that they depend on
- to keep it running as base-systems (kernels, drivers, etc.) change
- to port it to new OSes and architectures
- changes or additions to documentation

The simplest changes only involve a modification to the port Makefile and the distinfo file to represent the update. However, it is the responsibility of the port committer to analyze what's changed before committing: simply saying "distfile has been rerolled" isn't sufficient. This policy is in place to help guard against trojans being inserted into distfiles. However, most changes do fall into this category.

More extensive changes may also require changes to the "packing list" which is used to specify what files to remove on deinstall. Traditionally, the packing list was specified in the 'pkg-plist' file, but for simpler ports it can now be specified in the Makefile.

Unlike OpenBSD and other systems, there is no "staged install" where this step is taken care of automatically. This can be considered a bug. (To solve the general problem is fairly hard).

There is currently no perfect system for a port maintainer to be notified when a port update is available. The two most promising solutions are currently ports/newportsversioncheck by Edwin Groothuis, and Shaun Amott's portscout, the results for which are available at beta.inerd.com/portscout. newportsversioncheck allows you to install software on your machine which will scan through the download pages referenced by a port, and search for strings that appear to be newer distfiles than the current one. The portscout application has a similar algorithm but only presents its results.

Since there is no accepted way to catalog the meta-data for an application (e.g. download location, distfile versioning scheme, dependent applications, and so on), these heuristical approaches are the best ones we have right now. (The current author also has an alpha-quality implementation, and can attest to how hard the general problem is).

The disadvantage to all these methods are that they are a "pull" methodology instead of a "push" one. The long-term solution to the problem is to create a standard for publishing updates (e.g. via RSS) and have the ability for port maintainers (as well as the various monitoring programs) to subscribe to them. In fact, there are sites for which RSS feeds are available for individual ports, but without some kind of aggregation function, the "pull" methodology currently works best.

The current author doesn't know the percent coverage that these two tools have, but was

surprised to find that even his own naive algorithm was correctly identifying over 70% of the possible port updates.

Procedures To Mark Ports "Broken" And Procedures To Mark Ports That Have Reached Their End-Of-Life

Ports that remain marked BROKEN for a period of time will be marked with two more makevars, DEPRECATED and EXPIRATION_DATE. These are advisory variables only. However, portsmon sends a report every 2 weeks listing all the ports with an EXPIRATION_DATE, and includes, among other data, the location of the latest build error; any PRs against the port. In this fashion we hope to provide "fair warning" for any user who relies on that port to take some kind of action (e.g. help fix the problem; find another application; or make a local copy of the last-known working version that they have installed, if nothing else).

An intended side-effect is that the community feels involved in this process; rather than stale ports being deleted with no notice, there is now a way for everyone to see that process as it happens.

Every few weeks, a ports commiter will go through the list and delete the expired ports (including adding them to a file called /usr/ports/MOVED, which tell the automated tools that the port has been removed). This, in turn, will cause the port to be removed on a user's machine during the next run of the automated tool.

Removed ports stay in the MOVED file permanently (unless they are at some point reinstated, at which time the entry is removed).

There is no one particular ports committer that is responsible for this process; after the expiration date, anyone may remove the port.

The expiration dates are not intended to be cast in stone; they can be extended if someone is still actively working on a fix. The intention is simply to reap useless ports out of the Ports Collection, so that users do not go through the work of installing a port that doesn't work. This is both frustrating for the users, and detrimental to the reputation of the Project.

Support Resources for People Who Want To Contribute

The main method for distribution of information and sharing ideas is the freebsd-ports@FreeBSD.org mailing list. General discussion, questions, and proposals are all welcome here. The experience range is novice to expert. Here you will also find various HEADS-UP messages about changes to the Ports Collection that most users need to be aware of. (This information is also expanded upon in a file named ports/UPDATING; all users should track changes to this file. For instance, if the location or format of a configuration file changes, this is where that will be documented.)

On occasion, a notice about some particularly important change will be sent onto the moderated freebsd-announce@FreeBSD.org mailing list. All FreeBSD users should be subscribed to this low-volume list.

The mailing list freebsd-ports-bugs@FreeBSD.org echoes the contents of ports PRs as they come in; further discussion about those particular problems also takes place here. It is a fairly high-volume list due to the large numbers of PRs (sometimes more than 40 per day).

Sufficiently interested people may also subscribe to the cvs-ports@ mailing list to follow all the port commits (and, sometimes, ensuing discussion). This is very much like drinking from a firehose, so be warned.

Many of the FreeBSD ports committers also contribute to an IRC channel on EFNet (name upon request :-)) While not moderated, this channel stays pretty close to the topic, and in particular, technical questions about "what's the best way to solve the following problem?" In this channel, people who are interested in becoming ports committers can learn some valuable information from those that already are. The experience range is advanced to expert.

There is a FreeBSD wiki, intended mostly for use by developers, at http://wiki.FreeBSD.org, that also contains some status information about work-in-progress, and ideas for future changes. Although incomplete, it is nevertheless a valuable resource.

The most interesting places to learn more about the Ports Collection are:

- FreeBSD Porter's Handbook (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/porters-handbook). This is the most complete technical reference outside of the ports/Mk files themselves. This document is large and dense and few people are able to understand it all at first. The information encompasses both hard-and-fast rules ("you must not break the INDEX build") as well as "recommended best practices" as advocated by the portmgrs. There is also some information about "procedures" here, although arugably it should be in the next document.
  The intended audience is port creators, maintainers, and committers; although there is some information aimed at end-users, it is not well separated from the above.

- FreeBSD Committer's Guide (http://www.FreeBSD.org/doc/en_US.ISO8859-1/articles/committers-guide).This is oriented towards those who have commit rights to the repository, and as such may be of less interest to users and maintainers. It contains both technical information about making commits, and documentation of procedures.

- FreeBSD Ports Managment Team (portmgr) web pages (http://www.FreeBSD.org/portmgr). This documents both some technical background of the Ports Collection (especially about the fact that the tree is not branched, and the implications thereof); what the portmgr team actually does, both during release cycles and between them; and explanation of rules such as exactly how long a maintainer-timeout is.

- Contributing To The FreeBSD Ports Collection (http://www.FreeBSD.org/doc/en_US.ISO8859-1/articles/contributing-ports) is oriented at users and maintainers to suggest how they can help to become involved

with the work of keeping the Ports Collection functioning and up-to-date.

- FreshPorts (http://www.FreshPorts.org) is a website that allows anyone to browse the state of the Ports Collection, search for applications, view the state of the port, read through checkin messages, and even subscribe via email to notification of port updates. All port maintainers are encouraged to subscribe. This system is written and maintained by Dan Langille as a personal project and is not under the auspices of the FreeBSD Project itself.

- The build error system is located at http://pointyhat.freebsd.org/errorlogs. The raw error logs are posted there under various buildenvs, with URLs that point to the logs themselves.

- portsmon (http://portsmon.FreeBSD.org) is the site that allows users to correlate the build errors with Problem Reports and other metadata about individual, or groups of, ports. It is written and maintained by the current author.

How To Help

- FreeBSD is always happy to have new ports contributors. Within the past 2 years, a large backlog of problem reports for existing ports, and a similarly large backlog of those for new ports, has been greatly reduced. In that time we have added around 2 dozen new port committers. However, there is always more to do:

- There are currently almost 16,000 ports in the collection. Over the years, a large number of these (4,000+) have become unmaintained. This does not necessarily mean that they are abandoned; many contributors and committers make periodic sweeps to make sure that as many ports as possible still fetch and build. However, having a maintainer almost always guarantees that a port is in better shape than otherwise. We encourage FreeBSD users to take a look at the above documentation and see if they can contribute back to the project by taking on one or more ports to maintain.

- Another focus needs to be getting the state of the ports on native amd64 closer to parity with i386. Although this will never be perfect since some ports include i386 binaries, there are still several hundred ports (in particular, ones more oriented to the desktop than the servers) which need to see wider use and greater testing.

- FreeBSD is currently evaluating the latest gcc, and past gcc version updates have shown that many ports will need to be modified. (In the past, FreeBSD has been among the first to adopt new gcc versions, which has led to a large number of patches being sent back to the authors). The latest experimental run showed over 1,000 ports that became broken. Several committers are looking through this backlog, but we need more.

- There are always more PRs in the backlog that need to be addressed. The recent low was 500 PRs (divided up into 1/2 existing ports, 1/4 infrastructure, just less than 1/4 new ports, and a handful of miscellaneous problems). To get there took a concerted effort from a large number of individuals. However, since then the steady-state

number has drifted up closer to 800. (It is always higher during release cycles).

- Since many FreeBSD users build all their ports directly from source, the state of the binary (pre-built) packages often lags behind. The current author is working on some tools that will help to identify problems in keeping the binary packages up to date in between releases to try to bring them closer to parity with the source. Having more users use, and thus report on problems with, packages would be helpful.

Design and Planning an AFS Cell

Fabrizio Manfredi

fabrizio.manfredi@gmail.com

Contents

ABSTRACT

This paper describes the Andrew File system and the best practices for setting up a new cell. Finally a case study will be given for evaluation in real world on BSD platform.

INTRODUCTION

Distributed file systems enable users to work in distributes computing in office/engineering environments. Their utility is obvious, they enhance information sharing among users, facilitate parallel processing and simplify the administration of large numbers of machine. In many cases, other services as electronic mail, printing and content delivery are layered on top of this system.

Andrew File System (AFS) Overview

Andrew File System is a distributed file system and was designed to handle terabytes of data and thousands of users distributed across large networks, the AFS works as a Unix and Windows NT add-on and replaces the usage of Network File System (NFS). Today AFS has a solid presence in very large commercial networks.

History

Over the years, AFS has developed a rich and interesting history, In the past The Andrew File System heavily influenced Version 4 of Sun Microsystems' popular Network File System (NFS). Additionally, a variant of AFS, the Distributed File System (DFS) was adopted by the Open Software Foundation in 1989 as part of their Distributed computing environment. Some important date:

1983 Andrew Project started at Carnegie Mellon University (CMU)
1987 Coda research work begun (based on AFS)
1988 First use of AFS version 3 First use of AFS outside Carnegie Mellon University
1988 Institutional File System project at University of Michigan - ports of AFS to mainframe, intermediate servers, disconnected operation, performance/security enhancements
1989 Transarc Corporation founded to commercialize AFS, formed by part of original team members
1993 Arla project started at Kungliga Tekniska Högskolan
1998 Transarc Corporation becomes wholly owned subsidiary of IBM
2000 IBM releases OpenAFS as OpenSource (IBM License), run at the Department of Computer Science at Carnegie Mellon University.
2000 OpenAFS release version 1.0 based on Transarc 3.6
2001 OpenAFS release version 1.2 first release with better support of new operating system and fix several memory leak
2005 OpenAFS release version 1.4 with a lot of new feature
2005 AFS was discontinued from IBM

The first three version of AFS were developed at Carnegie Mellon University (CMU), today the Andrew Consortium governs and maintains the development and distribution of the Andrew User Interface System and give good support for lot's of platform. A very important implementation for *BSD come from Arla, that have also a distinguish feature in Disconnected Operation.

DESIGN

AFS was deigned to serve the filing needs of the entire CMU campus. Each users was expected to eventually have their own workstation, implying a scale of nearly 10.000 nodes. This was at least one order of magnitude larger than any distributed file system built or conceived of at that time. Not surprising, the scale of the system became the dominant consideration with security ,transparent access and management.

Scalability

The designers used a client-server architecture to implement Scalability goal. AFS provides location independence that scales widely and stores and retrieves data transparently across a network of many computers. Files in AFS are as accessible as any stored locally file system on a personal computer's hard drive. In fact, AFS stores files on a subset of the machines in a network, called file server machines. File server machines provide file storage and delivery service, along with other specialized services, to the other subset of machines in the network, the client machines.

- **Client Caching**
  AFS uses client side caching to improve global efficiency. Caching improves efficiency because the client does not need to send a request across the network every time the user wants the same file. Network traffic is minimized, and subsequent access to the file is especially fast because the file is stored locally. AFS has a way of ensuring that the cached file stays up-to-date, called a callback. All these operations are made by Cache Manager . The Cache Manager reside on client and determines file location automatically and puts it into the cache (an area of the client machine's local disk or memory dedicated to temporary file storage), only when the file is saved does the Cache Manager send changes back to the server.

- **Replication**
  AFS enables administrators to replicate commonly-used data (volumes), such as those containing binaries for popular programs. Replication means putting an identical read-only copy (sometimes called a clone) of a volume on more than one file server machine. One benefit of replicating a volume is that it increases the availability of the contents. If one file server machine housing the volume fails, users can still access the volume on a different machine transparently to users' work, . Replication permit also a load balance on data access, in this way a server does not become overburdened with requests for common files (volume access).

Security

The distributed control of machines, widespread access to the network, and relative anonymity of users make security one of the major concern at large scale. AFS addresses the security problem in three ways:

- **Physically security of server machine**
  Physically security is ensured by keeping servers in protected rooms and running only trusted system software on them.

- **Authentication and secure communication**
  Secure connection and authentication phase are arranged through a variant of original protocol of Kerberos IV (), today is possible to use more secure Kereberos V System. AFS requires mutual authentication between servers and clients whenever they communicate with each other. Tokens that pass between them must be recognized as the valid tokens of authenticated users before files

can be accessed and services and software delivered.

- **Authorization and flexible access control**
  AFS uses access control lists to determine who accesses information in AFS file space. An ACL exists for every directory and specifies what actions different users can perform on that directory and its files. Users themselves control another aspect of AFS security, by creating personal group and build personal ACL, all that makes it easy for groups of people to share information and work together in the same directories (folders), no matter where they are located or platform used. System administrators can also create groups containing client machine IP addresses to permit access when it originates from the specified client machines.

Transparent Access and the Uniform Namespace

Like UNIX, AFS uses a hierarchical tree-like file structure (fig.1). Users logged in and authenticated to AFS will see the /afs root directory in all of their file and directory paths (on windows is mapped on to disk letter). AFS enables cells to combine their local filespaces into a global filespace, and does so in such a way that file access is transparent, users do not need to know anything about a file's physical location in order to access. AFS provides all that with three major component:

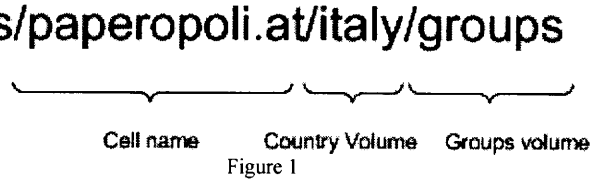


Figure 1

- **Cells**
  The directories under /afs are cells. Cells are independently administered sites running AFS and consist of a collection of file servers and client workstations defined as belonging to that cell. A computer can belong to only one cell at a time. A cell's administrators determine how workstations and servers are configured for that cell and how much storage space is available to each user, software package, project locker, etc. Each cell can also connect with the file space of other cells running AFS.

- **Partitions and Volumes**
  The storage disks on a server are sectioned into disk partitions. These partitions are further divided into volumes, which are "containers" or sets of related files and directories and forming a partial subtree of the namespace. Volumes have a size limit, or quota, assigned by the system administrator, but are (by definition) smaller than a partition. The AFS volumes, making it possible to distribute files across many machines and yet maintain a uniform namespace. Volumes are important to system administrators that can maintain maximum efficiency by moving volumes to keep the load balanced.

- **Mount Points**
  Access to a volume is provided through a mount point, which indicates the physical storage location of a volume on a server. Your own volume resides on one of many file servers, and the mount point is the pointer that AFS uses to find and retrieve it for you. A mount point looks and just like a static directory to the user, but it actually navigates the file system and servers to store and retrieve data transparently and automatically.

System Management

Establishing the same view of filestore from each client and server in a network of systems is useful to simplifies part of the systems management workload. Systems administrators are able to make configuration changes from any client in the AFS cell (it is not necessary to login to a fileserver). With AFS it is simple to effect changes without having to take systems off-line. Administrator have also a powerful tool for backup operation, they can create a backup volume version to preserves the state of a read/write source volume at a specified time (snapshot). You can mount the backup version in the AFS file space, enabling users to restore data they have accidentally changed or deleted without administrator assistance.

ARCHITECTURE

AFS is composed from servers and client. On the server side we have server machines run a number of processes, each with a specialized function called server processes, on the other hand Clients do not run any special processes per se, but do use a modified kernel that enables them to communicate with the AFS server processes running on the server machines and to cache files.

Server Machines

Server machines store the files in the distributed file system, and a server process running on the file server machine delivers and receives files. This modular design enables each server process to specialize in one area, and thus perform more efficiently. Not all AFS server machines must run all of the server processes. Some processes run on only a few machines because the demand for their services is low. Other processes run on only one machine in order to act as a synchronization site. We can identified four principals server machines:

- A simple file server machine runs only the processes that store and deliver AFS files to client machines. You can run as many simple file server machines as you need to satisfy your cell's performance and disk space requirements.

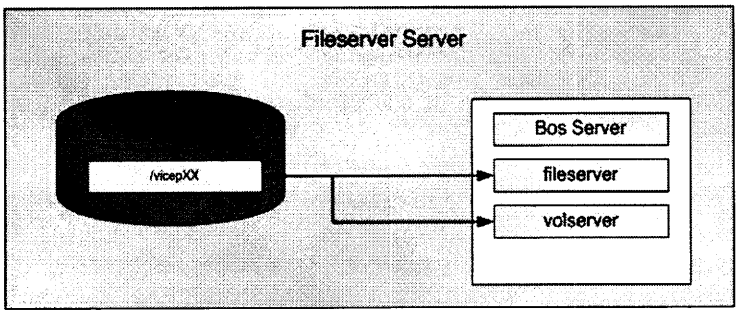


Figure 2

- A database server machine runs the four database server processes that maintain AFS's replicated administrative databases: the Authentication, Backup, Protection, and Volume Location (VL) Server processes.

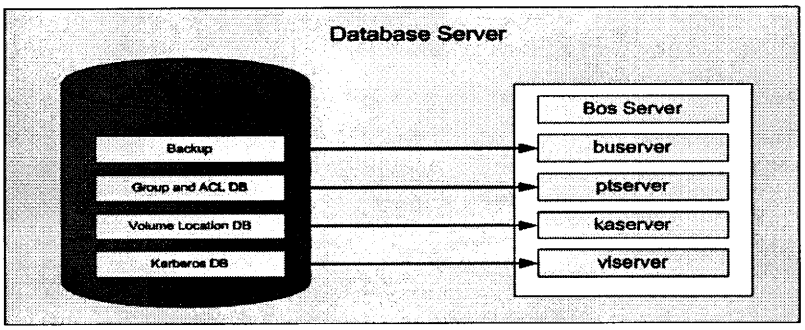


Figure 3

- A binary distribution machine distributes the AFS server binaries for its system type to all other server machines of that system type.
- The single system control machine distributes common server configuration files to all other server

machines in the cell. The machine conventionally also serves as the time synchronization source for the cell.

All this function could be hosted on the same system, because the server machines identification is based on the server process.

Client Machines

Client machines, provide users access to the files stored on the file server machines, as already mentioned the Cache Manager is the one component in this section that resides on client machines rather than on file server machines. It is not technically a stand-alone process, but rather a set of extensions or modifications in the client machine's kernel that enable communication with the server processes running on server machines. Its main duty is to translate file requests (made by application programs on client machines) into remote procedure calls (RPCs) to the File Server. (The Cache Manager first contacts the VL Server to find out which File Server currently houses the volume that contains a requested file. When the Cache Manager receives the requested file, it caches it before passing data on to the application program. The Cache Manager also tracks the state of files in its cache compared to the version at the File Server by storing the callbacks sent by the File Server. When the File Server breaks a callback, indicating that a file or volume changed, the Cache Manager requests a copy of the new version before providing more data to application programs.

Server Process

The following list briefly describes the function of each server process.

- **File Server**, the most fundamental of the servers, delivers data files from the file server machine to local workstations as requested, and stores the files again when the user saves any changes to the files .
- **Basic OverSeer Server** (BOS Server) ensures that the other server processes on its server machine are running correctly as much of the time as possible. The BOS Server relieves system administrators of much of the responsibility for overseeing system operations.
- **Authentication Server** (kaserver)helps ensure that communications on the network are secure. It verifies user identities at login and provides the facilities through which participants in transactions prove their identities to one another (mutually authenticate). It maintains the Authentication Database.
- **Protection Server** (ptserver)helps users control who has access to their files and directories. Users can grant access to several other users at once by putting them all in a group entry in the Protection Database maintained by the Protection Server.
- **Volume Server** performs all types of volume manipulation. It helps the administrator move volumes from one server machine to another to balance the workload among the various machines.
- **Volume Location Server** (VL Server) maintains the Volume Location Database (VLDB), in which it records the location of volumes. This service is the key to transparent file access for users.
- **Update Server** distributes new versions of AFS server process software and configuration information to all file server machines. It is crucial to stable system performance that all server machines run the same software.

- **Backup Server** (buserver) maintains the Backup Database, in which it stores information related to the Backup System. It enables the administrator to back up data from volumes to tape. The data can then be restored from tape in the event that it is lost from the file system.
- **Network Time Protocol** Daemon (NTPD) is not an AFS server process. It synchronizes the internal clock on a file server machine with those on other machines. Synchronized clocks are particularly important for correct functioning of the AFS distributed database technology (known as Ubik);

Ubik

Ubik is a distributed database. It is really a (distributed) flat file that you can perform read/write/lseek operation . The important property of Ubik is that it provides a way to make sure that updates are done once (transactions), and that the database is kept consistent. It also provides read-only access to the database when there is one (or more) available database-server(s). All servers in AFS use Ubik to store their data.

CONVENTIONS AND BEST PRACTICES

Step for planning a new AFS installation :

- AFS file space layout
- Server planning
- Volume naming and schemas
- Volume replication
- Username schemas
- Partition Filesystem
- Backup planning
- Security consideration
- Client Cache tuning
- AFS limitations

AFS file space layout

Your cell name is very important because distinguish your cell from all others in the AFS global namespace. By conventions, the cell name is the second element in any AFS pathname and follow the ARPA Internet Domain System conventions. The better choice is the company Internet domain name or kerberos realm if present.

- Max size cell name is 64 characters, but shorter names are better because the cell name frequently is part of machine and file names.
- Cell name can contain only lowercase characters, numbers, underscores, dashes, and periods to guarantee it is suitable for different operating system. Do not include command shell metacharacters.
- Cell name can include any number of fields, which are conventionally separated by periods

Server planning

In general, you need at least three database server machines for high availability (election algorithm is used for identify master database) More small Fileserver machine permit better spread the load and replicate/backup volume for a redundancy, this also permit to increase capacity on demand. Oldest documentation indicate as limit to client/server rate 200:1, today some cell use 1000 for server. Split database server as well as the fileserver is useful for performance and security.

Volume Naming and Schemas

One of the problem in AFS cell is to identified volume mount point, for these reason the first step is a volume naming convention. The naming convention must explain, with the name, what a volume contains and where it is mounted. Many cells find that the most effective volume naming scheme is to puts a common prefix on the names of all related volumes. (sample ?)

Volume name restrictions:

- Read/write volume names can be up to 22 characters in length. The maximum length for volume names is 31 characters, and there must be room to add the .readonly extension on read-only volumes.
- The .readonly and .backup extensions are reserved word The Volume Server adds them

automatically as it creates a read-only or backup version of a volume.

- The volumes named root.afs and root.cell are used for default (mounted respectively at the top /afs and for cell /afs/foo.com).

Mount point

Installation with more than a few hundred users sometimes find that mounting all user volumes in a single directory results in slow directory lookup. The solution is to distribute user volume mount points into several directories :

- Distribute user home directories into multiple directories that reflect organizational divisions,
- Distribute home directories into alphabetic subdirectories of the home directory
- Distribute home directories randomly but evenly into more than one grouping directory

Volume replication

Replication refers to making a copy, or clone, of a read/write source volume and then placing the copy on one or more additional file server machines. Replicating a volume can increase the availability of the contents. best practices is :

- Replicate the **root.afs** and **root.cell** volumes because the Cache Manager needs to pass through the directories corresponding to the **root.afs** and **root.cell** for reach all other volume.
- Replication is not appropriate for volumes that change frequently. The synchronization is manually with vos release command.
- User volumes usually exist only in a read/write version and each user home have its own volume, for simplify load balance operations (move volume)
- Sometime could be useful for backup volume use the same partition in this way changes substantially does the read-only volume consume significant disk space (it is a copy of the source volume's *vnode index*)

Partition Filesystem (inode vs namei)

AFS servers and clients use a Unix file system for low-level storage. The first implementation on server side was access files directly by i-node number, this type of configuration is very fast but its needs :

- Dedicated partition, isn't enough a simple directory for /vicepXX
- Special fsck for the system partition
- No journaling file system
- Restore on same filesystem layout (same inode structure)

In OpenAFS id possible decide the old method or a new one called namei, namei use Unix function that does pathname translation , this method has considerable overhead, advantage of namei are :

- OS fsck
- Filesystem independent, with advantage of journaling
- The aren't special requirement for /VicepXX, it could be a mounted
- Simply restore operation

Usernames

AFS associates a unique identification number, the AFS UID, with every username, recording the mapping in the user's Protection Database entry. Every AFS user also must have a UNIX UID recorded in the local system of each client machine they log onto, for these reason the best solution is LDAP/NIS backend for account information. One important consequence of matching UIDs is that the owner reported by the ls -l command matches the AFS username.

AFS imposes very few restrictions on the form of usernames:

- Characters, which have special meanings to the command shell
- The colon (:), because AFS reserves it as a field separator in protection group names;
- The period (.); it is conventional used to identify special username that have administrator capability (ex. manfred.admin)
- AFS UID, 32766, is reserved for the user anonymous.

Backup

Backup solution for AFS can be divided in three major classes:

- Native backup system and recovery, AFS can be configured to create a full or incremental backup
- Volume dump, this operation permit to create a binary file with all information of backup volume
- Backup system with AFS support for example Amanda or Bacula (and other commercial product)

Security consideration

Some possible to hardening of AFS should be :

- **User Accounts**:
  - Kerberos integration with modified login utility, OpenAFS support basic Kerberos 5 (2b protocol), replace kaserver with Unix Kerberos solution or Windows AD
  - including the **unlog** command in every user's **.logout** file or equivalent
- **Server Machines**
  - Change the AFS server encryption key on a frequent and regular schedule.
  - Particularly limit access to the local superuser **root** account on a server machine. The local superuser **root** has free access to important administrative subdirectories of the **/usr/afs** directory.
- **System Administrators**
  - Create an administrative account for each administrator separate from the personal account and assign AFS privileges only to the administrative account. Set the token lifetime for administrative accounts to a fairly short amount of time.

Client Cache

AFS client must have a cache in which to store local copies of files. To set up your AFS cache you must decide:

- **Cache Size**
  - single user machine 128MB
  - Multi-user machine 1GB/4GB
- **Cache partition**
  - Directory, the partition must grantee enough space
  - Disk partition, better performance (Terminal Server)

AFS Limits

The most serious limitation of AFS has been read/write data availability, in the latest release is present volume conversion from read-only to read write. Limits present today:

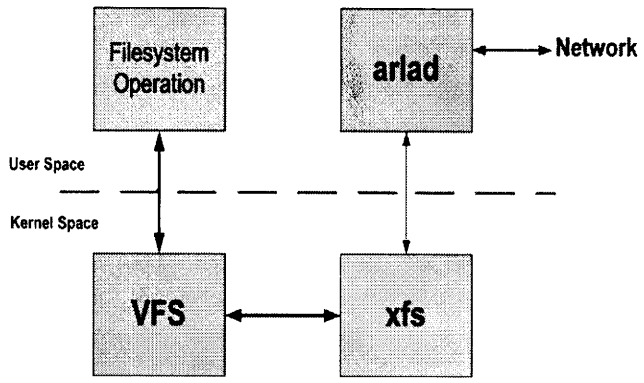
- OpenAFS can support a maximum of 104.000 clients per server, in general large cells do not exceed more few thousand clients per server.
- AFS does not support mandatory file locks, byte-range locking, only advisory locks are supported (work in progress)
- AFS does not support UNICODE file names
- ACL works only in directories,
- AFS does not allow certain type of file, like pipes, device files, sockets
- tmpfs no work as AFS Cache, (ramdisk work)
- AFS support max 255 partition per server (/vicepa-/vicepiv), no limits in partition size
- AFS support 4,294,967,295 volumes per partition (this a limit of VLDB), current limit of volume is 2TB
- Directories can hold a limit of 64,00 files per directories if the filenames are all less than 16 characters. The number decrease if the filenames are upper of 16 characters because there are 64.000 slots per directory each slot takes 16 characters.
- Write-on-close, the changes are synchronized only on close operation, for these reason vi create a temporary file
- No integration on Microsoft DFS
- No support for files greater than 2GB on windows platform.

ARLA

The original goal of Arla was to have an AFS free implementation. Another important goal is add support for platforms that don't have AFS support from Transarc or OpenAFS today seplcial way on *BSD platform. Important feature (and distinguish from standard version) of Arla is the disconnected-operation,.

Internals

Arla consists of two parts, a userland process (arlad) and the kernel-module (xfs). Arlad is written in user-space for simpler debugging (and less rebooting). To avoid performance loss as much as possible, xfs is caching data. xfs and arlad communicate with each other via a char-device-driver. There is a rpc-protocol currently used specially written for this. Theoretically, xfs could be used by other user-space daemons to implement a file system. Some parts, such as syscalls, are arla-specific. These parts are designed to be as general as possible.



SUPPORTED PLATFORM

The OpenBSD support AFS with Arla since release 3.4. Currently Arla 0.35.7 is supplied with OpenBSD 3.9 Release . OpenBSD is one of supported platform from stable release of OpenAFS, (version 1.4.X) Unfortunately on client side no other *BSD platforms are supported from OpenAFS project , NetBSD and FreeBSD must use Arla code (last is 0.43)

Official support of OpenAFS 1.4.X:

- AIX 4.2, 4.3, 5.1, 5,2, 5.3
- HP-UX 11i (pa-risc), 11.22 (pa-risc), 11.23 (ia64)
- Solaris 7 (sparc,x86), 8 (sparc,x86), 9 (sparc,x86), 10
- (sparc,x86,amd64)
- MacOS X 10.3 and 10.4 (ppc, intel)
- Microsoft Windows 2000, XP (x86, amd64), 2003 (x86, amd64),
- 2003 R2 (x86, amd64), Vista Beta 2 (x86, amd64)
- Linux 2.4 kernel: x86, x86-uml, amd64, ia64, pa-risc, ppc, ppc64,
- s390, s390x, sparc, sparc64
- Linux 2.6 kernel: x86, x86-uml, amd64, ia64, ppc, ppc64, s390x,
- sparc64
- OpenBSD (x86) 3.3, 3.4, 3.5, 3.6, 3.7, 3.8,3.9
- NetBSD (x86; server only) 1.5, 1.6, 2.0, 2.1, 3.0
- FreeBSD (x86; server only) 4.7, 5.3, 6.0-beta
- SGI Irix 6.5

CASE STUDIES

The "company" is nowadays the biggest Italian fully independent forwarding company covering any service related to transports and logistics with a worldwide agency network. It have Head Quarter (HQ) in Italy and 21 Branch Office worldwide. The primary operating system was Windows XX on 8 Windows NT Domain for a total of 550 users. The HQ and the branch office are connected with Wide are Network. They don't have IT stuff presence on the branch office.

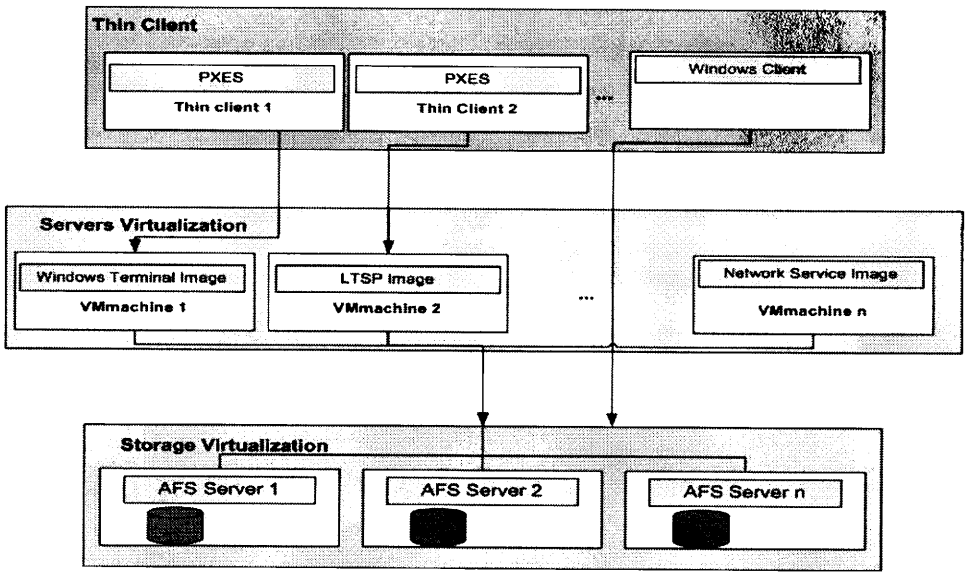
Business need

Primary goals of the project was to reduce cost of Software License and simplify System Administration task.

Solution

In design phase we identified two main actions: the first was the use of Free Software as desktop replacement for cost reduction and second was the centralization of user profile and system management for system administration task. The two areas was divided in three main steps :

- Thin client replacement. The usage of Windows Terminal Server and Linux Terminal Server Project (LTSP) gave a big license reduction. The thin client has PXE boot for simply the administration task
- Server Virtualization. The use of virtualization technology for run terminal server image and other network service reduced a downtime and increase a TCO (with usage of inexpensive machine). The choice for this step was VMware Server (free available) on Linux platform (CentOS)
- Storage Virtualization.All the services before explained need a transparent filesystem layout, with a good manageability and redundancy. The choice for this layer was AFS on OpenBSD platform.



Architecture

The architecture was divided in two type HQ and Brach Office. In HQ there were the great majority of users (350) and we placed all the main servers. In the biggest Brach Office we placed some server for better performance.

Head Quarter

- 3 Fileserver Machines with a 120:1 User:Server rate. The read-write information volumes are replicated with circular schema (the information on Fileserver n are replicated on Server n+1 and so on, the last server is replicated on the first server). The volumes of binary and programs are replicated on all fileserver. The fileserver are based on OpenBSD 3.9.
- 3 Database Servers installed on the same machine of fileserver, the database server do not include the kaserver, the authentication system is based on dedicated Kerberos machine.

- 2 Authentication Servers, on this machine are present Heimdal Kerberos with ldap backend, Samba for domain controller with ldap backend and Openladp server. The OpenLdap is used for authentication and profile information (account home directory).

- 8 VMmachine, the vmserver machine is a simple Linux installation with VMware Server, this machine can run 3 type of image:
  - Linux Terminal Server, this type of server is used as windows desktop replacement.
  - Windows Terminal Server, is as workstation replacement.
  - Network service, based on OpenBSD with DNS, DHCP, Proxy, Centralized backup and other network service.

Branch Office

For the branch office with more of 20 users we have installed 2 small servers. The two VMmachines have Terminal image, AFS fileserver and ldap slave all that for increase performance data access.

Cell name and File space

The cell name was taken from the domain name of the company, same decision was taken for Kerberos realm name. For the file space structure the decision was to reflect the geographic organization of the company (similar decision was for LDAP DIT).

The first level was made with country name, the second level with the city of the branch office, the HQ has the same structure of a branch office

Volume schema

The "branch volume" is dedicated to branch office and include 6 volume types

Volume Type	Naming prefix
user home directory	user
Common	common
software distribution	software
Groups home	groups
Apps	apps

VMware image storage (inventory)	image

Naming example

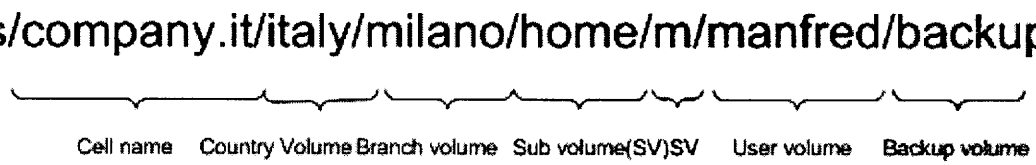
Directory usage	Volume name
User home	user.username
User home backup	user.username.backup
Application	apps.applicationname
OS Software	software.soname
Groups	groups.groupname
VMware image	image.osname

Replication schema

Volume type	Access type	Replication Type
User home	RW	Only backup replica
Groups home	RW	Only backup replica
Root.afs	RO	Replicated on all servers
Root.afs	RO	Replicated on all servers
Software	RW	Replicated on all servers
Apps	RW	Replicated on all servers
VMware image	RW	Replicated on single server

Home directory mount point

For home user directory we have decide to distribute the homes into alphabetic subdirectories, and mount each users backup as a backup directory :



Partition Filesystem

After some test we decided to use a inode OpenAFS partition backend.

Security

To simplify identity management all users profile are stored in Openldap. The Directory server is used from Linux with pam_ldap to get user account information, from samba as backend for windows authentication and from Heimdal for Kerberos authentication. OpenAFS authentication (kaserver) was disabled and all operation are made by heimdal Kerberos. Advantage of heimdal usage are ldap backend, afs integration and incremental slave propagation.

Cell auditing

One of the most important task is monitor systems and application status. We have implemented a monitor for :

- All Ubik services
- All encryption keys
- All ser ver process
- Volumes (missing,offline,orphan,size,quota)

Backup
The backup is made with a centralized system based on Amanda software. The Amanda backup can handle all the features of AFS volume. Amanda works on the backup volumes, these volumes are synchronized every day before the start of backup.

Hardware
- Fileserver /DbServer: 1GB of RAM, single processor. The disk subsystem is based on 2x36Gb SCSI RAID 1 for operating system partition and 4x 143GB SCSI RAID5.
- Authentication server: 1GB of RAM, single processor The disk subsystem is base on 2x36Gb SCSI RAID 1.
- VMmachine: 4GB of RAM dual processor. The disk subsystem is based on 2x36Gb SCSI RAID 1 for operating system and local vmware image.

Consideration
The result can be: a collection of small number of inexpensive fileservers provides equivalent performance of "big iron" machine. This approach offers an inexpensive incremental increase in capacity, better manageability and redundancy. One set of tests using NFS file sharing found that switching to AFS resulted in several performance improvements. For the same NFS type of workload, AFS resulted in a 60% decrease in network traffic. The server's load was decreased by 80%, and task execution time was reduced by 30%.

Benefits: Improved manageability, full disaster recovery protection, reduced software licensing costs for 150.000 Euro. Result of new distributed system is data accessible from Spain to Singapore with a Increase performance (Server and Desktop), high security level, Single sign-on and reduced down time.

GPL ALTERNATIVES

Distributed file systems

- Network File System (NFS) originally from Sun Microsystems. NFS may use Kerberos authentication and a client cache (NFS Version 4).
- Server message block (SMB) originally from IBM (but the most common version is modified heavily by Microsoft) is the standard in Windows-based networks. SMB is also known as Common Internet File System (CIFS) or Samba file system.

Distributed fault tolerant file systems

Distributed fault tolerant replication of data between nodes (between servers or servers/clients) for high availability and offline (disconnected) operation.

- Coda from Carnegie Mellon University focuses on bandwidth-adaptive operation (including disconnected operation) using a client-side cache for mobile computing. It is a descendant of AFS-2.

Distributed parallel file systems

Distributed parallel file systems stripe data over multiple servers for high performance. They are normally used in high-performance computing (HPC).Some of the parallel file systems may use object-based storage device (OSD) (In Lustre called OST) for chunks of data together with centralized metadata servers.

- Lustre from Cluster File Systems. (Lustre has failover, but multi-server RAID1 or RAID5 is still in their roadmap for future versions.).

Distributed parallel fault tolerant file systems

Distributed file systems, which also are parallel and fault tolerant, stripe and replicate data over multiple servers for high performance and to maintain data integrity. Even if a server fails no data is lost. The file systems are used in both high-performance computing (HPC) and high-availability clusters.

- Gfarm file system uses OpenLDAP or PostgreSQL for metadata and FUSE or LUFS for mounting. Available for Linux, FreeBSD, NetBSD and Solaris under X11 License.

In development

There are many research on the university on this theme, today the focus are on cache coordination and peer-to-peer transfer see, Shark, Coral and many others interesting project.

Using Ants to Secure your Network.

Those of us who managed diverse networks of systems, especially those that sadly include Windows(tm) systems, are constantly fighting different network intrusions. The catch is that we currently can easily fight only the ones we know about. What about the ones we do not know about?

One way to look for unknown and undiscovered network attacks is to filter the output of tcpdump, removing known packets, and leaving one with packets that might be of interest. Sadly on most networks you are left with masses of data to scan and grep/perl/awk/human eye just does not work.

This paper will present the preliminary results of a program which scans the pcap file format written by tcpdump and separates out unusual packets which can then be examined more closely by a human. It uses techniques from Ant Colony Optimization so that it is not necessary to predetermine which packets might be unusual but rather unusual packets are separated out as the result of the program running. In addition this technique works even for anonymous packets.

Background:

Internet security just is, it can not be improved. -- way long time ago, pre-google so I can not attribute it.

We currently approach computer security a few ways:

- We firewall or similar. The goal here is that outsiders are bad, insiders are good. In the case of firewalls we restrict assorted bits of our connections to and from the outside. This worked somewhat well until the web. Now everything gets wrapped up over port 80 and most of us do not have the heart (or job security) to block port 80. Combine that with the the assorted bugs and design features of browsers and you get a disaster. Note that your status of an insider or outsider is known by your physical position relative to the firewall. If per chance your physical position changes then your status changes as well. This causes problems with wireless access points and/or complex network designs.

- Virus/spyware/whatever checkers. These accept the fact that some less well designed systems can be compromised. Either a program is known to be bad, or it is ok. Note the binary distinction again.

- The Unix way. root is all powerful, and you work very hard to keep users away from root and each other. OpenBSD works to be the peak of this but most of the rest of the world does a pretty good job.

A few features stand out in the three above cases.

- It is very binary. Therefore it is easy to tell who is good and who is bad.

- You must have prior knowledge about who is good or bad, ie, port 1434/UDP is probably bad to keep open, no password on root is considered poor form, and opening email on a windows system claiming to be from your good friend with pictures of some star without clothes is likely to be a bad idea, especially if your email reader likes to help you out by running programs.

- As a result of the prior knowledge problem above there is a (possibly long) delay from the production of a security exploit and your fixing of the exploit.

While this has worked moderately well in the past this will work less well in the future for two reasons:

- The reasons for breaking into systems have changed. It used to be for bragging rights, interesting files, cpu time or disk space. These last two are very cheap these days. Now it is increasingly for money or information to be sold for money.

- Many of the "break-ins" look less like break-ins these days. The user willingly ran the program, they just did not know about all of the program's interesting features.

This last point deserves more than just casual dismissal, ie, *I* am smart enought not to run those programs that catch the less experienced users, right?. Maybe not. Most people run some sort of web browser. Surely you all have read and understood ALL of Firefox, right? Now go read about the Underhanded C Contest. Are you SURE that Firefox is not sending out private data on you? You can partially solve this on reasonable systems by having different users for different classes of web access, i.e. one user for banking, one for amazon/ebay/others, and one for general browsing. Still, how many of us do this?

Overview:

This paper will take a step back. We will not care what actually runs on the individual computers. That is not to say that is not a problem, just that is not the problem we are going to look at. What we are interested in here is the communication to the outside. Why? I care greatly who gets my banking details and so on. I care greatly that Sony installed a root kit on the kids computer.

I care much less that someone's windows system is slow. I, of course, do care about my main SPARCstation 20 slowing down, but, that should be pretty easy to debug. It is slow after all.

Our Assumptions:

- *YOU* do not know everything running on your computer. Works well for 99% of the world. The remaining 1% is probably mistaken unless they run a very limited set of programs. This limited set can not include a web browser.

- You may or may not have "less secure" systems on your network.

- You allow at least port 80 access out of your network.

- You are willing to do some manual digging.

- You can set up tcpdump to grab samples of your network.

Thinking, "how hard can this be" you set one of your switches to forward all of your outbound traffic to some computer, start tcp dump on that interface, and wait.

Now, what are you going to do with the Gigabytes of pcap data you just captured. Well, you could run tcpdump on them and parse them through various pattern matches, likely in Perl, until you get a subset of packets that are suspicious and/or know to be bad. This is a good start but:

- You have to get a rule set.

- You have to trust the rule set.

- You have to keep it up to date.

- And on and on...

And you have basically the problem described above. Before you can catch suspicious packets you first have to know that they are bad.

A Partial Solution:

What I am proposing is a softer approach. You basically run a sort of the different packets and then throw away the big batches which are similar. You then can look at the odd ball ones. What this does is reduce the number of packets you have to go through. The sort is a special type of sort that, unlike deterministic sorts, is similar to how ants sort larvae.

It is also possible to reverse the algorithm if you think you have some massive attack, but, this is normally less interesting because when you have a lot of packets the bad ones are easier to find with simpler and less cpu intensive techniques.

What's good about this?

- No pre knowledge. You will not have to filter out all those port 80 packets to cnn.com, foxnews.com, and google.com since they will be grouped together and ignored. This does not sound like such the advantage until you remember that accessing one of those pages hits a half dozen other sites, some with odd names, just for usage tracking and ads. Which sites they redirect you to for ads might change hour by hour. You likely do not care and want to skip those sites for now.

- The technique can work on packets that have different parts hashed. As long as all the packet parts hash consistently then you can find the original packets. The technique does not depend on the fact that you know or do not know that this packet was sent to cnn.com. In addition, the current implementation does not look at the data portion of the UDP or TCP packet. This means that it could be zeroed or thrown away thus increasing privacy. Note that it's possible that future versions will need this data. This is still being researched.

I will note that "just the headers" are considered fair game these days in the US and many businesses in less enlightened areas consider these to be business data and their property.

- It can soak up all that spare CPU you have sitting around.

What's not so good?

- If your CEO visits spankme.com at 10:00am every morning those packets will probably stand out. You do not want the output of this technique going to an automated system, but, rather, to a person who can make judgements. Do remember that you are looking at data that most of your users consider a bit private, and, may even be protected by privacy laws.

- It will soak up what every cpu time you have to spare.

- It points out suspicious packets but it is not exact. Think of this as being the eyes of a policeman. Good police walk or drive around a town and have a "gut" feeling about situations. They then investigate. Most often it is nothing, but, sometimes it is important. PckSwarms helps to point out places to investigate.

- Sensitive to probability settings. Get the settings wrong and the algorithm does not converge or, converges too quickly to a local minimum so that you do not get good results.

- This is a NP-Complete problem. Ergo we do not solve it, we use a heuristic. You might not get a good answer. You can think of each packet as being a node on a graph, with the edges connecting to similar nodes via weights. This is a very highly connected graph. The NP Complete problem is similar to finding a clique, with the added glitch that we are trying to minimize the edge weight in our clique. This is similar to a graph partitioning problem as well.

Description:

The basic algorithm is quite simple. You should basically imagine a bunch of ant larvae sitting in a nest, and, a bunch of ants walking around, possibly picking up a larva, and, sometime later, possibly dropping the larva. The key to the algorithm is that the ants each make some sort of judgment about how self similar the larva they are carrying and the larvae where they are standing. Note that it is similarity, not exactness that is important here.

In this implementation each individual pcap record is a larva. These are distributed around a field and then ants are created. Each ant has a direction and a speed. The ants are then moved about the field, and at each step possibly picking up a larva or dropping a larva.

As a result, over time, pcap records which are similar start collecting close to each other in the field. After a number of steps we pass over the field finding the peaks, and remove the larvae which are at or close to those peaks.

The remaining packets are "interesting." Note that if you used anonymous packets the only real data you have to go on is the time stamps from the pcap file. You would have to give a list of timestamps to the custodian of the pcap data and they could retrieve the records.

It's quite possible that post filtering might help to make this a more automated process. If you know that your CEO likes spankme.com then you might as well filter those out so that you do not have to spend time chasing those leads down.

Current Results:

The program runs and does find odd packets, but, it does not converge as fast as desired. Work continues on tuning the parameters to get better convergence and better results. Come and listen to the EuroBSDCon 2006 presentation for more information or check the current status at http://edoneel.chaosnet.org/PckSwarms.html.

Implementation:

Since this is a spare time project I used what I was comfortable with, ie, Common Lisp. I started with clisp and reading the pcap data files directly rather than using libpcap. I switched to ECL (Embedded Common Lisp http://ecls.sourceforge.net) so that it would work on my "fast" systems, the 300 and 400mhz Sun Ultras. Even on an older system (200mhz SPARCstation 20) it runs in a reasonable time for one run.

Reference:

Please see Swarm Intelligence: From Natural to Artificial Systems (Santa Fe Institute Studies on the Sciences of Complexity) by Bonabeau, Dorigo, Theraulaz, Section 4.2, Cemetery Organization and Larva Sorting.

Bruce O'Neel has worked many years in different bits of the computer world. His non-paid work these days concentrates on OpenBSD.

Third-party software management under BSD

Andrew Pantyukhin <infofarmer@FreeBSD.org>
EuroBSDCon 2006

Preface

When I set out to write this paper in July 2006 I was a FreeBSD ports committer, determined to find something new in OpenBSD ports, NetBSD pkgsrc, as well as in a number of software management systems for Linux. I was hoping to find a way for the BSD community to exchange ideas with each other and to learn a lot from our Linux colleagues. Now, three months later, I'm still a FreeBSD ports committer, and I'm still hoping for us to work together, but I sure have gone a long way, longer than I ever expected to. The thing is, software management is developing so rapidly, you can never expect anything from it until you go and see for yourself what's happening.

In July I was pretty sure what I am going to write about, but a few weeks after I started the research, I was abashed by the affluence of information and I knew it was impossible just to describe solutions and discuss their implementation. In this paper, in addition to some factual background, in a clumsy, but purposefully informal and easy-going way, I try no more but to convey my own impressions from my venture into the world of package management.

Introduction

Operating systems come bundled with software. As removable media grows in size, leaving developers, trying to fill it up with code, far behind, we can fit more and more on a CD, DVD, Blu-ray Disc and what not. But while it seems to many end-users that somewhere there's a perfect combination of tools to cater to all their needs, they fail to see some simple points, exposing this illusion:

- However huge data storage is and however fast it grows, the number of software projects is overwhelming. With over 130 thousand projects at SourceForge alone, and many similar repositories amassing dozens of thousands more, it is absolutely clear why we just cannot jam everything into one distribution and present it to somebody other than a football-field-sized data center owner.
- We can greet a user with gigabytes of the most popular software in the world, and many Linux distributions do just that. But in our naturally heterogenous IT world, there's always a great deal of unsolved problems. And once some piece of software answers a need, users want it. They won't wait until the next version of the whole distro, they won't wait until the packagers actually notice the new tool, the want it here and now.
- We can't pretend every user has enough resources to install a multigigabyte chunk of software just like that. There's embedded market where you need to enjoy your life on a shoestring, there are users with legacy hardware, there are users multibooting in 10 different systems, there are virtual private servers - and in each case any piece of software can be required, but it's not possible to install all the software at once.

Hence packages. Traditionally, package management can be integrated into packages themselves, or into the operating environment. The first way is decentralized by design, and popular among commercial closed-source software vendors. They don't like to conform to cheaply advertised standards or to wait for anyone to accept their package into a repository, so they just bundle their programs with installers, and sometimes deinstallers, and make it available as an executable. That's the way most packages come on proprietary desktop operating systems and many proprietary packages on other systems, and unfortunately that's the way to give your system administrator nightmares. The other way usually involves some guidelines, which package developers, or packagers, have to take into account in order to build a conformant package. Such packages are usually easy to install, deinstall and upgrade through a common interface.

History

Package management in UNIX

Before package management existed, as we know it now, developers preferred spending their time troubleshooting installation issues to thinking about deinstallation. This approach became deeply rooted, and remains so to this day, in the Windows world. Back then a user usually had to get a file archive, extract it, optionally hack it and compile it, and install it. Surprisingly, today some administrators, especially those dealing with more obscure proprietary systems, regard this routine as quite straight and normal. Additionally many operating systems came bundled with all the software you were supposed to ever need.

That's the way BSD systems went, coming with rich userlands so that users might have a chance to never think about anything third-party. That's the way early Linux distributions were - it was all OS developers' job to decide what's important, compile it, integrate it and give you a nice versatile bundle.

But of course it couldn't stay that way for long. Eventually Unix got its System V (Solaris) PKG format and users started using binary packages, which they didn't have to hack or compile, or even extract. A simple pkgadd command would "transfer" the package to their system, and pkgrm would remove it. Pkginfo and half a dozen extra tools were also there to constitute one of the first Package Management Systems (PMS).

Package management in BSD

In August 1993 Jordan Hubbard committed his package install suite, and almost exactly a year later he presented us with his new ports make macros, also known as bsd.port.mk. NetBSD imported the pkg tools in summer 1997 and later that year they adopted the ports technology under codename pkgsrc (because the word "port" already meant a hardware architecture in NetBSD). OpenBSD inherited pkg_install suite and ports from NetBSD; pkg tools were rewritten in Perl by Marc Espie in 2003, but this new version has remained limited to OpenBSD to this day.

Initially the FreeBSD ports system was just a facility to ease building binary ports, a collection of macros written in make, which later became a vital part of all three major BSD OS's.

Package management in Linux

Year 1993 welcomed Slackware, Debian, RedHat and Bogus distributions to the scene - and each came with its own PMS. By mid 1994 there were Slackware packages, a modestly-named PMS system in Bogus, RedHat RPP and Debian dpkg solutions. RedHat later developed a new system called RPM, which are, together with Debian packages the two most popular PMS for Linux today.

When Gentoo Linux 1.0 was released in 2002, it came with a system called Portage. Based on FreeBSD ports, it was powered by bash and python instead of make and shell. With over 11000 of official separate packages, it is one of the most comprehensive centralized repositories of third-party software for Linux.

Today

Today there are dozens of systems, allowing to manage software on Unix-like systems in one way or another. They can be divided into binary-based, where you only deal with binary packages, source-based, where you install everything by compiling from source and hybrid, where you can do a little bit of both. In fact, all systems deal with source code at some point and all systems deal with binaries when the software is installed, so the real difference is how they manage to compile software, install it, remove it, and perform other management tasks.

Early PMS did not provide much help in compiling the code. More often than not, you were required to compile it by hand, move binaries to a special place - and use some tool to archive it along with some metadata into a package. That wasn't a very pleasant job, especially if you consider the wealth of open source software and the frequency of updates. Ports makefiles, RPM recipes, Debian control files, Portage ebuilds - are all there now to ease the task by automation and modularization of common actions.

You can hardly imagine building thousands of packages by hand, if you take into account that you have to do that for several versions of an OS, multiplied by several hardware architectures. Today in the FreeBSD ports system less than a megabyte of uncompressed core make macros make it possible for the other 375 megabytes of package-specific code perform this task with excellence, compiling over 15000 pieces of software, which amount to tens of gigabytes of non-bloatware source code, into packages.

BSD ports and especially Portage have very advanced macro systems, while RPM and dpkg mostly utilize separate packages to perform common actions. All these systems deal with pristine sources, i.e. they store all the information needed for the original upstream source code to be compiled into a package. Lots and lots of portability issues have resulted in many macros, which effectively unburdened thousands of software developers, and let us compile code written without much portability in mind with no showstopping trouble.

Of course, compilation is only a part of the whole picture. We can't just throw binaries at users, we have to make installed software easily available. For plain old console apps it just means placing binaries in a PATH-exposed directory. For daemons, we have to help user stop and start them at reboot automatically. For X11 apps we may have to install some Gnome- or KDE-specific files. And things just get more complicated when it comes to web- and SQL-based software, and other modern software usage paradigms, like virtualization, clustering, and so on. Some of these issues are solved in many PMS, others are not even planned to be alleviated or even not recognized as problems, but believed to be there to entertain system administrators.

We'll now look at some popular contemporary PMS, at issues and solutions, at what users and porters expect from the infrastructure, and we will try to understand why there is so little collaboration between very similar projects and how people can start working together.

FreeBSD ports

Most of us know how FreeBSD ports system works in general. It's written in make and shell with every port having its own Makefile along with some other files, like patches and checksums, but the way we see it as multiple files in multiple multilevel directories is only a matter of organization. In fact, we could have had everything fit into a handful of makefiles and specify what port we're going to operate on every time we invoke make. There are countless ways to organize these hundreds of

negabytes of code. With shell and make being comparatively simple languages, we've seen snippets ported from one to the other and back.

Core ports macros are located in a special Mk directory. They can be used by ports directly, or through the main macro package, bsd.port.mk, also located there, by setting special USE_XXX flags. A number of additional macros is located throughout the ports tree. In theory, you can create a port without using a single macro package, but macros ease the task immensely. You would have to program all the actions manually, from fetching, building and installing the software to creating a standard package. While most actions can be redefined, no port ever required to redefine all of them (there are over hundred actions defined in just bsd.port.mk).

Thanks to macros most of the work is already done for you. In many simple cases, all a porter has to do is to write down a name, version, and download URLs for a piece of software, along with a short description and a list of files it installs - and a port is all ready. You can install it, remove it, make a package and submit it for inclusion into the official ports tree. But you only begin to experience the power of the ports system when you have some trouble with an app. You can solve most problems with a couple of tweaks, but there are hard nuts, when you spend hours trying to figure out what to patch and why does it segfault at start. There's always room for automation, though. Many porters find themselves doing the same hacks over and over again, - and only reluctant to automate it all because it's not that easy. Portage has a well thought-out eclass system to encourage streamlining all kinds of hacks, we'll look at it later.

NetBSD pkgsrc

Many users think that OpenBSD and FreeBSD ports are very similar, because they are both "ports", and NetBSD pkgsrc is something alike, but still different, because it sounds very different. In fact, like we said a bit earlier, pkgsrc would probably have been called ports if only the word "port" had not already had an entirely different meaning in the NetBSD project. It's a challenge to find out whether it was OpenBSD or NetBSD who has done more work on ports, but at first sight pkgsrc feels more like FreeBSD ports than OpenBSD ports do. It is probably because OpenBSD guys had rewritten the pkg_install suite from scratch (and renamed it to pkg_add to avoid a directory name clash during the transition). Along the way, they introduced many improvements into the infrastructure, as we'll see in a minute.

Now NetBSD still uses the original pkg_install suite, although John Kohl has contributed to the code and many of the refinements made it back to FreeBSD. Pkgsrc also got many interesting features, to name a few random ones:

- licensing notion - ports refer to license names, which are located in a separate common directory. A user can restrict available ports to a subset of known licensing options
- print-PLIST target - simple, but nice automatic plist generation tool, it uses "find -newer" and some awk/mtree magic
- good documentation - pkgsrc.txt is a comprehensive guide for users, porters and developers
- buildlink3 - symlinks required libs and headers into WRKDIR at pre-build
- builtin.mk - decide if system or installed lib should be used
- pkginstall framework - some common tasks for install/deinstall scripts have been automated, like user/group creation and dealing with config file
- pkg options framework - options have been reworked to allow for easy customization
- more flexible subst framework
- policy-prodded unique dist_subdirs for rerolled distfiles

OpenBSD ports

Like I just mentioned, OpenBSD ports infrastructure seems to have changed a lot since it was inherited from FreeBSD. The fact is it might have experienced much less development than pkgsrc has, but the changes affected it in a more visible way. And that's what any infrastructure should probably be aiming at - little changes in the core producing much positive effect in the consumers.

- fake build environment - when you install a port, it is first installed into a wrkdir called fake root, then package is built and only then is it installed
- immaculate documentation - many comments made it from makefiles into manpages, many concepts are now described in dedicated manpages
- options reworked into flavors, a little less flexible, but a lot cleaner mechanism
- multi-packages - building several packages from a port the smart way
- packages with different options or different subpackages in a multi-package have different filenames
- you can act on several ports in a go, grouped by package name, category or maintainer
- locking-supported parallel builds
- built-in updating support

The whole pkg_install suite has been rewritten in Perl, and became arguably a lot smarter. I won't discuss it right now, but the ultimate target of OpenBSD ports developers is to integrate most package management tasks into the base system.

Other Worlds

Many of us remember that there's much more to operating systems than BSD, some even recognize the word Linux when they hear it. Apart from BSD ports there are three big package-management players in the Unix-like world: RPM (RedHat, SuSE), dpkg (Debian, Ubuntu) and a rising star named Portage (Gentoo). And there are dozens of other most diversified approaches, which I won't discuss in detail, but will mention when I talk about some interesting features.

RPM is probably the best-known package format in the world. It is associated with a package manager of the same name. RPM manager can run on most Unix-like systems and has been employed as a built-in feature in many Linux distributions. Binary RPM packages are built from source ones, which usually contain pristine sources, patches and a spec file, much like a BSD port's makefile. There is no central repository of macros, so packagers are restricted to RPM built-in functionality. Binary packages from one system are usually unusable on another, or even on a different version of the same. Unfortunately, source packages usually obey the same rule, which limits RPM in its success as a universal package manager. When vendors publish packages, they usually have to provide one for each OS it is supposed to run on. There are efforts under way, most prominently Linux Core Consortium, which is behind Linux Standard Base, to alleviate the problem of incompatible packages.

Dpkg approach, also known for it's .deb packages, is a lot like RPM, but thanks to rigorous packaging practices has much fewer compatibility issues. Binary packages from one Debian-based system usually run on another one. Lately there have been some issues with Ubuntu, the most popular Debian derivative, about package compatibility. We can only hope that Ian Murdock, Debian founder and ex-leader, will do everything he can to prevent RPM chaos from coming into Debian family (he's also working on LSB), while we discuss some other dpkg highlights. Documentation is extensive and quite impressive, leaving no room for questions from a novice, but the thing packagers profit the most from is probably debhelper suite, and lately Common Debian Build System (CDBS). Debhelper is a collection of tools which can be called from rules files - makefiles controlling how package is built. CDBS is a collection of macros packages, much like dot.mk files in BSD infrastructures. They can be included into rules files to use predefined targets and other handy make macros. CDBS builds on debhelper, and together they can bring packagers even more convenience then ports currently do.

Last but not least is the youngest, most vigorous system named Portage, as a tribute to BSD ports. Its original developer, Daniel Robbins, took a foray into FreeBSD and later used his impressions to design a new PMS in Bash and Python, which is now the heart of Gentoo Linux. He did a great job at studying what other systems did, so he laid out a pretty slick design and implemented it successfully. Somewhat like RPM, Portage uses Bash scripts, named ebuilds, to control the building process. To provide debhelper and CDBS functionality, he designed a system of eclasses, also Bash scripts, which are a lot more fun to use than make macro packages. All in all, Portage does not introduce any revolutionary practices in PMS world, except for bringing it home that source-based PMS can be a success on Linux, but its straightforward design and the power of Bash at its core attracted many developers and made it grow as fast as no one could expect.

Why Bother?

So there are BSD ports, Linux packages and a lot of other systems. Maybe we could take a look and learn something new, but at any rate, we should probably try to save our individuality and leave other projects well enough alone; diversity is good, right? Well, the problem is that no package management system of today can cope with users' demands. Whatever OS you use, you'll always meet some mishaps and shortcomings. First of all, there is enormous amount of open source projects. Whenever we tell a user "you'll find everything you need right here in our collection" we are lying. He'll be lucky to find a few most popular percent of currently available software, and he'll be very lucky to find that most of them are up to date and usable. And by only exposing the most popular programs, we are actually raising barriers for them to become popular in the first place. And by saying "you don't need that and that anyway" we begin to dictate our opinion.

And the problems are not just in the numbers. It's a topic for another pile of papers, whether it's right or wrong to present users with a zillion of useless tools, whether diversity on its own is vile or virtuous. But there is so much more to both qualitative and quantitative metrics describing the way PMS serves its purpose. In a minute we'll start looking at some issues and solutions, and will hopefully discover that no project alone can embrace even a list of problems it would want to solve. Sometimes users are so loyal they mistake shortcomings of the systems they use for the way things should be, or even consider them advantageous. For instance, those who use binary-centric systems exclusively often frown upon source-based ones, because they are unaware of the problems which they can solve. And the other way around.

The interesting thing about packaging is that we all use the same software. At the operating system level, all we might care about is interoperability standards; implementation can work in ten different ways under the hood. But when it comes to third-party software - we're actually using the very same source code on all the different platforms. So while developers might pride on their distinctiveness and isolationism, packagers just can't do that. Be it FreeBSD, Linux or Mac OS, we should look for ways to work together, or we'll end up thinking that we're doing great when in fact we're suffocating both our users and software vendors. And the current situation of three BSDs working on three separate ports systems is just inconceivable. We're so close together we could fall on each other - and yet we find it much more comfortable to tweak things on our own.

We shall consider how to meet each other halfway later on, and now let's take look at what's bothering us, and what PMS projects are having fun about.

What's up?

Scalability - Package Building

One of the main problems in any PMS is package building. Most porters acknowledge this, and the FreeBSD portmgr team could probably write an epic about it. Basically, FreeBSD package building cluster is a bunch of donated boxes. When building the whole tree takes desperate amounts of time, we ask for more hardware resources - and sometimes we even get them. As a result we've got one of the most up-to-date PMS trees out there and one of the most outdated package collections. Most Linux distributions don't seem to have these problems, but in reality they are just cheating. Fedora builds only the core, official packages, plus a generous amount of extras, - and lets users go find all the software they need anywhere else. Portage only builds at release times. Debian allows porters to build and upload packages themselves. BSD ports might have something to learn from each of them.

Firstly, traditionally we always try to build the whole tree, but we really don't have to. When it comes to a point when we just can't handle more package building, we either don't accept new ports or don't build them. Whichever is lesser evil is a hard question, but while we can handle a lot more code in our VCS there's no reason not to allow it to be added.

Secondly, also by tradition, we keep package building centralized. Centralization is always a two-edged sword. It keeps us from wasting coding efforts on redundant solutions, i.e. encourages collaboration, but it also demands non-trivial hardware resources when it comes to hungry tasks. At this point we can't just let porters build packages themselves and upload them, because it's a commonplace to customize build environments, but it's possible to automate standards-compliant builds, and in a way less painful than tinderbox to set up. Once porters can build standard packages, it can be automated. Everyone takes the ports he maintains and builds them on whatever archs he can, pkg_adds them, tries to run, uploads. Once the building part is automated, we can distribute tasks among both porters and non-porters. And distribution of hardware-hungry tasks seems to always solve the problem. Of course, there are security ramifications to be thought about, but in general, we have to trust people. Package signatures will be a must, though.

System Resources: Using, Keeping Track

In a way, every PMS solves a problem of managing system resources, like disk space, file namespaces, user names, etc. It's just that few people put it this way. When we think about a program which requires a specific user name, we imagine a script to create it at install, remove it at deinstall, spit out some warnings if the user already exists in our system and so on. Why don't we call it a resource and acknowledge that the app needs it. We might have one and we might not; some resources, like user names, can be shared between a number of different programs; some, like a TCP port at a specific IP address, usually can't. Whatever we should call a resource depends on our imagination.

To reiterate, among the things that can be actually spent or saved or wasted, programs usually require:

- disk space - this is ignored much too often, but it's far too important. A PMS of the future should probably provide a means of package-specific runtime disk space quotas, which are requested at pre-install time and prevent programs from filling up /var with logs and other similar issues. A user should also be able to view requirements of the packages he has installed, is installing or is planning to install, so that he can decide on his hard disk layout, or what to share via NFS, or numerous other points of administrative design.
- directory/file namespaces - facing a problem of having multiple instances of the same packages (of one or several different versions) installed at the same time, we should think about naming problems.
- user/group names/ids - many programs require a dedicated username (and for security reasons we might want to encourage it where it is optional), some share it with other programs (e.g. many webapps share user/group with webserver programs), but there's always a problem when it comes to adding/removing user accounts. There are ways to run a program under whatever user we like, so we should avoid hardcoding user id's or specific username.
- TCP/UDP ports - we are accustomed to seeing ports as some hardcoded property of a program. In fact, almost any network-enable program provides a way to specify what port it should use. And we should leverage it in order to automate installation of several similar apps on one box.
- CPU, RAM, disk throughput, number of processes, number of open files, etc. - of course it would be cool if we could distribute performance based on priorities, soft/hard limits or otherwise between all the packages we have installed. Unfortunately, few operating systems have enough built-in functionality to implement that. Of course, we could employ some clever wrapper scripts or other hacks, but an efficient solution would still require OS support.

There's more to Resources

Now that we've seen how packages consume resources, why don't we allow packages to provide resources? Databases, virtual hosts, pixel-based on-screen real-estate, client connections to some persistent antivirus engine - you name it. Is it possible to automate it in a safe way? - Why not? And who could possibly do it in a better way than the maintainers themselves, who usually know more about their particular piece of software than most other users do. Of course, there are security issues to consider, but in fact many administrators choose less secure configurations in favor of more complicated ones - just because they haven't got enough time or zest. Apache runs chrooted on OpenBSD by default, it's not a port, but that's an accomplishment all the same. I doubt half of FreeBSD users chroot Apache by hand, in spite of all the security benefits. And what does it take porters to automate this setup? Probably less than it would take a new Apache user to do it the first time.

Of course, flexibility issues arise when porters try to make mandatory decisions for users. Well, it usually only takes one "if" clause to make some action optional. Moreover, porters should try to allow for many common choices. Let users prefer Postgres to MySQL, or database on another host to local one, and let applications take that preferences into account.

Customization

Resource management is tightly coupled with a more general problem of software customization, from setting preferences to applying useful functionality-enhancing patches. I must have installed phpMyAdmin for a hundred times and almost each time I had to edit the configuration file to make the very same change - enable cookie-based authentication and set a blowfish secret. It would probably take less than an hour to implement some "with_" variable and automate the whole process. Many other webapps offer generous web-based installation wizards, but they always ask almost the very same questions. What would it take to let user say "I've got this database on this host with these admin credentials, please manage db/user creation for me"?

Sometimes programs require particular settings tweaked in other programs. A well-known example is php.ini settings. Should we make user deal with it herself or should we outline requirements and automate all the necessary tweaks if some super-manual-override mode has not been enabled in make.conf?

User interface

Most PMS have a unified interface to perform all the tasks related to software management. Here the simplicity of management contradicts flexibility and complexity of operation. I've always liked the way VCS clients deal with the problem. One main program, comprehensive easy-to-use help system, orthogonal switches, dozens of completely different functions performed by intuitive concise incantations. OpenBSD has taken pkg_install suite there (by rewriting it in Perl from scratch), Portage has emerge, Debian - apt. For a long time now FreeBSD has relied on portupgrade. Doug Barton has been working on a new tool called portmaster, written in shell, so that it can be integrated into the base system. But we have still to see a tool to let us customize ports. The way users are asked to set options now is strange to say the least. There is a tool named portconf, but it's more of a hack than a solution.

Choosing what (not) to install

Most users crave an easy way to say, what he wants to be installed, what he considers OK to be installed and what he doesn't want to be installed at all. At any given time, the PMS should know which of the installed packages are actually required by the user and which are installed as requirements for other packages. Sometimes it's important to be able to mark packages not to be installed under any circumstances. For example, a user might not want X.Org libraries on a server with constrained resources - or just to keep system clean for that matter - and he would prefer some graphic app failing to install instead of having a bunch of heavy-weight packages installed.

Where do the old versions go?

FreeBSD ports pride upon being one of the most current repositories of open-source software in the industry, without having too much of stability hassle. This makes it possible for all kinds of users to stay on the edge. But a lot of users require much more than just that. There are countless situations when an earlier version of a program is required. Most PMS, including BSD ports, try to solve this problem by providing several major versions as separate branches of a package. But what if a user requires an earlier version on a branch? Currently the only two solutions are to hunt for old packages or to downgrade the infrastructure itself. Both are good ways to mess up your system.

Portage has a lazy, but a better way to deal with it. They keep several versions of ebuilds (counterparts of our Makefiles) in directories of many ports. It's not very VCS-friendly, and they have to maintain each of the ebuilds, but it works.

Multiple problems arise when we talk about multiversioned ports and packages. To introduce full support into packages, we would have to redesign the whole concept of package dependencies. For the time being we might be better off leaving packages well enough alone, depending on a single version of each required package. The versions might be explicitly specified, designated as default in the dependency itself, or just the latest ones.

Metadata storage, as well as distfile storage are of particular interest. With metadata (makefiles, distfiles, patches and so on) we might go the Portage way and keep different versions in separate files. A more efficient solution might be to keep them on different branches in our VCS. As for distfiles - we may choose to drop support of unavailable versions, or, much better, mirror older distfiles. Of course, just to mirror them would put a substantial strain on disk space resources of our mirrors. Therefore, we should consider a possibility of keeping distfiles on vendor branches, also in out VCS. By all means, this repository may be separate from the one where we keep the OS and ports sources, but in fact it won't create much pollution in a change-set based VCS. Every update is just one changeset. As for digests, we'll have to keep per-file ones in addition to per-distfile ones. A successful solution will probably require extensive checkout capabilities, so that users could get a .tar.bz2 archive via http or ftp, containing all the sources of a particular version. It's not impossible, though again places additional load on the mirrors. On the other hand, per-file digests will make it possible to choose new compression algorithms in a trouble-free way. Some hosts might choose to offer LZMA-compressed checkouts, which will help users cut down on their traffic.

Repository-based PMS is not news. RPath, Inc. presented a system named Conary back in 2004. Conary implements a new vision of package management, proudly called software provisioning. It is based on the concepts of DVCS, peering far into the future. There's even a Linux distro called Foresight based on Conary (not to mention rPath's own Linux of the same name). Unfortunately, Conary is not very active right now, but it has already generated a wealth of documentation for us to learn from.

Fetching sources

Speaking of distfiles, there are more ways today to get them than just fetch(1). People make software available in form of RCS files, anonymous VCS access, p2p shares and metalinks. We make porters deal with that by providing traditional archives via http and ftp links. Some distfiles can not be downloaded automatically because of licensing restrictions. In such cases we usually weed out the lazy users by telling them to go to such and such URL, register and download a file with such and such name. Instead we could present him with a text browser window and even a preregistered bugmenot-like account. Not that we should encourage the use of non-free software, but we don't make life much easier for users when we strongly discourage that.

Incremental distfiles

Some users still have very slow and/or expensive bandwidth. Many of them look at the rate our OpenOffice port is updated at and wish they could always have the current version, but they just can't afford downloading 300 Mb several times a month. What if users could update their distfiles incrementally? A bzip2-compressed diff between OOo 2.0.4 RC1 and RC3 is about 1 Mb, which could result in 300 times less traffic consumed for the upgrade. And we already have a solution which takes care of the ports collection itself - portsnap. It's not an easy task to marry portsnap's concepts to distfile updates, and again, we have the problem of keeping the distfiles in a versioned environment. We even have a highly efficient bsdiff binary diff solution from Colin Percival, and some room for its improvement in a doctoral thesis by the same author, just in case we decide to version-control closed-source or non-textual data.

Functionality providers

Many PMS (like Debian and Portage) implement so-called virtual packages, where several programs with similar functionality (e.g. mail clients, or web servers) are united into one, "provide" the virtual package. Several "providers" can be installed at the same time, one of them presented to the user via a uniform wrapper script, or a symlink (e.g. type "mail" and one of providers - whichever priority is highest - will be launched). Not only is this a user-friendly way to present some functionality, but also a convenient paradigm when it comes to other programs depending on some kind of facility, e.g. a webserver or a MTA.

Multiple instances of the same program

Portage has a feature called slots, where multiple versions (branches) of the same package with different slot numbers defined can coexist on one system. FreeBSD also has this feature in form of version-suffixed port and package names. A little bit earlier we were talking about how cool it would be to have access to all versions of an app at once. Indeed, this is especially true in high-availability environments where you can't afford downtime and should test every new version before deploying it. While a separate sandbox is always advisable, why not just allow to install the new version on existing system without deleting the old one? This way a roll-back will only take a few seconds. Moreover, no matter what app we're talking about in most cases you'll be able to provide users with access to both versions at the same time.

Tobias Oetiker, the man behind the ubiquitous RRDtool and MRTG, has once been challenged with package management across 400 Unix workstations. Of course, he developed his own system named SEPP and his users were happy ever since. The fact is that whenever an upgrade took place, they always could launch the old version of a program. And before each upgrade they were given a chance to try out the new version. In fact they could keep all the versions they wanted for as long as they liked. Nowadays this has many security implications, but we'll talk about security later and now let's just say there's Debian which almost always applies security patches to older versions, so it's not impossible in practice. SEPP installs programs into versioned directories. Inter-package dependencies are supported, but Tobias recommends keeping everything a program requires in one package. It's impractical in most cases, but in some cases this approach can be beneficial. There are wrapper stub scripts and symlinks making software available to users, keeping statistics and providing for some additional run-time configuration.

There are other systems that keep packages installed in separate subdirectories. Keeping them versioned solves many issues, such as file namespaces we were talking about. There is system integration to think about - manpages, rc.d scripts can all be made versioned, but require non-trivial design decisions to be made. Last but not least, if we allow several versions of a package to be installed simultaneously, why not allow same versions to be installed in the same fashion. It would not require much more - just a special instance identity to augment versioning and avoid name clashes. If we couple this functionality with resource management, running three daemons of the same version but with different patches applied will become a hassle-free operation.

As for running several instances of the very same binary - it can be also be achieved by launching it with different options. This has a benefit of run-time configuration as opposed to build-time in multiple instancing with different binaries. Unfortunately, some apps have hard coded values. They will have to be either patched to be run-time configurable or

configured at compile-time only.

Movable packages

Whether installed into private subdirectories or not, it would be great to allow user to relocate a package installation from one place to another without disrupting it. The problem is reconfiguring it at runtime so that it is not surprised by new paths.

More runtime customization

Post-installation resource management is very important in dynamic environments. You have a webserver listening on port 80, you install a new version and it listens on port 81. Once you verify that the new version works OK, you have to be able to exchange resources between the servers. Apart from port numbers, there might be different document root paths, database users and so on. Of course, we can make the user to just reinstall the daemon, but while a runtime reconfiguration should only take a few seconds, a reinstallation might require much more time.

Non-privileged package management

Non-root installation is advertised as one of the Holy Grails in the new breed of user-friendly clickety-clack systems, such as Klik and Zero Install. Any PMS could benefit from this functionality. Hacky solutions can be based on running PMS in a chrooted environment, but a real solution should introduce a notion of user installs into the core of PMS. Ideally users should be able to choose what to depend on: only system-wide packages, only user-installed ones or both. And runtime relocation and customization facilities should be able to make a user-specific package system-wide and the other way around.

Smart techniques could be employed to watch if more than one users install the same package and save disk space in some way. An even smarter, but a lot more security-encumbering solution would be to install all packages in system wide locations, but mark them available per-user and deal with per-user customizations. This would save space by default. VDS and shell providers will appreciate this kind of functionality.

Click!

One simple feature most users come to appreciate very much is the ability to do powerful clicks. I.e. you see a nice icon on a website, you click on it - and the next thing you know is a full-blown application installed and launched on your computer. The security implications will probably make some people pant, but smart design should yield some decent insulation. The matter is too many new packaging systems attract users with this kind of features. Even Microsoft gave in to the tide and announced its ClickOnce solution where there's no setup.exe, but only a mouse, a click and a working application.

Appliances

RPath, the company behind the aforementioned Conary system, advertises a concept of software appliances. Their marketing materials are quite vague, but the idea is simple. Instead of distributing your application alone, trying to make sure that it will work in many environments, you can marry it to an operating system, and distribute it as one package, guaranteed to work on most hardware configurations. RPath provides solutions for bare hardware based on their own Linux distro, virtual appliances to be used on virtual hardware and hardware appliances which is software appliance bundled with a computer. If we're talking about FreeBSD, we can extend this concept to jail appliances. You plug them in - and they just work. And you can plug a lot of them in a single system. Sounds exciting and in fact not staggeringly difficult to implement.

Distcc/ccache

Portage and pkgsrc have built-in support for both distcc and ccache, two solutions to speed up the builds. Problem number one is getting ports to respect the designated compiler. Two is looking for issues that inevitably happen due to parallel compilation. Many users also report problems with ccache, apparently results of configuration issues. Built-in support means hassle-free automation, so all configuration problems should be sorted out with all kinds of environments in mind. Problem three is distcc on heterogenous systems, i.e. setting up an old box running FreeBSD 4.x/i386 do all the building on a fast 6.x/amd64 system, or even on machines running another OS. This brings us to cross-compilation.

Cross-compilation

It has been accepted as a fact that whether distcc is involved or not, cross-building packages is not an easy task. Nevertheless, Krister Walfridsson, presented a new concept at EuroBSDCon 2004 and implemented it in NetBSD pkgsrc in 2005. His idea is to substitute the calls to some programs during the build process with calls to emulated programs. Granted, this depends on NetBSD emulation framework, but a similar solution might be feasible on FreeBSD.

Security

Vulnerability and exposure tracking is one of the most underdeveloped processes today. There are literally dozens of commercial, community and governmental security trackers, aggregators and copycats, but they are trying so hard to keep at a distance from each other that there's no reliable source of security-related information. Fortunately there is the CVE dictionary

backed by the Mitre Corporation and the U.S. Department of Homeland Security. Most of the time it provides us with useful references so that we can say "a-ha, we are talking about the same issue". But neither does it provide comprehensive information about each particular issue, nor does it cover them all.

There is still no centralized community-based security database and PMS need it bad. Until such a facility appears, we'll continue maintaining our project-specific databases, which is not a completely lost, but mostly a wasted effort. When the database comes, each project can choose how to use it. You can either put references to fixed issues directly into packages or you can maintain a database with very simple entries: a reference to the issue in the central DB and affected packages. No descriptions, no reference hunting - these are centralized. But until version numbers and package names will become standardized, which I doubt will ever happen, PMS will have to maintain thin compatibility layers.

Porter perspective

A PMS does not only serve users, it's also there for the sake of those who actually make software available to users, i.e. porters, packagers, uploaders and whatever we might call them. Ideally, everything that can be automated, should be automated. If a program requires a library, it must be easy to designate it, without research into the current state of PMS. If a program needs a dedicated username, UDP port and or a SQL DB, a porter should be spared the effort to reinvent all the automation tricks and knacks. We have already mentioned a need for resource management, that's something both users and porters will profit from. Now, how about -

Dependencies reloaded

First of all, we can go ahead and say dependencies are also a special kind of resource, always reusable and never depletable. The problem is how we define them. First, whatever type of dependency we're talking about, most often than not we can accurately guess what port to depend on by looking at what we really need. Be it a library, an executable or a package, it's not an impossible task to automatically find the port we need. RPM has applied this approach from start by depending on files instead of packages. This proved to be a big headache, and by all means should be avoided, but the concept is sane. Even if we can't come up with a bright idea about how to implement it, let's face, it would be nice if we only had to say that we need this binary and have the infrastructure look up all the necessary stuff for us.

And like they say, the Beastie is in the detail. Poor porters have typed the word ${PORTSDIR} in the dependency specs for over 21000 times. OpenBSD for one inserts it automatically if the port origin is not absolute. And the way we depend on Perl ports is a joke. Perl ports have long been our blemish and a scapegoat, much undeservedly, too. I'm sure there are lots and lots of scripts scatter all over mailing-lists, written to ease the pain of doing something as simple as porting a fully automated CPAN module in such a routine way. From time to time we even hide a module inside of the port of another one - just to watch how a porter will port it, get a pointy hat and remove it.

When a shared lib updates its major version, it's a special treat. If you're lucky, you'll only have to bump portrevisions for a few ports. A few hundred if you're not. And it only takes a single include file with a list of all the current major/minor versions and some very simple magic to spare the whole effort.

Automating hacks

Many if not most of the ports contain some kind of hacks. With only a few dozens of ports I maintain, sometimes I find myself solving the same problem all over again just because I forgot that I've already done it, or have no time to look through all my ports, and I never take time to document little hacks. And porters never do document them either. What kind of reusable code can we talk about if there's no way to know about it, or if it's unreasonably difficult to find what you need. Discussions on IRC help, but that's just a handful out of all ports committers participating, and we obviously need something more outreaching. Writing up every trick in the Porter's Handbook is very difficult, not only because some have no wish to learn DocBook, but also because it's a Handbook, not a Cookbook. In my opinion, a wiki would suit us, but then again we should consider encouraging modularization of hacks. If everyone put every hack, however inconvenient it may seem, in Mk/hacks, with proper comments, then it would be much easier to find what you need and reuse it.

Speaking of collaboration problems inside our project, we've approached a general topic of cooperation in package management, when it comes to multiple projects across multiple environments.

Collaboration

Education

PMS developers should take an active interest in other projects. That starts with learning about them. There are not numerous enough to bury you under piles of white papers, manuals and guides. Of roughly a hundred projects only a couple of dozen have decent documentation. The thing is not only that we shouldn't reinvent the wheel, but that some decisions we are going to make might have already proven to be ruinous in other systems. Also, whatever project we consider, its developers should recognize that the bulk of users are happy with other tools. Ports might be a natural monopoly in FreeBSD, but however inconceivable it may seem to some of us, our users are just a subset of the Unix-like user base. Each PMS is only known to a fraction of users, which also means that most development and advances are happening outside. While some isolation seems to

rectify our own in-house practices, which are so dear to us, it can only hurt by filtering all the most important outsider information. Recognizing the need for more sources of input is often a non-technical problem of developers' attitudes.

Spirit

This is a complicated topic. It's known for a fact that every good engineer has an itch to implement everything to his own liking, not out of vanity, but perhaps because you can easily accept imperfections only when you're the one that's responsible for them. Anyway, most PMS founders at one point in time felt that they would be better off writing something new from scratch or forking off a seemingly stagnant project, than talking to the developers of existing systems and trying to do something together. While this has unerringly resulted in most wonderful diversity, we're still at the point of having no solution to cater to even basic user demands.

So in order to move forward developers should probably accept, at least temporarily, that (1) it's not the time to start a brand-new project that is doomed to follow in others' steps and only stand out thanks to a shiny website or a few catchy taglines; (2) it's not the time to burn a bridge and fork, this will lead to either a suffocating dead-end where there are neither interested users nor developers,or to another bridge left burning by yet another generation of successors; and (3) it's not the time to keep isolated and work on your own. Learning from each other's mistakes and successes while looking through a wall of glass, but without interactivity little progress will be made.

But even as we reach for each out, we'll find much controversies in our opinions, the same disputes that lead to forks, splits and other quarrelsome counterproductive events. We have to be prepared for that and we have to find a way to deal with it. It means FreeBSD will be working with DragonFlyBSD, NetBSD with OpenBSD, BSD with Linux, Solaris, and other Unix-like systems, and so on. And instead of standing on what we can't agree on we'll have to find solutions to problems some of us still have.

We all use the same software, the very same source code. We don't need to write something new, just to discuss how we can all use what has been written already. That's why I think we'll succeed through all our differences - because we are all doing the same thing, we just need each other's help to do it properly. If our cooperation gives life to some common portable implementation - great, but tools are not nearly as important as design points.

In any case, we can't claim we are able to change how people are, so we'll just have to find people in other projects that are willing to communicate.

Communication

A decade ago a newsgroup, a mailing-list, or some kind of other centralized communication method would probably do it. Today, it's hard to put such conventional limitations on the processes inside wide-scale highly-distributed communities. We have seen task forces, working groups, standardization committees proposing brilliant guidelines which were ignored altogether, because people are just too busy. Package management does not tolerate stagnation. With hundreds of updates out each day, we're like some news clerks, running around without looking sideways. But paying attention to what's happening over the hedge does not only help us find solutions in our everyday routine, it also makes us want to respond, to take part in foreign discussions. If we accept that we're part of the same process, subscribe to each other's mailing-lists, make comments in blogs, contribute to bug-tracking systems - and, most importantly, make acquaintances, get to know our colleagues by names - then it will truly be communication.

Afterword

I tried to insinuate in the Preface that we couldn't possibly cover even the most important issues in a limited amount of time. There are countless more technical and non-technical problems, we would hopefully enjoy to discuss, and I hope we will, eventually.

On the dark side of my message it is written that in spite of relentless activity, FreeBSD ports have not moved much forward during the last few years. On the bright side it says that we have always had fun doing whatever seemed right for us and our friends and users, and we have never been shy to expose our shortcomings, to acknowledge mistakes, to look into the future.

Let us look at the world and understand what has changed and what is changing. Let's accept the changes and react to them. Let's talk and listen to people we don't know, but who do the same work we do. Let's value each other's ideas, respect constructive action. And most importantly, let's have a great lot of fun doing it all together!

References

http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/handbook/ports.html
- FreeBSD Handbook: Installing Applications: Packages and Ports
http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/porters-handbook/index.html
- FreeBSD Porter's Handbook
http://www.NetBSD.org/Documentation/pkgsrc/
- The pkgsrc guide

ttp://www.OpenBSD.org/porting.html
Building an OpenBSD port
ttp://www.OpenBSD.org/ports.html
OpenBSD Ports and Packages
ttp://www.DragonFlyBSD.org/docs/goals.cgi#packages
DragonFly BSD Design Goals: Dealing with Package Installation
ttp://www.Gentoo.org/doc/en/handbook/handbook-x86.xml
Gentoo Linux x86 Handbook: Working with Gentoo
ttp://www.Gentoo.org/proj/en/devrel/handbook/handbook.xml
Gentoo Developer Handbook: Guides, Policies
ttp://devmanual.Gentoo.org/
Gentoo Development Guide
ttp://www.Gentoo.org/proj/en/portage/index.xml
Gentoo Linux Portage Development
ttp://www.Debian.org/doc/FAQ/index.en.html
The Debian GNU/Linux FAQ: Basics of the Debian package management system and other chapters
ttp://www.Debian.org/doc/debian-policy/
Debian Policy Manual
ttp://www.Debian.org/doc/manuals/developers-reference/index.en.html
Debian Developer's Reference: Managing Packages, Best Packaging Practices and other chapters
ttp://www.Debian.org/doc/manuals/maint-guide/index.en.html
Debian New Maintainers' Guide
ttp://www.Debian.org/doc/manuals/apt-howto/
APT HOWTO
ttp://slacksite.com/slackware/packages.html
Slackware Package Management
ttp://kitenet.net/~joey/pkg-comp/
Comparing Linux/UNIX Binary Package Formats
ttp://www.rPath.com/technology/techoverview/
Repository-Based System Management Using Conary
ttp://www.freestandards.org/en/LSB
Linux Standard Base
ttp://www.pathname.com/fhs/
Filesystem Hierarchy Standard
ttp://www.rpm.org/max-rpm-snapshot/
Maximum RPM (development snapshot)
ttp://fedora.redhat.com/docs/drafts/rpm-guide-en/
RPM Guide by Eric Foster-Johnson
ttp://fedoraproject.org/wiki/Packaging/Guidelines
Packaging Guidelines by Tom 'spot' Callaway
ttp://www.sepp.ee.ethz.ch/
SEPP - Software Installation and Sharing System by Tobias Oetiker

ast, but not least - dozens of Wikipedia articles on package management and related topics; over a hundred of manpages from SD, Linux, Solaris and other operating system, as well as those coming with third-party tools; countless discussions in ailing-lists archives; popular interviews, memoirs, essays concerning package management.

Interrupt filtering

Paolo Pisati
Dip. Scienza dell'Informazione
Universitá Statale di Milano
Milano, Italy
Email: piso@FreeBSD.org

Abstract— **Adapting the FreeBSD operating system to scale with the growing parallelism of today systems consisted of major redesign of its architecture. In particular, the kernel evolved around the concept of multithreading, with threads being used in every subsystem. Interrupts were threaded too but this solution, even if proved to be reliable and orthogonal with the rest of the system, showed different problems and thus others models were investigated.**

Index Terms— **Operating system, interrupt, thread, Unix, FreeBSD**

I. Introduction

The UNIX operating systems was not designed to run on SMP hardware, and FreeBSD was not an exception. The first approach to make FreeBSD run on SMP hardware was really simplistic [1], and consisted of a global spin-lock called the 'Big Kernel Lock'. The BKL covered the entire kernel space, and every processor trying to enter the kernel had to acquire it and, in case of contention, spin on it.

Obviously, the Big Kernel Lock was a big bottleneck for multiprocessor system, and with the growing parallelism in today systems, grew the necessity to replace the global lock with a better mechanism/policy to exploit the exceeding cpu resource available.

That's why, in 2001, building on the experience of other operating systems like BSD/OS and Solaris, the FreeBSD project started a major redesign of its source code with the ultimate goal to improve the parallelism inside the kernel through the addiction of fine grained locking: this project was called SMP Next Generation (or SMPng in short).

With SMPng, the FreeBSD kernel went under a major overhaul, and many subsystems changed significantly. In particular, the interrupt handling moved from a low level approach based on interrupt level rising/lowering, to a solution based around threads that better integrates with the rest of SMPng project.

Unfortunately, even if the interrupt thread model proved to be a reliable solution, its performance was not on par with the preceding scheme, that's why other solutions were investigated.

In the rest of this paper, i will focus on the evolution of interrupt handling in FreeBSD, how it changed as the kernel moved from a monothread to a multithread architecture, what were the advantages of every model and in the end i will introduce a new interrupt handling scheme designed around a two-level approach.

II. Principles of operation: interrupt handling, in general

While the hardware changed significantly with time passing, the principles behind interrupt handling didn't, and we could roughly summarize the steps necessary to serve an interrupt as follow:

- whenever a device needs attention, it generates an interrupt rising the level on the irq line
- a low level routine(ISR) associated to that line is run
- the IRS performs some janitorial work, and call the real interrupt handler
- the interrupt handler serves the interrupt and return
- the interrupt is turned off at the controller level (ACKed)

The ISR and the interrupt handler usually run in the context of the interrupted thread (but this changed lately with SMPng as we will see later), and the masking/unmasking of interrupts is omitted from the above explanation because it's implementation dependent.

What really changes across the different schemes presented below, is the synchronization mechanism used between the asynchronous part of a driver (the interrupt

Work partially supported by Google Summer of Code™ 2006

handler) and the rest of the kernel, and which mechanisms are employed to avoid races between the two parts.

III. Interrupt handling in 4.x

When FreeBSD had still a monothread nonpreemptive kernel (and hence only a processor was allowed to be in kernel space), the entire interrupt handling process was governed with the masking/unmasking of interrupts at different level. Through the accurate usage of a series of functions of the *spl* family, the kernel manipulated the interrupt priority level of the cpu, preventing interrupt handlers at a lower level to run.

With this mechanism based on lowering and rising of the interrupt level, the asynchronous part of a driver (the bottom half) and the synchronous part (the top half) were able to rule out each other from processing the same data structure at the same time, achieving this way a crude (but functional) synchronization.

A. Device Polling

Handling interrupts is not an overhead-free process because every time an interrupt arrives, the CPU has to save and restore its state, switch context, and pollute/flush caches around the invocation of the interrupt handler. Moreover, it's compulsory for the interrupt handler to serve the event that generated the interrupt before returning, because a second interrupt won't be generated for it. This scheme has the drawback that interrupt handling is not an upper bounded process, and in case of a frequently interrupted system, the kernel could consume most of the cpu time handling interrupts with little or no time left for user land tasks. In particular, a situation where the system is frequently interrupted arises when there's a high network traffic, and to cope with exactly this situation, a new interrupt handling scheme called "device polling" was introduced [2].

With device polling, the system saves the context switch time that occur during interrupt handling because the operating system can check frequently interrupting devices when it's already in kernel context (during a timer interrupt or at the end of a trap handling), turning interrupt handling from an asynchronous driven mechanism to a synchronous one. Moreover, device polling give the system the ability to carefully choose how much time to devote to handling interrupts, limiting the time spent handling a single event, and returning to it later after some user land processing.

IV. Interrupt handling in the SMPng era

The main goal of the SMPng project was to increase the parallelism inside the FreeBSD kernel, and to achieve that, a fine grained locking scheme was applied to the entire system, replacing the aging BKL [3], [4]. Unfortunately, the new locking scheme was not compatible anymore with the interrupt synchronization mechanism based on *spl*, and in particular, on SMP systems, masking an interrupt on a cpu doesn't prevent the other cpu to serve the same interrupt, and thus having two different threads running on two different CPU racing on the same data structure. Moreover, even if masking/unmasking interrupts between CPU was technically feasible, the solution would cost too much in terms of cpu time, and that's why a new solution had to be found.

Following the path already taken by the Solaris operating system [5], [6], FreeBSD decided to adopt a private thread context for interrupt handlers, giving them the ability to block (if necessary). Moreover, with threaded handlers, the distinctions between kernel threads and interrupt handlers ceased to exist, and all the mechanisms and policies previously developed for normal threads could be easily adopted for interrupt threads too:

1) locking mechanisms: to synchronize a kernel thread and an interrupt handler, a normal lock variable (like a sleep lock) could be used, turning a piece of code that previously required direct manipulation of the interrupt controller, into a pair of simple locking operations.
2) priority influence: to reduce the interrupt response time, the priority of interrupt threads was raised to real time.
3) cpu accounting: as well as all the threads in the system, even for interrupt threads it's finally possible to know how much cpu time they consume, through normal cpu accounting.

A. Kernel preemption

The interrupt threading approach used in FreeBSD is defined 'heavyweight', because to serve an interrupt a full context switch from the currently running thread to the interrupt thread is necessary, and this adversely affect the interrupt response time. In particular, in case the running thread is in kernel space, the interrupt thread had to wait until the interrupted thread would relinquish the cpu spontaneously(sleep) or exit the kernel, because threads running inside the kernel were unstoppable.

To mitigate the delay the interrupt thread could potentially incur, the kernel was made preemptible: when a new thread is made runnable, the scheduler checks the priority of this new thread, and if the priority is higher then the priority of the currently running thread, the currently running thread is stopped and the new thread can execute.

For performance reasons, the preemption implemented is not a full preemption but a simplified version, and two aspects in particular were modified from the pure theoretical model:

1) preemption works only for real time priority thread: currently running thread will be preempted only in case the new thread has a real time priority.
2) preemption works only on local cpu: to avoid thrashing caches with a ping-pong effect across different cpu, preemption won't move threads on different processors even if a better scheduling situation would result.

B. *Time critical handlers*

Even with a preemptive kernel, the latency of interrupt threads was too high for some time critical handlers, that's why some handlers were modified to run directly in the context of the interrupted thread: these handlers are called "fast handlers". Fast handlers don't have to wait for a new thread to be scheduled, and thus they can serve an interrupt quicker. On the other hand, fast handlers share the context with other threads so they can't do any operations that could potentially block (like allocating resources) nor use any blocking locks.

C. *Lightweight context switch*

A further optimization that could be implemented to reduce the overhead of a full context switch, consist in a lightweight switch mechanism called "context stealing". Since an interrupt thread runs in kernel context, it can borrow the vmspace of any interrupted threads without any restrictions. Thus, when an interrupt arrives, the interrupt thread instead of performing a full context switch, could just borrow the vmspace of the interrupted thread saving a considerable amount of time in the context switch process. Obviously, while using the vmspace of the interrupted thread, that thread is not allowed to execute (to avoid the vmspace to disappear) until the interrupt thread either blocks or finishes execution. In case the interrupt thread blocks, when it's resumed, it will use its own thread context, releasing the vmspace of the interrupted thread.

Context stealing is an attractive optimization that in theory could help to further reduce the time necessary for an interrupt thread to start servicing the event, but so far the benefits of this approach are still undemonstrated and further work is needed.

V. Problems with the threaded model

Threaded interrupts represented a reliable solution while the kernel moved from a monothread to a multithread design, but some aspects of this solution were less than optimal.

In particular, some of the issues presented by threaded interrupt handlers were:

- high response latency: with fast interrupt handlers unable to perform any blocking operation (and hence relegated in their usage to clock and serial interface handlers), all the other device drivers were converted to use a fully threaded handler, that allowed the driver to perform blocking operations, but on the other hand, introduced an high latency due to the action of scheduling a new thread every time an interrupt came in.
- contention with shared interrupt line: pci bus encourages sharing of interrupt lines, and this translates into different interrupt handlers contending for the same interrupt thread. A device driver sharing the same interrupt line with other drivers, potentially would have to wait all the other handlers to run/block/sleep before it has a chance to serve its interrupt event.
- no feedback from interrupt handlers: handlers don't return any information about the status of event processing back to the system, and this could be a real problem when hardware misbehave or when an unexpected event happens (like a stray interrupt).

A. *Interrupt masking at the controller level*

Last but not least, a problem that arised with the transition to interrupt threads, regards the acknowledging of interrupts at the controller level.

Pci interrupts are level triggered, that means before returning from the ISR, interrupt must be turned off or masked, else the irq line will stay triggered and the system will suffer an interrupt storm (and possibly a live lock). While this situation is not a problem when the interrupt is fully served before ISR returns (like in fast handlers case), it's an issue when the system has to schedule an ithread, and to wait for completion of

the ithread before turning off the interrupt. To solve this situation, it was decided to mask the interrupt line at the interrupt controller level before ithread scheduling happens, and unmask it after ithread has completed to run. This decision avoided to move interrupt handling in the drivers itself (and so it avoided modifying a lot of code), but proved to be unfortunate: due to buggy hardware, or simply interrupt controllers that don't fully employ specifications, sometimes it happens that masking an interrupt line provokes an 'aliasing effect' on another pin, while other times the controller simply ignores the masking command, and leave the line asserted.

To overcome the problems of threaded interrupt handlers listed so far, other solutions were investigated, with interrupt filtering being one of that.

VI. Interrupt filtering

While the concept of a multilevel interrupt handling scheme was already present in Mac OS X [7], details about the implementation were absent and the advantages of this approach were not obvious.

The rationale behind interrupt filtering is really simple, and we can summarize it here:

given a device driver, divide it's logic in two distinct parts (the filter and the handler). The filter runs in primary interrupt context, and its goals are to analyze the arriving interrupt, decide if it belongs to this device driver and, if possible, serve the interrupt. In case potential blocking operations are needed for interrupt handling, the filter will ask the system to schedule the handler in a new interrupt context, allowing it to block if necessary.

This scheme, with a two-level logic gives the best mix of advantages of the previous schemes, giving the driver writer the ability to perform some operations with little delay directly in the filter (like the fast handlers did), and later schedule a new thread to perform any other operations remaining and with no context restrictions (like a normal ithread).

A. Modification to the code base

The actual implementation of interrupt filtering in FreeBSD had the ultimate goal of exporting the new API while preserving all the code written so far, and to achieve that i tried to limit at minimum the modification to the source tree:

- fast handlers become filters: first and foremost, all the handlers that registered itself like **INTR_FAST**, were modified to have a return value.
  The possible return values for a filter are:
  - **FILTER_STRAY**: the filter analyzed the interrupt event, and it decided it doesn't belong to this device driver.
  - **FILTER_HANDLED**: the filter handled the interrupt, the operating system can acknowledge the interrupt line at the interrupt controller level, and go on processing the next event.
  - **FILTER_SCHEDULE_THREAD**: the filter recognized the event but further processing is needed, so it asks the system to schedule the handler in the ithread.

  No other return values are allowed together with **FILTER_STRAY**, and if a filter wants to schedule the ithread, it has to return the value **FILTER_HANDLED** | **FILTER_SCHEDULE_THREAD**. In case all the filters on an interrupt line returned **FILTER_STRAY** or there were no handlers registered at all, the line will be deasserted and masked at the interrupt controller level.
- newbus's *bus_setup_intr()* was extended: to accommodate the presence of a new function being part of a device driver, *bus_setup_intr()* grew a new parameter of type *driver_filter_t*. The new *bus_setup_intr()* now looks like (from *sys/kern/subr_bus.c*):

```
int
bus_setup_intr(device_t dev,
               struct resource *r,
               int flags,
               driver_filter_t filt
               driver_intr_t handle
               void *arg,
               void **cookiep
);
```

  and driver_filter_t looks like (from sys/sys/bus.h):

```
typedef int driver_filter_t(void*);
```

- **INTR_FAST** was retired: with the introduction of the filter argument to *bus_setup_intr()*, it's now possible to define an handler running in the interrupt context (previously called **INTR_FAST** handler) without using any additional flag, so the **INTR_FAST** flag was retired. Moreover, the PowerPC architecture that implicitly marked any **INTR_FAST** handler as **INTR_EXCL** too (then

forcing to have no more than one **INTR_FAST** handler per interrupt line) was fixed to support sharing the interrupt line by different interrupt handlers.

- part of interrupt handling logic turned into MI code: part of the interrupt handling logic that previously was machine dependent code (i.e. *sys/i386/i386/intr_machdep.c::intr_execute_handlers()*), was pushed into *kern/kern_intr.c* in the function *intr_filter_loop()* and is now machine independent code, and an attempt to turn more interrupt handling code into MI code is undergoing.
- stray storm mitigation: in case an interrupt appears on a line with no handlers, the system wouldn't know how to handle it, and to avoid the line to stay triggered (and hence the subsequent interrupt storm), the system would mask the irq at the interrupt controller level. To mitigate this situation, a new mechanism kicks in every time an interrupt fires on a line with no handlers. The new logic will simply unmask the irq line after a period of time, and check for a new handler that could take care of the event. In case a new handler is found, the line is unmasked and the handler can serve the interrupt. Otherwise, in case no handlers reclaim the interrupt, the line is masked again and the check will be repeated later.
- interrupt turned off before ithread scheduling: to prevent problems with interrupt controller misbehaving, after the execution of the filter function, the interrupt is acknowledged at the controller level, circumventing the necessity to mask the interrupt line (and thus avoiding the problems listed above about the interrupt controller).

VII. Conclusion

In conclusion, with the introduction of interrupt filtering, FreeBSD now has the following methods to handle interrupts:

- **FILTER** (replace fast interrupt handlers): runs in interrupt context, low latency but it can't perform any blocking operations (no sleep lock, no malloc(WAITOK), etcetc). Mainly used in time critical handlers like clock or serial line handlers.
- **ITHREAD**: runs in the interrupt thread context, doesn't have any restriction on operations, incur in high response latency and contention in case of shared interrupt lines. So far, the majority of drivers used this mode.
- **FILTER+ITHREAD**: the filter runs in interrupt context, while the rest of the handler (if needed) run in the interrupt thread context. Low latency response, no contention for shared interrupt line and a better handling of interrupt controller.

VIII. Future work

While interrupt filtering is an improvement over the plain ithread model, it's not a perfect solution and two aspects of interrupt handling could be further improved:

1) context stealing: even if a lot of work went into ithread scheduling, it's still an heavy task that adversely affect performance. In this scenario, a technique like context stealing could help, but it's effectiveness and its benefits are still to be demonstrated.
2) preemption effectiveness: preliminary analysis of the interrupt handling process showed a limited impact of preemption on thread scheduling latency, but a weird behavior when it came to reschedule the preempted thread.

IX. Acknowledgments

Thanks to John Baldwin and Scott Long for mentoring and helping me with this project.

References

[1] C. Schimmel, *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*, 1994.

[2] L. Rizzo, "Device polling support for freebsd." [Online]. Available: http://info.iet.unipi.it/ luigi/polling/

[3] G. Lehey, "Improving the freebsd smp implementation," in *Usenix*, 2001.

[4] J. H. Baldwin, "Locking in the multithread freebsd kernel," in *BSDCon*, 2002.

[5] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond multiprocessing: multithreading the sunOS kernel," in *Proceedings of theSummer 1992 USENIX Technical Conference and Exhibition*, San Antontio, TX, USA, 1992, pp. 11–18. [Online]. Available: citeseer.ist.psu.edu/eykholt92beyond.html

[6] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, and D. Williams, "Symmetric multiprocessing in solaris 2.0," in *COMPCON '92: Proceedings of the thirty-seventh international conference on COMPCON*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 181–186.

[7] L. G. Gerbarg, "Advanced synchronization in Mac OS X: Extending UNIX to SMP and real-time," 2002, pp. 37–45. [Online]. Available: http://www.usenix.org/publications/library/proceedings/bsdcon02/gerbarg.l

The Kylin Operating System

Qingbo Wu, Huadong Dai, Xiaojian Liu, and Hua Feng

(Department of Computer Science, National University of Defense Technology, China)

Abstract

Kylin is a server operating system focusing on high performance and security. It was first funded by the Hi-Tech Research and Development Program (863) of China in 2002. In this paper, we proposed a hierarchical kernel structure for Kylin operating system. In this structure, Kylin is organized as two layers. The basic kernel layer is responsible for initializing the hardware and providing basic memory management and task management. And the system service layer is based on FreeBSD 5.3 providing UFS2 file system and BSD network protocols. Kylin has been designed to comply with the POSIX standards and is compatible with Linux binaries. First we discussed the motivation of this novel hierarchical operating system kernel model. Then we introduced the kernel's infrastructure and key techniques about its security guarantees. Last, we presented the performance comparison of Kylin, Redhat 9.0 and FreeBSD 5.3 with standard benchmarks. Finally, we discussed our future directions.

1 Introduction

Operating system has made a considerable progress in the theory and practice since it was born in 1960s. With the emergence of Multi-core CPU [1] and new I/O architecture, computer hardware becomes more and more powerful and the nature of operating system becomes more complex. It is large amount of hardware resources and complex software levels that lead to complexity control of operating system and make OS technologies face enormous challenges. The Kylin Operating System's hierarchical kernel structure can effectively meet the high scalability, high performance and high security demands of network server operating system applications. It was funded by the Hi-Tech Research and Development Program (863) of China in 2002. Kylin is compatible with Linux binaries and can make full use of the rich application software from open source community to expand the application range.

This paper mainly studies how to improve the performance, security and scalability of Kylin from the structure design view of the operating system kernel. The rest of the article is organized as follows: section 2, the OS kernel technology; Section 3, hierarchical kernel design; Section 4, the security design; Section 5, performance analysis; Finally, conclusion and future work.

2 Related work on OS kernel structure

As the foundation of operating system, the structure of the OS kernel has a direct impact on performance, security and scalability of an operating system. The study of OS kernel structure has been a hot topic in the area of operating system.

There are two major server operating system kernels: monolithic kernel and micro kernel structure. Monolithic kernel is represented by traditional UNIX and Linux OS. These OS usually have a big kernel which includes almost all the functions, such as task management, memory management, file system, network and device drivers. And all of the functional modules share the same address space. Monolithic kernel OS is powerful and efficient, but its scalability is inadequate and unsuitable for network computing environment or the future requirements of distributed computing environment.

Compared with the monolithic kernel, Mach[2] and L4[3], as representatives of the micro-kernel operating system, provide only the simple functions, such as basic memory management, basic task management and inter-process communications. While other system functions, such as file system and network service, are provided by servers that running in user space. Micro-kernel structure is flexible, scalable and robust. It is also applicable to distributed computing environment. But due to frequent user/kernel space switch, micro-kernel OS usually run in lower efficiency than those with monolithic kernel. For the reason of efficiency, Microsoft, Apple and other

companies have been doing optimization and improvements against micro-kernel structure. In their newly coming products, such as Windows Vista and Apple's Mac OS , they put corresponding services originally running in user space into kernel space in order to improve the kernel performance.

3 **Hierarchical kernel structure designs**

Kylin operating system makes reference to other OS kernel technology, such as commercial UNIX, Linux, FreeBSD [4], Mach and K42 [5]. It satisfies the needs of high performance computing, network services and application security. Moreover, it takes the advantages of monolithic kernel and micro-kernel into consideration. Kylin uses hierarchical kernel structure composed of the basic kernel layer and the system service layer, as illustrated in figure 1.

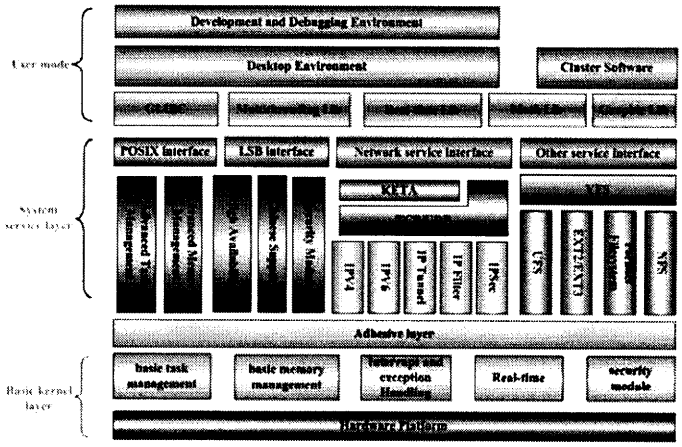


Figure 1 The structure of Kylin Operating System

The basic kernel layer is responsible for basic hardware initialization, basic task management, basic memory management, interrupt and exception handling and cryptographic services framework. The basic kernel layer provides abstract management system for hardware platform. It also supports functions of task management, interrupt processing and storage management for system service layer. The basic kernel layer uses modular design methods. With features of clear structure, weak dependence between modules and condensed kernel, it facilitates the maintenance and portability of the operating system kernel.

The system service layer makes improvement and optimization on FreeBSD. It provides users with industrial standard network and file system interface. It includes Linux binary compatibility module, high availability module, and various security mechanisms. At the same time, it fully maintains the stability and rich industry standard interface of BSD operating system.

The root environment is based on GNU/Linux. It uses X-Window as the basic graphics environment to support KDE or Gnome desktop environment. It is designed in Windows style desktop and control panel, providing simple and friendly installation interface. Besides, it provides graphic management tools and a variety of Chinese character input methods.

The basic task management module of the basic kernel layer is composed of main task scheduling, clock and timer, critical resource management. It provides the system service layer with a management interface for creation and destruction of tasks. The interface supports displaying tasks and setting scheduling parameters. The main task scheduling module provides resource reservation, scheduling interruptions, triggered scheduling, task distribution and context switching functions. Clock and timer module provides system clock, scheduling events production and timing response mechanism. Critical resource management module supports correct critical resources competition between multiple tasks, providing spin lock, mutex, condition variable and semaphore mechanisms, as shown in

figure 2.

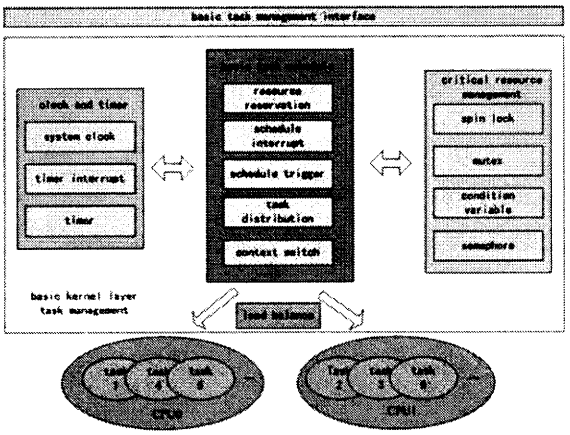


Figure 2 Basic task management

The hardware interrupt is captured and received by the basic kernel layer, and then is forwarded to the basic service layer or specific tasks of the basic kernel layer according to interrupt handling request. The basic kernel layer manages interrupt descriptor table which associates each interrupt vector with the address of the corresponding interrupt handler. To support efficient mechanism for transmitting the interrupt, the basic kernel layer of Kylin operating system introduces an optimized event mechanism to transfer interrupts. The basic kernel layer generates basic events, as well as SMP IPI events. The interrupt and exception handling mechanism of Kylin operating system is shown in figure 3.

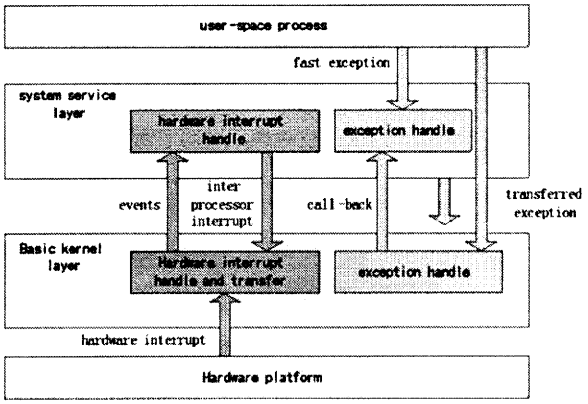


Figure 3 Interrupt and exception handling

The basic kernel layer and system service layer share the same address space by using a global mapping of physical address to linear addresses. The basic kernel layer provides page table update verification mechanism. But it does not support user level Pager to ensure efficient memory management. The system service layer has only read attributes to page table, all page table update operations must be validated by the basic kernel layer.

Cryptographic service framework is located in Kylin OS kernel, across the basic kernel layer and system service layer. Upwardly, it provides Encryption File System (SFS), SSL and other security services with cryptographic service through the API. Downwardly, it supports encryption algorithms development with a unified interface through the SPI. Framework that located in the basic kernel layer includes the dispatcher, Crypto-SPI, hardware cryptographic service provider and pure software cryptographic service provider. The system service layer is composed of Crypto-API, virtual cryptographic device driver and cryptographic service daemon. These two parts are blended into a whole to provide interfaces that include Crypto-API, Crypto-SPI and virtual crypto

device.

4 Security design

The Kylin first implements traditional operating system security mechanisms. On the other hand, it designs and implements characteristics security mechanism The structure of Kylin security system is shown in figure 4

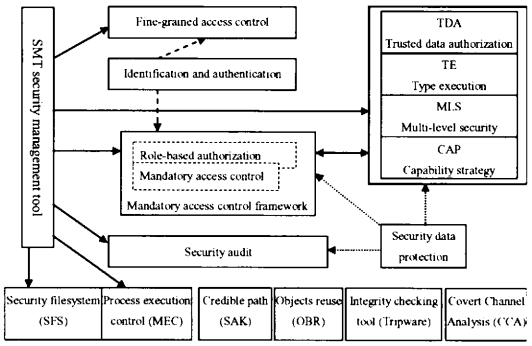


Figure 4 Security structure

The Mandatory Access Control Framework is constituted by RBA and MAC with the combination of MLS, CAP, TE and TDA strategy. AUDIT is responsible for the security audit: activating audit services, completing audit records. ACL uses access control list to achieve fine-grained access control. OBR reuses memory and hard disk objects. IDENT is in charge of user identifier and authentication for RBA and ACL. SFS manages encryption data of file system. MEC can control the process image loading when process is running. Trust path provides safe attention key SAK and starts trust login process. Tripware is a tool for file integrity checking. SDP protect cryptographic data, including audit data, mandatory access control attributes and user authentication data. SMT provides security management tools for easy security configuration and management which includes mandatory access control management, defining role security strategy attribute, defining file mandatory access control security attribute, defining file access control list, defining audit configuration and audit log files. CCA analyzes and processes the convert storage channel. Next, we will introduce two characteristics security mechanism in our security structure.

4.1 Encrypted file system

Encrypted File System introduces the encryption service for file system to prevent the lost of hard disks and other problems. Kylin integrates encryption services into file system so that the file can be automatically encrypted and decrypted. And the encryption and decryption process is transparent to users. If the unencrypted files are copied or moved to encrypted directory, then they will be automatically encrypted. The encrypted files will be unencrypted when legal users use them.

The Encryption File System framework is shown in figure 5. It mainly consists of two modules: safe deposit box management module and encryption filter module. Safe deposit box management module provides users with safe management functions (create, delete, update, etc.) through proprietary system call interface. While encryption filter module embeds in VFS layer through several hook functions. It automatically identifies if the files operated by users are in safe deposit box. If true, encryption is made before writing or decryption is made after reading according to safe deposit box attribute.

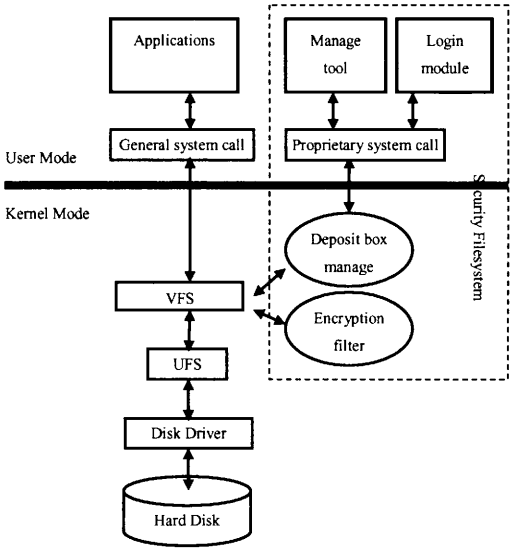


Figure 5 Overall framework of encryption file system

Encrypted File System segregates users by providing a key (public and private key) for each active user. The public key is stored in the system, while the private key can only be seen by its owner and can be stored in a smart card. Each encrypted file has a key that is generated randomly. And the key is encrypted by public key and is stored in the file's extended attributes.

4.2 Role based authorization

Kylin design role based authorization with privilege and user with role. Different from traditional RBAC, Kylin associates RBAC with mandatory access control. A role's privilege includes CAP attributes which determines role privilege and MLS attributes which determines role security level that is the top confidential information represented by a role. In addition, the current role of our system may influence the conversion of process type in TE strategy.

Through role based authorization control, there is no super user in Kylin who can do whatever he want. The privileges of super user are split into three types: system manager, security manager and audit manager. System manager is responsible for the daily management such as user management and network management. Security manager is responsible for the work related to safe configuration and management, including creation and deletion of the role, setting and modifying role's privileges, setting and modifying of the user role, modifying file security attributes, distributing and managing smart cards. Audit manager is responsible for work that related to security audit work.

Role based authorization control structure is shown in figure 6. Chain initialization functions read role definition file and role attribute file to construct global role authentication chain in kernel initialization, and read user role configuration file to construct global user role chain. The PAM module will read these two global chains to set label of session process when a user logins in. And it will determine the MAC label of a process through process label setting functions when the user is executing commands. The external interface operates on role authentication chain and user role chain by calling role configuration and user role correlation functions. It also deals with role definition file, role attribute file and user role configuration file.

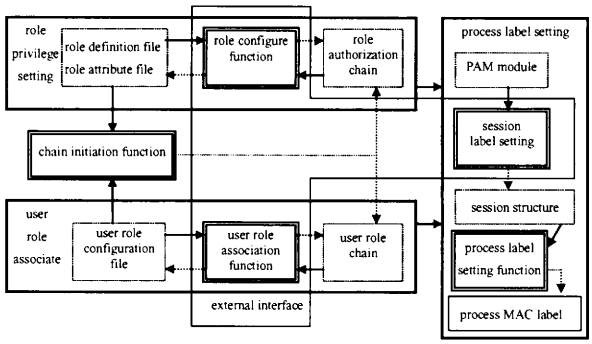


Figure 6 RBA structure

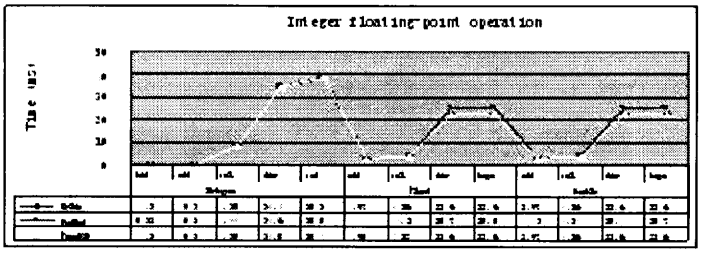
Through role based authorization control technology, security manager can create new system role such as roles with different privileges to satisfy the needs of certain role privilege and roles with different security level to satisfy certain role security during runtime of Kylin operating system. Through role based authorization control technology, Kylin implements a flexible user authorization.

5 Performance Analyses

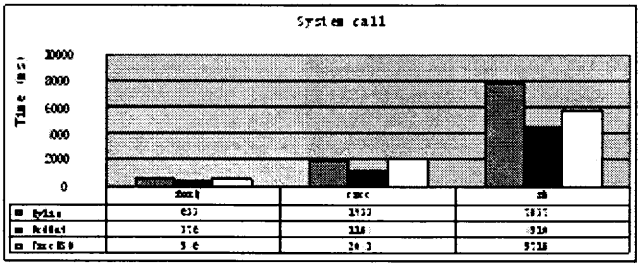
Kylin operating system has made some progresses. This section shows the analysis of basic performance evaluation of Kylin. We also test and analyze the web performance.

5.1 Basic performance tests

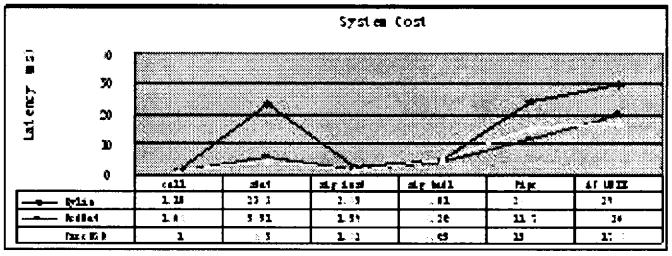
LmBench is a set of utilities to test the performance of a UNIX system and is very famous in open source community. It includes a series of benchmarks, i.e., integer floating-point operation time, system call overhead, file system and virtual memory, system bandwidth, context switch time and so on. The test results are running on two-way Xeon 1.7GHZ, 1G DDR memory, 36G SCSI disk, and 100M net card.



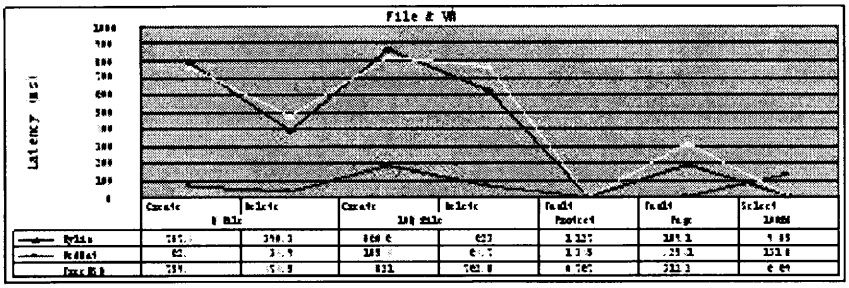
a) Integer floating-point operation time, the smaller the better



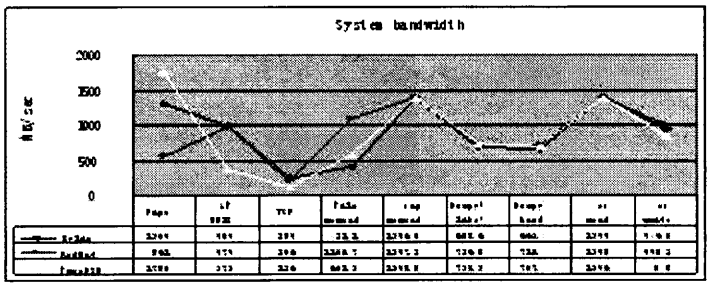
b) System call latency, the smaller the better



c) System overhead, the smaller the better



d) File system and virtual memory latency, the smaller the better



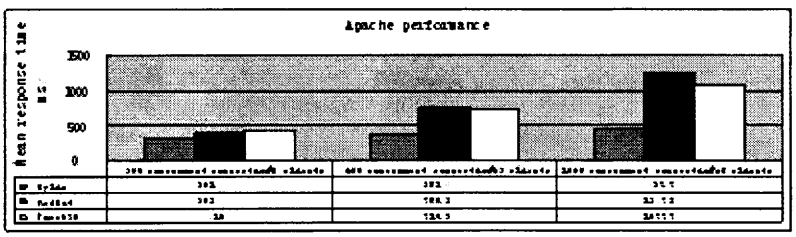
e) System bandwidth, the bigger the better

As the result shows, Kylin 2.0 is better than RedHat 9.0 and FreeBSD 5.3 in integer floating-point operation. They are almost the same in system bandwidth. Kylin 2.0 is as better as FreeBSD 5.3 in system call, file system and virtual memory latency. But there is still a gap between RedHat 9.0, the causes are as followings: 1) Kylin 2.0 and FreeBSD 5.3 adopt UFS2 file system, but RedHat 9.0 use Ext 3 file system; 2) Kylin 2.0 didn't directly use SYSENTER/SYSEXIT instructions that provided by X86 architecture. We will further optimize the file system performance and system call interface according to LmBench test results.

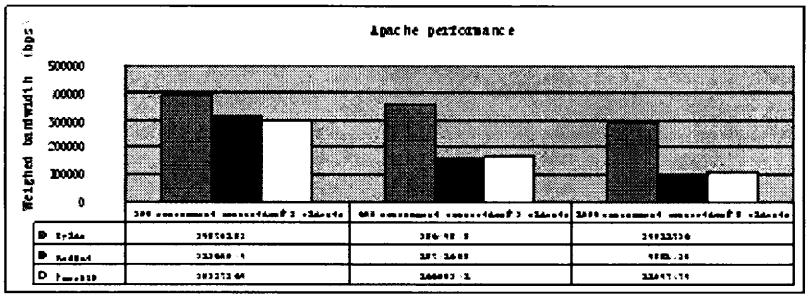
5.2 Web server performance test

SpecWeb99 is a benchmark for web server that is developed by Standard Performance Evaluation Corporation (SPEC). It measures the largest number of concurrent connections of web server that meets specific throughout and request-response rate. It will be vailid if Baud Rate of concurrent connections is in the area of 320Kbps to 400Kbps. The server is configured with 2-way 2.4GHz Intel Xeon, 2GB DDR memory, 36GB 1000 RPM SCSI disk, and 1GB net card. The client is equipped with 2.4GHz Intel Pentium IV, 512MB DDR memory, 40G disk, and 100 Mb network card. We use gigabit switching network.

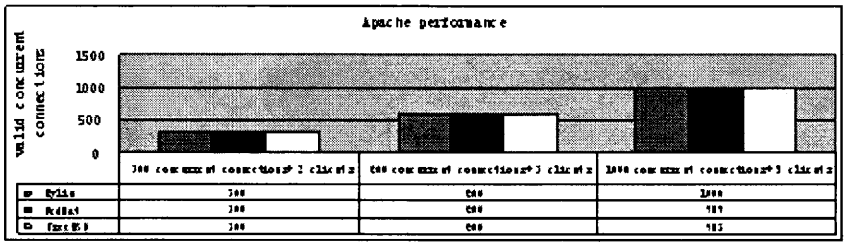
Test results are shown in following tables. The weighed bandwidth is computed by integrating sub-weighed bandwidth that is synthesized with all kinds of requests by SpecWeb99. Its data units is bits per second. Each connection can be understood as a separate user. Valid connection means the ratio of incorrect responses is relatively low, and invalid connection means that of incorrect responses is too high.



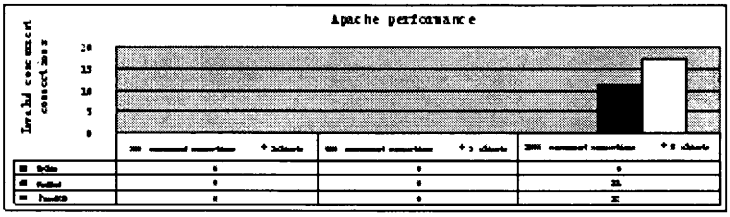
a) Mean response time, the smaller the better



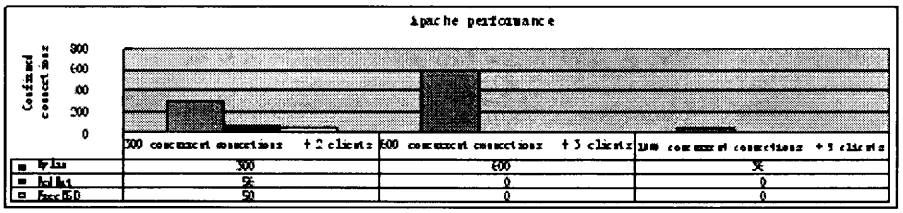
b) Weighed bandwidth, the bigger the better



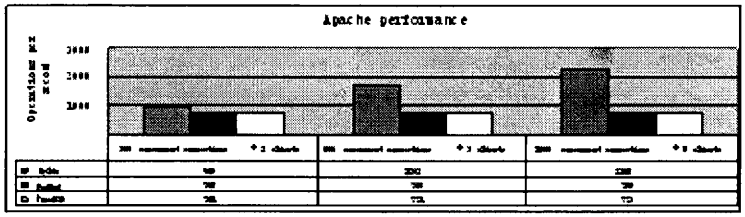
c) Valid concurrent connections, the bigger the better



d) Invalid concurrent connections, the smaller the better



e) Confirmed connections, the bigger the better



f) Operations per second, the bigger the better

From the test results, it can be seen that Kylin 2.0 is superior to Redhat 9.0 and FreeBSD 5.3 in Web performance.

6 Conclusion and future work

Kylin's hierarchical structure takes advantages of monophonic kernel and micro-kernel. It provides better scalability, stronger security, high availability and so on. Hardware initialization is implemented in the basic kernel layer. By using modular design methods, it can be easily ported to a new hardware platform. The traditional micro-kernel operating system provides small abstract core primitives, so the system service layer will be more

complex. Different from micro-kernel operating system, the basic kernel layer of Kylin provides richer interface primitives, and can flexibly support the services layer according to different needs.

The code of the basic layer of the hierarchical kernel structure is small which facilitates the formal verification [6]. It can construct trusted platform by combining with future trusted hardware platform [6] or security BIOS. And it shares the same idea with New Generation Computing Platform (NGSCB) of Microsoft. Currently existing security attacks tends to be directed against a specific operating system. For example, in the first half of 2005, the number of new virus that against Windows reaches nearly 11,000, which increased by 48% over the second half of last year. The hierarchical kernel structure of Kylin operating system can effectively defense against security attacks towards a specific operating system.

The basic kernel layer of our proposed hierarchical kernel structure is condense, while the system service layer is more complex. But the latter can be adjusted according to the application requirements, which is adaptable to current trend of advanced server architecture development, such as IBM's Capacity on Demand and Intel's Adaptive Server Architecture. Kylin meets the needs of dynamic resource allocation and Customization services of high-performance server, and forms a good foundation for realizing self-management and self-adaptive server system.

Now Kylin research group is improving the system performance and hardware adaptability. Our future research work includes dynamic resource management [7], higher available system [8], and higher security [9] to achieve high reliability and high performance in developing server operating system platforms.

References

1 Alexandra Fedorova, Margo Seltzer, Christopher Small and Daniel Nussbaum. Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design. USENIX'05, 2005

2 The L4Ka Project Home Page. http://www.l4ka.org

3 The Mach Project. http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html

4 Marshall Kirk McKusick, George V. Neville-Neil. The Design and Implementation of the FreeBSD Operating System. Addison-Wesley Professional, ISBN: 0201702452. 2004

5 Jonathan Appavoo, Marc Auslander, Maria Burtico, .K42: an Open-Source Linux-Compatible Scalable Operating System Kernel. IBM Systems Journal, pp. 427-440, Vol. 44, No. 2. 2005

6 Harvey Tuch, Gerwin Klein, and Gernot Heiser. OS Verification—Now!. HotOS X, Tenth Workshop on Hot Topics in Operating Systems. 2005

7 J. Jann, L. M. Browning, R. S. Burugula. Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers. IBM Systems Journal, Vol. 42, No 1. 2003

8 Martin Rinard, Cristian Cadar, Daniel Dumitran, et al.. Enhancing Server Availability and Security through Failure-Oblivious Computing. 6th Symposium on Operating Systems Design & Implementation. 2004

9 Tal Garfinkel, Ben Pfaff, Jim Chow, et al.. Terra: A Virtual-Machine Based Platform for Trusted Computing. 19th ACM Symposium on Operating Systems Principles. 2003

A Scalable Framework for Compact Flash Booting NetBSD Network Appliances

FINAL DRAFT: 10/16/2006

Brian A. Seklecki <lavalamp@spiritual-machines.org>
Collaborative Fusion, Inc. <bseklecki@collaborativefusion.com>

Table of Contents

Synopsis

BSD has long served as the reference platform for many new networking technologies. The first implementations of TCP/IP were developed on BSD and BSD led the way on IPv6 development. NetBSD set the Internet2 Land Speed World Record. Today BSD has the most robust, stable, secure TCP/IP stack of an F/OSS operation system. BSD development models and practices are the watchword of network security. Traces of BSD technology are found in dozens of commercially sold products. So why are organizations so slow to deploy BSD based network appliances internally?

First we must identify what defines a "Network Appliance", then we must explore why organizations prefer commercial products to internally developed solutions, and finally examine a project in progress to meet some of the existing challenges.

Challenges to Adoption

The challenges that exist to wide spread adoption of BSD based appliances by small to medium sized businesses are numerous. They range from availability of appropriate hardware platforms to political and policy conflicts within IT departments. A variety of BSD-**based** products on the market can be purchased, however non-commercial solutions tend to be excluded from consideration in all but a few F/OSS oriented organizations. When SMB IT managers consider solutions, arguments in favor of F/OSS adoption go unheard in light of counter-arguments in favor of commercial product features. This due in great part to the fact that the well known benefits of F/OSS software do not necessarily translate into quantifiable features that can easily be compared to those of equivalent commercial solutions on the market.

Appliance Defined

In the network hardware industry, the definition of "Network Appliance" is as loosely applied to both commercial and home-grown solutions as the term "Firewall" is applied to systems capable of forwarding TCP/IP packets. It is sufficiently ambiguous to describe any single/multi-function product a vendor choses to market. The term must be more clearly defined. For the purposes of this document, we will examine the attributes which we believe are differentiating factors:

- **Class of hardware:** Frequently OEM, low profile, compact or sub-compact server class hardware
- **Nature of the operating system:** Frequently a customized, embedded OS
- **Style of administration:** Frequently exposure to the underlying OS is restricted from the end user by a graphical user interface; often jointly administered by both the end user and the vendor.
- **Functions performed and/or services delivered:** Frequently delivers a discrete, specialized service, possibly on an ongoing subscription basis.

With the advent of web services, the definition of "appliance" is changing. Examples of appliances on the market today: E-Mail SPAM Filtering, Anti-Virus, Token Authentication, Intrusion Detection, Web Searching, & GIS. These devices are essentially compact, high-end servers installed within an organization's private network to function as a localized delivery point for a subscription service. However, for the purposes of this discussion, the term **"BSD Appliance"** will refer to a more discrete device.

> *Any combination of OEM hardware and a *BSD OS that serves in the place of a variety of traditional commercial network device (e.g, Router, Firewall, Access Point, etc.)*

Hardware Challenges:

In preparation for, the out outcome of a proposal to, or following the arbitrary decision to deploy a BSD appliance, a delivery platform that exceeds the expectations of the equivalent commercial product is needed. It helps to examine a very typical scenario an SMB IT professional may encounter if proposing or given authorization to deploy a BSD appliance within at their organization.

Scenario: Deploy a BSD based "firewall" for a branch office facility

Firewall Requirements:

- Serve as the CPE for the WAN connection (T1)
- Provide stateful firewall inspection
- Provide NAT+PAT and public Internet connectivity
- Support dial-on-demand backup ISDN connectivity or out-of-band remote dial-in administration
- Support HSRP LAN gateway configuration
- Transmit Syslog, IDS data, and SNMP traps to a central remote server destination
- Provide IPSec tunnel transit to the company headquarters
- Provide QOS and Traffic Shaping
- Integrate with centralized authentication (LDAP, RADIUS, etc.)

The defined appliance is acting in three traditional roles: A 'perimeter router', a stateful inspection 'firewall', and as an IPSEC 'VPN' termination point. However, note the lack of other LAN support services : DHCP, DNS, Transparent Proxy. This suggests that there is another host on the network already providing these services. The *BSD OS easily meets the software features requirements.

An SMB will frequently contact server hardware vendors and enter into the competitive bidding / solicitation process to acquire a hardware delivery platform. The end result of this process tends to be the acquisition of the most cost-effective server class hardware available. As a byproduct, the acquired hardware is a often a suboptimal choice for the delivery platform. Consider the obvious technical concerns:

Hardware Concerns

- Excessive RAM capacity, CPU speed, and Fixed Disk storage resources for the functions services delivered
- Consumes more space than necessary for the number of physical interfaces provided
- Consumes excessive power, Generates excessive heat, Omits excessive noise

Moreover, consider the scalability and management concerns that this ad-hoc hardware acquisition process might may have on growing businesses` IT department:

- **Hardware Reliability & Cost of Ownership:** Because server hardware is deployed as a network appliance, it is fallible to all of the standard hardware component deficiencies a standard server is: power supplies, hard disks, chassis/CPU fans, etc. Component failures, high MTBF rates, and support contract costs are integral to hardware vendors` business model. Component-level redundancy is assumed by server vendors, where as network vendors assume device redundancy. Network vendors often offer discounts for active/standby fail-over bundle purchases. Pricing of server hardware is also normally oriented around a different average product ownership life cycle. Pricing reflects related support contracts the vendor expects to incur; thus unit cost of the hardware platform is not apt for deployment in a large scale, specifically in redundant configurations.
- **System Administration Style:** Because the OS is installed onto fixed disk storage and administered in the fashion of a traditional UNIX-like server, it is vulnerable to the same attack vectors of a BSD server. Those include poor system administration practices and administrative neglect of patching and updating. This can result due to increased demand placed on IT staff due to additional installations and the overhead associated with synchronizing security policies across a potentially large volume of new systems.

- **Vendor OEM Hardware Continuity:** Because large server vendors frequently do not control the manufacturing processes at OEM manufacturers, there is no guarantee that the server vendor will not substitute underlying integrated hardware components and begin selling new systems with the same major model number under a minor revision (e.g., the onboard Ethernet controller could change). Such a change might inadvertently effect deployment schedules to accommodate for additional development and Q&A time. It may also compel discontiguous software revisions to be deployed where homogeneous systems are desired.

Commercial Products Advantages Contrasted

Hardware delivery platform limitations are obvious, however there are other general concerns that should be identified and addressed. The standard BSD system has historically been developed for use on server class hardware for use and installation onto fixed-disk based systems, therefore certain natural disadvantages must be overcome.

- **Hardware-optimized Network Operating System (NOS):** Commercial network operating systems (NOS) have the natural advantage of having been developed in a customer/profit oriented 'cathedral' model[2] to run as binary-only code optimized for a custom hardware platform. The combination of custom hardware and the specialized NOS constitute the product. In contrast, because BSD is a standards-driven, multi-platform system designed to run on a variety of platforms, much work to "profile" the system is required.
- **Reduced Memory Usage:** With commercial NOS, the memory footprint of the NOS is significantly smaller than the POSIX equivalent because only the kernel and a small CLI need to be loaded into main RAM. Secondary OS support services (such as NTP, SMTP, Syslog, etc.) are compiled as binary-only code and run as child processes of the kernel outside the control of the user. Moreover, utilities needed to manipulate and configure kernel interfaces and services can be omitted and replaced with a singular CLI. However, with all POSIX based systems deployed as NOS, a UNIX file system must also be loaded as a RAM-disk into system main memory. Steps must be taken to offset the impact by reducing the size of the utilities that must reside on such a RAM disk.
- **Unified Administration and Debugging Interface:** Another natural advantage of commercial NOS is that only a CLI is needed to configure all services. Most vendors` CLI configuration syntax is standardized syntactically in structure for all services within a single unit using similar nomenclature (e.g., the same ACL syntax is used to accomplish traffic filtering, define VPN peers, restrict SSH access, apply QOS policies, etc.). The CLI is also homogeneous across disparate products lines, permiting for simplified administrative tasks such as configuration-comparison, backup & recovery, and replication. Additionally, the debugging mechanism is also unified, guaranteeing the ability to easily debug any service in a unified manor. Unfortunately, due to the 'evolutionary' nature of POSIX systems, almost all system services have a different style of configuration syntax and debugging mechanism. Attempts to create abstraction layers and unified configuration repositories (such as GUIs and macro-languages) tend to restrict configuration flexibility.
- **Configuration Abstraction:** NOS vendors strive to limit end user awareness of the underlying operating system internals. Via the CLI, the end user is only aware of the configuration and the OS version installed onto permanent storage. The operating system usually consists of one binary image file and the configuration in ACSII format. This creates an abstraction between "configuration data" and "operating system". This arrangement eases deployments of similar or identical configurations.
- **Proprietary Hardware Acceleration Technology:** Commercial vendors unconcerned about hardware/software interoperability outside of a product line are at their discretion to move certain software functions from software to hardware, such as accelerated Ethernet packet switching using specially designed ASICs and system-in-a-chip design for kernel function offload. An excellent example of a platform-neutral hardware/kernel interfaces have been developed for accelerated IP/UDP/TCP MD5 network checksumming, VLAN tagging, cryptography, and other functions. Future work may look for ways to accelerate stateful packet inspection.
- **High Availability (HA) and Advanced Features:** Modern BSD systems feature many of the features previously only found in commercial projects: VRRP, stateful packet filtering, packet sanitation, integrated routing/bridging, state table synchronization, IPSec SA synchronization, VLAN routing, QoS, interface aggregation, WAN interfaces, serial console, BGP/OSPF routing, modularized authentication, cryptographic acceleration, VoIP support. Where features are lacking are in the realm of "administrative mechanisms" that commercial NOS excel at (e.g, SNMP MIB for specialized data structures, SNMP traps, unified debugging mechanisms)
- Standards Compliance: Commercial products are subjected to rigorous scrutiny during the product QA process to ensure that they meet industry and government security and reliability guidelines standards. Fortune 500 Enterprise clients and, Telcos, and Government agencies may also stipulate compliance in order to bid for contracts. Examples include DoD Security Guidelines, NEBS environment, RoHS hazardous materials, FCC interference, EIA/TIA etc. Some of these issues can be addressed by choosing the proper delivery platform, others require long and involved application and

testing.

- **Training & Certification:** Most NOS vendors feature a variety of professional certifications curricula to help customers address the human resource challenges of network growth. The result of a lack of a Unified Administration Interface and Configuration Abstraction is a BSD environment is a demand for highly specialized administrative talent: UNIX administrators familiar with network technology. Larger organizations concerned about ongoing human resource support needs for specialized technology may tend to shy away from adopting technologies for which a dearth of talent may exit. As an offset, projects such as the BSD Certifications project can offer companies a train and guage by which potential employees exits.

Summary:

Given the challenges (hardware platforms, operating system optimization, cost of ownership, life cycle management, scalability, manageability, platform homogenization, service assurance) the decision often becomes one of vendor loyalty. However, if an organization has sufficient resources to allocate, many of these competitive arguments can be addressed and resolved. No single project can address all of the challenges, however this project tackles a select few. It is the combined responsibility of the members of BSD community who have a vested interest to ensure the going viability of the system.

The System

This document outlines a set of technologies developed to adopt the NetBSD operating system into a NOS optimized for Compact Flash booting network appliances. The resulting network operation system optimized for routers, firewalls (stateful policy routers), access points, IDS Sensors, DNS slaves, Terminal Servers (RAS/Console), Environmental Sensors, NAS, and other network appliances.

Background: Original soekris256 Project

The original soekris256 scripts[1] simplified the process of translating the file systems for the target CF booting OpenBSD system into "RD" and "CF" file system images using the **vnd(4)** mechanism. However, they did not:

- Provide a mechanism to automate the building of file-system hierarchies that would reside within said images.
- Provide a mechanism for stateful configuration data (the ability for changes to */etc/** and */usr/pkg/etc/** to persist through reboots)

These two limitations are addressed using native mechanisms within BSD.

Primary Goals:

- Address previously outlined deficiencies and concerns that give commercial products a competitive edge
- Identify OEM hardware solutions that address the outlined hardware deficiencies
- Develop a framework of utilities to simplify deployment and management of code
- Optimize the system to realize the advantages of Compact Flash storage
- Implement IOS-like "configuration" v.s. "system" abstraction
- **Secondary goals:**
  - Minimize the customizations and deviations from standard installed NetBSD system
  - Integrate as much work as possible back into the NetBSD project minimizing the number of manual patches that must be maintained and re-applied when deploying a system. Where possible, add functionality as optional features in the main source tree.
  - Provide for a semi-stateless operating system which can be administered in a manor similar to that of a commercial NOS and yet is not foreign to BSD administrators.
  - Simplify upgrade and maintenance process by allowing for binary updates.
  - Reduce the maintenance involved in adapting new code trains by minimizing userland / kernel deviations and documenting such changes where possible

Terminology Reference:

- *SMB:* Small to Medium Size Business
- *soekris256* - a set of scripts written by Gray Watson for booting OpenBSD diskless on a Soekris hardware platform and the origin of *rd_image.sh* and *cf_image.sh*. The original inspiration for this work.

- *RD/MD* (Ram Disk / Memory Disk) – **rd(4)** is the OpenBSD in-kernel root file system, **md(4)** is the NetBSD equivalent; used interchangeably here to refer to a systems that boot from an image.
- *CF* – Compact Flash
- NOS – Network Operating System
- *build.sh* – The NetBSD system bootstrap system
- *$DESTDIR* – directory defined in NetBSD *src/build.sh* build system as specified by the "-D" flag

Format Reference:

- $VARABLE – An environmental or **mk.conf(5)** variable
- **manpage(4)** – A reference to any object in the system: config file, command, API, system call, C function, etc, by its respective man page entry.
- */path/to/dir/filename* – Any file system path
-

 # command
[output] - Interactive commands and results to **stdout**.

- "-Z" - flag to a command

OEM Hardware Delivery Platforms:

Commercial vendors feature highly customized hardware solutions for their products. At present, however, the major BSD projects exhibit a dearth of information and resources for helping ISVs, Systems Integrators, and SAs to use as a reference hardware platforms. The result is the Server-as-a-Router conundrum described in the scenario above. Fortunately, there is no shortage of comparable hardware available; many OEM vendors develop custom chassis to enclose single board computers (SBCs).

Frequently these SBCs are based on next generation low-power CPUs designed for industrial and embedded applications: **Intel Xscale**, **Pentium-M**, **VIA C3**, **AMD GEODE**, and **SIS 55x**. OEM vendors provide an array of services such as chassis branding and customization, OS image burning, and global distribution channels. Specifically, vendors offer products ideal for deployment in both Commercial (19" rack-mount) and residential environments (low profile set-top enclosures).

For set-top, residential class appliances, pricing can be cost prohibitive when compared to equivalent commercial products. Large commercial entities are able to use market power to implement low-cost manufacturing and distribution processes and save costs. This situation is unlikely to change due to the business structure of such companies.

For commercial grade solutions, however, hardware pricing is extremely competitive. Moreover, OEM solutions offer the features resellers expect from commercial class network devices:

- Commercial hardware and support and service availability
- High interface density
- Modular hot swap components: Power Supplies, Interfaces, Feature Cards
- Low Power, Fanless (set top) CPUs
- Flash Storage/Booting
- Resiliency to hostile environments
- 19" EIA/TIA Rack Mounting
- RS232 Serial Console
- DC Power Capable
- Standards Body Certification, Safety Standards, and Regulatory Compliance
- Guaranteed interoperability in a homogeneous vendor environment
- Predicable and tested MTBFs

Note: SBC/OEM hardware frequently feature smaller maximum memory capacities than server class hardware of the same CPU family, but comparable limitations to commercial network hardware. However, OEM hardware RAM (industry standard DDR frequently) is significantly less expensive than RAM for commercial systems. Memory pricing for commercial network hardware is frequently determined based on discrete capacity limitations of users/sessions/features/software versions that exist at certain memory installation sizes.

Commercial network hardware vendors tend to price hardware based on associated costs: ongoing support, interface modules, major software features. With OEM/SBC hardware platforms, associated costs are limited hardware replacement support and staffing human resources needed to maintain software. Hardware

costs can savings offset any NOS feature deficiencies.

A Booted System

To understand the system, it helps to examine the output of **dmesg(8)** and **df(1)**:

```
>> NetBSD/i386 BIOS Boot, Revision 3.2
> booting hd0a:netbsd
6392696+10868324+313008 [354272+323612]=0x116983c
Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004,
2005
    The NetBSD Foundation, Inc.  All rights reserved.
Copyright (c) 1982, 1986, 1989, 1991, 1993
    The Regents of the University of California.  All rights reserved.
NetBSD 3.0_STABLE (CFRDMDROOT.MPACPI) #0: Sat Jun 17 01:16:42 EDT 2006
        root@thunderwing:/nbsd/objdir/sys/arch/i386/compile/CFRDMDROOT.
MPACPI
total memory = 511 MB
        [...SNIP...]
boot device: wd0
root file system type: ffs
NetBSD Pre-OS MD/RD Bootstrap Starting:
Running version: $rc.patch 55 2006-07-05 11:40:45Z seklecki $
Image Built: Fri 07/28/06 08:29:55 -0400 EDT by root
Re-mounting /dev/md0a as / (root/slash) in RW
Making /usr MFS filesystem
Checking /dev/wd0a
** /dev/rwd0a
** File system is clean; not checking
Mounting /dev/wd0a on /cf
Populating static /usr from archive on wd0a
  100% |***********************************| 150 MB  3.63MB/s  00:00
ETA
Checking /dev/wd0b
** /dev/rwd0b
** File system is clean; not checking
Mounting /dev/wd0b on /shadow
Resuming standard startup rc(8):
Sat Jul 29 17:25:34 EDT 2006
        [...SNIP...]
NetBSD/i386 (sin) (console)
login:
```

A disk capacity **df(1)** output from a booted system illustrates the general file system layout. The booted file system system appears visually identical to a traditional system, however a few foreign mount points are visible here. The mount table is cluttered due the necessity of both **md(4)** and **mfs(4):**

```
$ df -h
Filesystem                Size       Used      Avail Capacity  Mounted
/dev/md0a                 9.6M       8.8M      296K    96%    /
mfs:16                    194M       167M       18M    90%    /usr
mfs:413                   1.8M       512B      1.7M     0%    /tmp
/dev/wd0a                 489M       182M      282M    39%    /cf
/dev/wd0b                 465M       376K      441M     0%    /shadow
```

Note: Near-capacity file systems are not a concern because (*/*) and */usr* will not actually sustain write activity. Additional customizations are made to prevent growth of the file system.

Unlike the original soekris256 project, this system divides the CF media into two UFS slices within a single NetBSD MBR partition. The first slice ("a") is 256 or 512MB, sufficient in size to hold:

- the RamDisk enabled Kernel
- *usr.tgz*,
- the 1st , 2nd stage boot loaders.

The second slice ("b") contains the directory hierarchy mounted on the live system as */shadow* (both *usr.tgz* and */shadow* described later)

Memory Demands

The above system illustrates an entire NetBSD distribution loaded into an large MFS */usr* mount. In design theory, this is an egregious misappropriation of system resources and is shown here only to illustrate one major shortcoming: the demand for RAM traditionally unneeded in commercial NOS environments. The kernel, the system's primary service processes, the file I/O cache, and the MFS mount are forced to share the commodity resource of RAM.

Fortunately:

- File systems growth is controlled
- The **md(4)** and **mfs(4)** file systems contents can be pruned precisely for your environment, eliminating any wasteful allocation of memory to MFS file systems.

Persistent Configuration Data

With the exception of a limited few security conscious deployments[1], a solution is needed to address the static */etc* file system in the original soekris256 system. When utilizing **md(4)**, */etc/** resides within a read-only file system image embedded into the kernel. Without the ability to have persistent configuration data, system administration becomes very cumbersome as a new CF image must be burned for every subtle configuration change.

The */shadow* file system provides a location to mirror the read-only */(root)* and */usr* mount points of the MFS/RD file systems with limited read-write abilities. The contents of the MFS */etc* and */usr/pkg/etc* consist of symbolic links to their equivalents in */shadow/${path}*, thereby certain system-specific configurations will be persistent across reboots. Specifically network configuration, authentication, startup settings, daemon setting, etc.

An illustration of symlinks:

```
lavalamp@deadset:/$ ls -laF /etc /usr/pkg/etc |egrep -i "\@"
lrwxr-xr-x   1 root  wheel         24 Jun 21 02:38 daily.conf@ -> ../shadow/etc/daily.conf
lrwxr-xr-x   1 root  wheel         24 Jun 21 02:38 dhcpd.conf@ -> ../shadow/etc/dhcpd.conf
lrwxr-xr-x   1 root  wheel         19 Jun 21 02:38 fstab@ -> ../shadow/etc/fstab
lrwxr-xr-x   1 root  wheel         19 Jun 21 02:38 group@ -> ../shadow/etc/group
lrwxr-xr-x   1 root  wheel         19 Jun 21 02:38 hosts@ -> ../shadow/etc/hosts
lrwxr-xr-x   1 root  wheel         24 Jun 21 02:38 ipsec.conf@ -> ../shadow/etc/ipsec.conf
lrwxr-xr-x   1 root  wheel         29 Jun 21 02:38 localtime@ -> ../usr/share/zoneinfo/EST5EDT
lrwxr-xr-x   1 root  wheel         22 Jun 21 02:38 ntp.conf@ -> ../shadow/etc/ntp.conf
lrwxr-xr-x   1 root  wheel         20 Jun 21 02:38 passwd@ -> ../shadow/etc/passwd
lrwxr-xr-x   1 root  wheel         21 Jun 21 02:38 pf.conf@ -> ../shadow/etc/pf.conf
lrwxr-xr-x   1 root  wheel         21 Jun 21 02:38 rc.conf@ -> ../shadow/etc/rc.conf
lrwxr-xr-x   1 root  wheel         25 Jun 21 02:38 resolv.conf@ -> ../shadow/etc/resolv.conf
lrwxr-xr-x   1 root  wheel         25 Jun 21 02:38 sysctl.conf@ -> ../shadow/etc/sysctl.conf
lrwxr-xr-x   1 root  wheel         25 Jun 21 02:38 syslog.conf@ -> ../shadow/etc/syslog.conf
lrwxr-xr-x   1 root  wheel    29 Jun 21 02:45 ntpd.conf@ -> /shadow/usr/pkg/etc/ntpd.conf
lrwxr-xr-x   1 root  wheel    30 Jun 21 02:45 snmpd.conf@ -> /shadow/usr/pkg/etc/snmpd.conf
lrwxr-xr-x   1 root  wheel    27 Jun 21 02:45 sudoers@ -> /shadow/usr/pkg/etc/sudoers
```

NOTE: */shadow* is mounted read-write with the "noatime" flag to reduce write operations.

Mount Table (some lines omitted) from **mount(8)** w/ "-v":

```
/dev/md0a on / type ffs (local, root file system, reads: sync 6321 async 0, writes: sync 228 async 106)
mfs:16 on /usr type mfs (synchronous, local, reads: sync 227694 async 0, writes: sync 86929 async 232)
mfs:413 on /tmp type mfs (synchronous, local, reads: sync 6 async 0, writes: sync 0 async 2)
/dev/wd0a on /cf type ffs (read-only, noatime, local, reads: sync 5 async 0, writes: sync 0 async 0)
/dev/wd0b on /shadow type ffs (noatime, local, reads: sync 101 async 0, writes: sync 3 async 4)
```

NOTE: Notice the low read-to-write ratio on */shadow.*

There are two major limitations to */shadow* (both of which are explored later):

- **chroot(2)** environments
- Database-Style Persistent Configuration Data

Read-Only Compact Flash & I/O Operation Reduction

With any compact flash booting system, it is critical to minimize the number of write operations to the CF media as to extend the life of the media. Newer CF media is more resilient to write operation induced failures[2], however it is still not advisable to have a live UFS file system operating upon it considering the overhead of physical memory paging/swapping and *atime* modification.

The operating system boots from the CF media and populates the root file system into an MFS read-write partition, after which the boot partition it is mounted read-only for administrative purposes only.

Database type Persistent Data

Some data (such as runtime databases) is constantly being written to/from disk. An example would be the **dhcpd(8)** / **dhclient(8)** lease database, the **ntpd(8)** drift file, the Net-SNMP **snmpd(8)** exec-cache, and the ISC BIND **named(8)** slave zone cache. In these cases, a traditional symlink out of */var* to */shadow* would incur too many write operations on the CF media.

As an alternative solution, periodic event-based synchronization of and in-MFS copy with the on-CF copy in */shadow* occurs. The ***rc.d/*** script mechanism provides a convenient way to move the respective copy to temporary/permanent storage upon a startup/shutdown event. This approach is illustrated in *dhclient.diff* and *dhcpd.diff* (abridged below). For **named(8)**, the automatic re-retrieval of all slave zones at boot time and incremental updates based on "NOTIFY" messages suffices.

A **diff(1)** against /etc/rc.d/dhcpd:

```
--- /home/nbsd/src/etc/rc.d/dhcpd       2004-08-13
14:08:03.000000000 -0400
@@ -13,7 +13,39 @@
pidfile="/var/run/${name}.pid"
+
+shadow_path=/shadow/var/db
+local_path=/var/db
+filename=dhcpd.leases
+
+#required_files="/etc/${name}.conf /var/db/${name}.leases"
+required_files="/etc/${name}.conf  ${shadow_path}/${filename}"
+
+start_precmd="dhcpd_precmd"
+stop_postcmd="dhcpd_postcmd"
+
+dhcpd_precmd() {
+
+       cp -fp ${shadow_path}/${filename}
${local_path}/${filename}
+
+       if [ $? -gt 0 ]; then
+               echo "Failed to copy on-disk copy into MFS RD."
+       fi
+}
+
+dhcpd_postcmd() {
   [....]
+}
```

This example illustrates a locally-maintained diff that utilizes the routines provided by **rc.subr(8)** can easily overcome the problem of Database-type Persistent Data. The disadvantage to his approach involves data loss in the event of a non-graceful shutdown. In that event, **cron(8)** can be used to periodically synchronize.

System Resources and Binary Only Code

A side-effect of running diskless, fanless, low-power hardware is that traditional system administration tasks, such as on-host software compilation, do not apply due to resource limitations (disk, CPU, memory). This compels the use of binary-only redistributed packages for tasks such as patching and upgrading components of the system. The system designed to simplify generation of binary images is described in the System Architecture section.

Mount: */cf and Remote Upgrading Procedures*

The /cf mount point is a legacy idea from the original soekris256 scripts. On the new system, however, it provides two important functions. It eliminates the need to maintain a copy of the kernel file in the */(slash)* of the MFS file system by allowing */netbsd* to be a symbolic link to */cf/netbsd:*

```
# mount -o update,rw /dev/wd0a /cf
$ ls -alF /netbsd
lrwxr-xr-x  1 root  wheel  12 Jul 28 08:29 /netbsd@ -> ../cf/netbsd
$ ls -al /cf
total 343243
-r--r--r--   1 root  wheel       53820 Jul 27 15:55 boot
-r--r--r--   1 root  wheel        6144 Jul 27 15:53 bootxx_ffsv1
-r--r--r--   1 root  wheel         512 Jul 27 15:50 mbr
-rwxr-xr-x   1 root  wheel    18056720 Jul 28 09:39 netbsd
-rw-r--r--   1 root  wheel   157495296 Jul 28 08:34 usr.tgz
```

Note: This is important because the the average profiled NetBSD kernel size (2mb) becomes significantly bloated when an **mdsetimage(8)** appends a **vnd(4)** to it.

/cf is normally mounted read-only, however it also provides a mechanism by which the system can be self-upgraded by simply pushing/fetching new out a new version of files: */cf/netbsd* and */cf/usr.tgz.* This allows bug fixes and minor updates to be applied without the need for an on-system toolchain. This also partially implements system v.s. configuration abstraction.

Architectural Discussion

Choice of NetBSD System

NetBSD was chosen over OpenBSD and FreeBSD for the ***initial*** development of this project for a number of reasons, but mostly due to maximum flexibility and modularity of the system. This flexibility reflects the wealth of collective knowledge gained from historical development oriented toward embedded platforms. Those feature will be expounded in greater detail:

- **Embedded Platform Availability:** NetBSD's emphasis on platform independence makes it the watchword of portable F/OSS operating systems. "Production Quality" ports of NetBSD to embedded architectures such as ARM, XScale, PowerPC, MIPS, Super-H, and i386 clones makes it an optimal choice for ISVs.
- ***build.sh* Cross-Build Bootstrap Environment:** The *build.sh* cross build environment allows for development of code for embedded CPU platforms to be cross-compiled for the target platform on high-speed development systems saving costly licensing fees for embedded system development tools and compilers (eg., Altera, ARM).
- **Three-Tier Release Engineering Tree:** The tree-tiered release engineering branch gives systems integrators an increased variety of options for tracking the NetBSD development cycle against their internal release engineering cycle.
- **Formal Security Process:** The formal security process including a security officer and formal, well tested advisories are an important step in tracking of security issues and integrating fixes into internally maintained CVS trees
- **Pkgsrc:** Pkgsrc offers the perfect balance of stability and variety of 3rd party software package systems.
- **System Size:** All BSD systems are extraordinarily minimalistic, specifically NetBSD and OpenBSD citing size as a project primary goal, however NetBSD provides additional mechanisms to further reduce build size of the kernel and userland.

Default $DESTDIR sizes w/o kernel for various *BSDs with generic build environment:

System	*Size*
FreeBSD/i386 6-STABLE	149M
NetBSD/i386 3_0_STABLE	203M
OpenBSD/i386 3.9-STABLE	209M

Note: These sizes reflect **du(1)** on $DESTDIR after "make buildworld", not an extract of release tarballs on a live system.

Even with the kernel, userland, and a number of Pkgsrc packages, the entire system could easily reside in an MFS partition read from a tarball archive on the CF medially totaling less than 150MB.

- **Build Size:**

- **Dynamically linked userland:** The conversion to a dynamically linked userland in the 2.x branch conserved a significant amount of space
- **Minimal 3rd party in-tree / mk.conf(5) modularity:** A number of 3rd-party packages compose the default "base userland" of the system. Additional some kernel sub-systems and respective in-userland utilities are imported from 3rd parties as well. A significant number of these can be "conditionally" compiled:

NetBSD 3rd in Tree Party Packages :

GNU/Dist/Contrib	***Modularity***	***Sub-System***	***Modularity***
ISC DHCP	No	IPV6	[illegible]
ISC Sendmail	[illegible]	Kerberos	[illegible]
ISC Bind/libresolv(3)	No	RAIDFrame	No
GNU CVS / RCS	No	LFS	No
TCPDump	No	pf(4)	[illegible]
GNU GCC, GDB, Binutils	No	Systrace	No
GNU grep, groff, diff	No	IPF	[illegible]
Bzip2	No	ISDN4BSD	No
Postfix	[illegible]	CGD	No
NTP	No	OpenPAM	[illegible]

- **FreeBSD:** could provide for a small environment using the very granular "system packages" offered in a release CD, however reduced size is not one of the primary goals of the FreeBSD project. FreeBSD does however have a comparable number of **make.conf(5)** system conditional build time components.
- **OpenBSD:** Although OpenBSD's greatly enhanced packet filtering engine **pf(4)**, compiler security enhancements, improved **ipsec(4)** API, and a great many other desirable features, OpenBSD lacks a graceful mechanism, other than **$SKIPDIR** in **make.conf(5)**, by which to disable superfluous 3rd party default in-tree packages (Apache, Lynx, Perl, Sudo, etc.). These packages gain significant security enhancements by being maintained in-tree, however they present size concerns.

Pivot_Root() v.s. MD/RD+MFS

Original project goals called for a modified kernel and **rc(8)** that would bootstrap the system from a minimal **md(4)/rd(4)** then transition to a root */(slash)* which would reside entirely in an **mount_mfs(8)** file system thus eliminating the need to use both **md(4)** and **mfs(4)**. However, after researching and experimentation, it was determined that due to the lack of a functional **pivot_root()** sysctl, this is not feasible. Unfortunately, one cannot force-dismount the the *"root_device"* post boot (or technically, post **init(8)** process creation).

Chicken & Egg Scenario: You cannot create and MFS, populate it, and dismount the previously mounted *"root_device"* in one step!:

- Mounting MFS as */(slash)* overlaying *"root_device"* manually with */etc/rc* alienates all userland utilities required to continue the MFS root boostrap (**pax(1)** of *usr.tgz*, **sh(1)** */etc/rc*) by making */bin* and */sbin* unavailable.
- During research, experimenting with overriding sanity checks in *sys/kern/vfs_syscalls.c* in sys_unmount(). It was found that attempting to accomplishing the "one-step" work in C system calls at the end of *sbin/newfs.c* is impossible because you impair **init(8)**'s ability to operate when attempting to force-dismount */(slash)* when you call **umount(2)** because you revoke **init(8)**'s VNODE.

A minimal RD/MD image and a working **pivot_root()** would be ideal because it would require only a few utilities (**mount_mfs(8)**, **fsck(8)**, **pax(1)**, **gzip(1)**, **progress(1)**, etc.). The entire system could reside in MFS and the clutter in **fstab(5)/mount(8)** would be controlled after force-dismounting the original "root_device". Unfortunately, until **pivot_root()** is available[3], the original soekris256 layout remains with */(slash)* residing in RD/MD and */usr* residing in MFS. Very little code will change to adopt from *"usr.tgz" to "system.tgz"*.

Another compelling reason for **pivot_root()** is that there are size limitations to the size of the MD/RD

image that can reside within within an OpenBSD/NetBSD kernel. If the kernel file size exceeds that, the image will not boot. However, there are no such limitations on **mount_mfs(8)**, only available system resources (RAM).

Size Reduction: Prune v.s. Build System

The technique used by original soekris256 scripts called for conserving space by identifying unneeded files to be manually pruned out using **rm(1)**. This approach creates a great deal of manual administrative overhead. Such a list needs need to be manually synchronized with the ongoing release engineering process of the NetBSD system as binaries, libraries, man pages are added and removed. Doing so would involve close scrutiny of CVS commit logs.

A better approach is to slim down the $DESTDIR size using in-tree hooks to remove unneeded components. NetBSD provides this through the **mk.conf(5)** system. Regretfully, even with the existence of all of these mechanisms, some pruning of */etc*, */sbin*, and */bin* must still be done out of necessity. As an example, several utilities in the *"-rnetbsd-3"* release engineering branch exist that aren't applicable to an MFS booting NetBSD appliance that cannot be pruned out of $DESTDIR using **mk.conf(5)** hooks:

- */bin/systrace*
- */bin/csh*
- */sbin/dump_lfs*
- */sbin/vinum*
- */etc/mtree/**
- */etc/X11/**

These 5 paths alone total 1.2mb conserved in the ***/(slash)*** partition in RD/MD (for which only 9.5mb is allocated), which is a significant savings. The decision to include certain modules of the system must be weighed against hardware resources (limitations / budget), booting speed, environment expectations.

Populating $DESTDIR:

The first and preferred option for building a robust system that most-closely resembles a booted NetBSD environment is to utilize the $DESTDIR from *build.sh* release target in combination with **mk.conf(5)** flags and the manual pruning listed above. The other option is to use the tarballs resulting from "make release".

- **The mk.conf(5) + prune method:** Compare "make distribution" target $DESTDIR output **du(1)** sizes for CVS branch "netbsd-3-0" as of datespec "-D 07/25/06 16:21:56 EDT":

mk.conf Flags (Def.)	**MKPOSTFIX=no**	MKPOSTFIX=no **MKSKEY=no**	MKPOSTFIX=no MKSKEY=no **MKUUCP=no**	MKPOSTFIX=no MKSKEY=no MKUUCP=no **MKYP=no** **MKHESOID=no**	MKPOSTFIX=no MKSKEY=no MKUUCP=no MKYP=no MKHESIOD=no **MKNLS=no**	MKPOSTFIX=no MKSKEY=no MKUUCP=no MKYP=no MKHESIOD=no MKNLS=no **USE_INET6=no** **MKINET6=no**	MKPOSTFIX=no MKSKEY=no MKUUCP=no MKYP=no MKHESIOD=no MKNLS=no USE_INET6=no MKINET6=no **MKMANZ=yes**	MKPOSTFIX=no MKSKEY=no MKUUCP=no MKYP=no MKHESIOD=no MKNLS=no USE_INET6=no MKINET6=no **MKSHARE=no**
Size: 203M	194M	196M	194M	193M	187M	186M	163M	108M
Savings	-8432k	+1276k	-1636k	-718k	-6752k	-648K	-23324K	-56534k

- **The $RELDIR + flist Method:** As an alternative or hybrid method, "system packages" (as featured in the Sysinst program) may be utilized to build a $DESTDIR by un-**pax(1)**'ing "feature set" archives from a complete "Release Build". The "base" and "etc" packages would provide a $DESTDIR sufficient for many environments.

System Package Sizes:

System Package	Size (MB) Compressed	Size (MB) Extracted
base	22	64 (60 w/o rescue)
comp	21	77

etc	0.1	1.1
man	7.7	34
text	2.1	7.6
misc	2.8	11
games	2.9	7.3

Symbolic Links

During the *rd_root/* and *cf_root/* directory hierarchy bootstrap/preparation process on the host build, a number of symbolic links are built facing */shadow or ../../shadow* (on the destination host system). However, some complications arise and their work-around are described here:

> **Note:** there is inconsistency in the scripts' use of relative v.s. absolute symbolic links. Some of links are relative to temporary MFS mounts anchored at the $homedir, some are absolute (to /shadow on the build system) which causes pollution to the build system.

Symlinks: init(8), getpwent(3), and etc/rc.d/passwd

A stock $DESTDIR */etc/spwd.db* and */etc/pwd.db* are required to be present when **init(8)** exec's via the kernel. Originally during research, */etc/passwd*, */etc/master.passwd*, */etc/spwd.db*, */etc/pwd.db* would be symbolic links to */shadow*. These links did not resolve until after **rc(8)** started. However, **init(8)** requires the ability to resolve UID 0 to user root otherwise it drops into single-user mode.

As a solution, the stock *spwd.db/pwd.db* are installed into *rd_root/* as real files (no symlinks as expected) and are used at boot by **init(8)**. A custom *etc/rc.d/* script is then needed later to rebuild the system *{,***master***}.***passwd(5)* from the */shadow/etc/{,s}pwd* file system. The script has to be run early in the **rcorder(8)** process to ensure any localized users that are not in the stock password file are available (such as those that system daemon process may reduce user privileges to or run Privilege Separated as – ntpd, apache, named, isakmpd, syslog, etc.)

/etc/rc.d/passwd is a localized script to rebuild the */shadow/etc/master.passwd* by simply calling **pwd_mkdb(8)** script and rebuilding */etc/{,s}pwd.db* (which is not persistent, nor does it need to be). It must be called very early on in the boot sequence because some system service daemon users are included in addition to administrators that do not exist in the stock *$DESDIR/etc/{,s}pwd.db*

Symlinks: etc/rc.d/root and /etc/rc modifications

The original soekris256 scripts modified OpenBSD */etc/rc* slightly to **fsck(8), mount(8)** critical CF file systems prior to executing of */etc/rc*. Further modifications in *rc.patch* provide a manually mounted */cf* and */shadow*, thus resolving symbolic links, specifically */etc/fstab* and */etc/rc.conf*.

This method works in NetBSD, however **rcorder(8)** step five *(/etc/rc.d/root)* attempts to force-**umount(8)** *(/(slash)* after our */etc/rc* modifications have already been run. It makes these assumptions because traditionally the NetBSD kernel would mount the root file system temporarily as "root_device" based on the boot-disk from the 2nd stage boot loader and/or the kernel config file *"config root on ? type ?..."* stanza.

A patch to avoid this behavior is provided in *make_dir_mdrd.sh*. These steps must be worked around but not disabled entirely. */etc/rc.d/mountcritlocal* should honor the contents of our */etc/fstab* with the exception the */, /usr, /shadow, and /cf*, which we've already pre-mounted in **rc(8)**. To avoid issues with services that attempt to start prior to all non-critical file system being mounted (such as **mount_null(8)** type), $critical_filesystems_local is utilized.

Symlinks: MFS /tmp and /var/tmp

/tmp and */var/tmp* need to be consolidated with a symlink. Although the NetBSD system clearly designates them for different uses in several places, they have many historical differences on different platforms, the best administrative approach is to make */tmp* a small MFS mount outside of MFS */usr* and RD */(slash)*.

Symlinks: named(8) & chroot(2) v.s. /shadow, & mount_null(8)

The symlink configuration file strategy conflicts with the desire to run ISC BIND9 in a **chroot(2)** environment (*/var/chroot/named*) where symlinks to */shadow* cannot be resolved outside of the **chroot(2)** directory. A work-around is implemented by mounting */shadow/var/chroot/named/etc* against */var/chroot/named/etc* using **mount_null(8)**.

This approach assumes that you require only */var/chroot/named/etc/* to be available for stateful

configuration (such as query-only DNS slave that transfered zones from a central server master). If you are slaving a significant number of zones from a master server, you may wish to mount your "cache/" or "secondary/" directory as an MFS of appropriate size.

```
# df -h
/shadow/var/chroot/named/etc     465M     376K      441M   0%    /var/chroot/named/etc
/shadow/var/chroot/named/master  465M     376K      441M   0%    /var/chroot/named/master
```

Note: The downside to this approach is name space clutter in **fstab(5)** and **mount(8)** with lots of MFS and NULL file system mounts.

Moreover, the default **rcorder(8)** results cause *rc.d/named* to be started before *rc.d/mountall*, thus to work around that, */var/chroot/named/etc* must be added to $critital_filesystems_local in **rc.conf(5)** so that it is mounted in *rc.d/mountcritlocal* (before NETWORKING).

Populating /usr/pkg with Pkgsrc packages

The **pkg_{add,info,delete}(8)** "-K" and "-p" flags can be used on the build system against *$usr_root/pkg/* and *$rd_root/var/db/pkg* to install packages into the director hierarchy on the build machine that will constitute the target live file system. This is accomplished in *make_dir_cfrd.sh* based on the contents of *pkg_list.txt*

Note: A caveat is that the build environment is almost always required to be running the same branch to be able reliably build binary packages for the target system.

Pkgsrc and Build System mk.conf(5) for Cross-Building

It may be impractical to build NetBSD releases and Pkgsrc packages using an identical environment to the destination system. Your build system will want to be as generic as possible to avoid problems building NetBSD releases.

Moreover, Pkgsrc cross-platform building is generally not conceptually possible even with a cross-target toolchain due to the need to the frequent need of packages to run their own code during the build process. The advantages would be obvious however: compiler optimizations for the native platform (CPU Flags) and guaranteed compatibility due to linking problems running your customized NetBSD release with any optional subcomponents stripped out can ensure.

As an example, *pkgsrc/net/net-snmp* probing INET6 or *pkgsrc/security/sudo* probing Kerberos on the build system.

Pkgsrc does provide an insulation mechanism to avoid problems. Pkgsrc-specific **mk.conf(5)** flags on the build system must be updated to reflect the build flags of the destination system (see above on space conservation) using $PKG_DEFAULT_OPTIONS and $PKG_OPTIONS.${pkgname}=.

Example:

```
# If:
        USE_INET6=no
        MKINET6=no

# ...is set for the system build.sh, then the following
# Pkgsrc options should also be set:

        PKG_DEFAULT_OPTIONS+= -inet6
        PKG_OPTIONS.net-snmp= -inet6
```

To ensure that, although INET6 may be available on the build system, it is not configured into packages built for the destination system. However, this requires the Package maintainer to be aware of GNU Autoconf flags and macros that apply.

Burning the Image

The image is burned with **dd(1)**:

```
# dd if=cf_image of=/dev/rwd0d
bs=2048k &
# pkill -INFO dd
```

Note: Block size may vary depending on your IDE controller's performance and negotiated speed with your CF/IDE adapter. NetBSD requires the use of the "D" slice to ensure the first block offset is the

first block of the physical disk and not the MBR partition. In OpenBSD this would be the "C" slice.

Compact Flash Adapters: wd(4) v.s. umass(4)

Some CF media will detect with entirely different geometries for the same CF media depending on whether it is connected via **umass(4)** style USB->CF adapters instead of EIDE->CF **wd(4)** style adapters, however the latter is not hot swappable and requires a reboot to install.

Moreover, some EIDE->CF adapters have difficulty trying to negotiate UDMA mode. I have noticed this behavior is not limited to any particular IDE controller (including viaide(4) or ICH6 piixide(4)), CF media manufacturer or model, or IDE->CF adapter:

```
wd0 at atabus0 drive 0: <CF500>
wd0: drive supports 1-sector PIO transfers, LBA addressing
wd0: 1968 MB, 7872 cyl, 16 head, 32 sec, 512 bytes/sect x 4030464 sectors
wd0: 32-bit data port
wd0: drive supports PIO mode 4, DMA mode 2, Ultra-DMA mode 2 (Ultra/33)
wd0(viaide0:0:0): using PIO mode 4, Ultra-DMA mode 2 (Ultra/33) (using DMA)
viaide0:0:0: bus-master DMA error: missing interrupt, status=0x21
wd0: transfer error, downgrading to Ultra-DMA mode 1
wd0(viaide0:0:0): using PIO mode 4, Ultra-DMA mode 1 (using DMA)
wd0d: DMA error reading fsbn 0 (wd0 bn 0; cn 0 tn 0 sn 0), retrying
viaide0:0:0: lost interrupt
        type: ata tc_bcount: 512 tc_skip: 0
wd0: transfer error, downgrading to PIO mode 4
```

File System Growth from Periodic Scripts

In light of the previous notes about conserving space, the */(slash)* file system (where */var* and */etc* reside) has a tendency to grow in usage in the first 72 hours of installation due to periodic scripts that perform ***mtree(1)*** related checks and selective */etc/** backups, as well as **makewhatis(8)** and **locate.updatedb(8)**. Due to the nature of our system, we can safely disable these in **security.conf(5)** and **daily.conf(5)** in favor of the system-level IDS service which will be written to R/W CF.

/etc/security.conf:

```
check_mtree=NO
```

/etc/weekly.conf:

```
rebuild_locatedb=NO
```

Network syslog(4), newsyslog.conf(5), and Net-SNMP w/ Transport Mode IPSec

Additional growth of */var* can be controlled by diverting copies **syslog(4)** messages to a remote host while maintaining a very limited cache (256k) locally.

/etc/syslog.conf:

```
local0.*                        /var/log/named.log
local0.*                        @syslog.remote
local1.*                        /var/log/samhain.log
local1.*                        @syslog.remote
```

To maintain only a small syslog cache that does not cause */(slash)* to fill, **newsyslog.conf(5)** must be granularly adjusted:

```
# logfilename           [owner:group]    mode ngen size when  flags
/var/log/authlog                         600  1    64   *     Z
/var/log/maillog                         600  1    64   *     Z
/var/log/messages                        644  1    64   *     Z
/var/log/samhain.log                     644  1    64   *     Z
/var/log/named.log                       644  1    64   *     Z
/var/log/wtmp           root:utmp        664  7    *    168   ZBN
/var/log/wtmpx          root:utmp        664  7    *    168   ZBN
```

Note: This example will rotate the logs at 64K each, ensuring that at any given time, there are only 192k of uncompressed logs.

Additionally, utilization of an agent SNMP is a feature of most network appliances. However, traditionally, SNMP would only ever be run across a "trusted private administration network", even with SNMPv3 "encryption" enabled. Both protocols (Syslog and SNMP) use clear text UDP. Moreover, the appliance in question may be remotely deployed at a data center without a "administration VLAN"

IPSec is recommended. Establishing a Transport mode IPSec SA with a central management server is recommended for transmitting Syslog and SNMP traffic. It is important to install port-level IPSec SA policies that catch any traffic to a management node (which will prevent/block outbound traffic in clear text when the SA is not active/installed).

/etc/ipsec.conf:

```
spdadd 1.2.3.4 snmptrapshost[162] any -P out ipsec esp/transport//require;
spdadd snmptrapshost[162] 1.2.3.4 any -P in ipsec esp/transport//require;
spdadd 1.2.3.4 sysloghost[514] any -P out ipsec esp/transport//require;
spdadd sysloghost[514] 1.2.3.4 any -P in ipsec esp/transport//require;
```

Sendmail aliases:

As part of the boot strapping on the host system during *make_dir_mdrd.sh*, the aliases file in the *shadow_root/etc/mail* is rebuilt. A special *sendmail.cf* must be called with the following:

```
# location of alias file
        O AliasFile=/home/fsimages/shadow_root/etc/mail/aliases
```

Note: Hopefully future versions will have **sendmail(8)** removed in favor of Postfix. This will save space by eliminating the need for M4.

Starting local services from Pkgsrc (/usr/pkg/etc/rc.d/)*

It is important to declare $*rc_order_flags="/usr/pkg/etc/rc.d/**"*in **rc.conf(5)** to ensure that Pkgsrc entries are calculated in the startup **rcorder(8)**. This setting assumes that */usr* is not a separate file system. However, in our case, it is indeed a separate file system, however our */etc/rc* patch properly mounts */usr* prior to the sourcing of */etc/rc.conf* and/or execution of **rcorder(8)**. In other systems, a */etc/rc.d/localpkg* (FreeBSD mechanism), ***rc.local(8)*** stanzas, or forced installation of *pkg/rc.d/* scripts into */etc/rc.d* are required.

gzip(1), progress(1) and usr.tgz issues

Because **gzip(1)** and **compress(1)** are located */usr/bin*, they are not available to new code in */etc/rc* because */usr* does not become available until */cf* is mounted and */cf/usr.tgz* is extracted. During initial testing, *usr.tgz* was actually not a **zlib(3)** compressed archive because of the unavailability of **gzip(1)** to **pax(1)**.

Because of the choice not to prune */usr* down excessively, usr.tgz is approximately [65+] MB:

- The archive can take some time to extract thus delaying boot speeds even to an MFS partition due to not necessarily to the size or the CPU overhead of decompression, but because of the numerous number of smaller sized files.
- It also results in a larger memory footprint for MFS once extracted; -- approximately 190 MB depending on the Pkgsrc packages added.
- Progress bar and optional gzip compression could be added with specially compiling versions of **pax(1)/tar(1)/gzip(1)** that do are:
    - Not statically linked
    - Are dynamically linked against */lib* instead of */usr/lib*
    - Do not depend additional superfluous libraries such as libbz2
- A work-around to compile **progress(1)** correctly:
  By default, **progress(1)** is linked against the libc.so.12 in /usr/lib, which is actually a symlink to the real libc.so file in /lib:

```
# ldd /usr/bin/progress
/usr/bin/progress:
        -lc.12 => /usr/lib/libc.so.12
```

- ```
  # ls -al /usr/lib/libc.so.12
  lrwxr-xr-x  1 root  wheel  21 Dec 18 19:15 /usr/lib/libc.so.12@ -> /lib/libc.so.12.128.2
  ```

To compile against */lib*, two variables must be set in *src/usr.bin/progress/Makefile and src/usr.bin/gzip/Makefile* which adjust *src/share/mk/bsd.prog.mk* and *src/share/mk/bsd.own.mk*[4]:

```
SHLINKDIR=    /libexec
SHLIBDIR=     /lib
```

As well in src/lib/libbz2/Makefile:

```
USE_SHLIBDIR=  yes
SHLINKDIR=    /libexec
SHLIBDIR=     /lib
```

Init(8) with RD/MD FS Root(src/sys/dev/md_root.c)

By default, the kernel will pass the single user boot flag to **init(8)** if the root disk is an **md(4)** file system image. In order to avoid that check, must declare the following kernel options in your MD/RD kernel:

```
options MEMORY_DISK_HOOKS
options MEMORY_DISK_IS_ROOT
options MEMORY_DISK_SERVER=0
options MEMORY_DISK_ROOT_SIZE=21000 # 9 MB, negotiable
options MEMORY_RBFLAGS=0
```

Error Handing in Build Scripts (good2go() function)

The scripts are "batch files" that execute a series of commands in sequence and they lack a great amount of structure and uniformity, however there are a number of notable issues:

- The good2go() function nicely checks $? codes and increments the iteration step counter if called immediately after a step is executed.
- However, some "steps" are actually a "series" of commands that constitute a single step, so:
  - A re-write of the function to use a bitwise boolean that can be called several times by commands in the same step, each of which can flip the bit, and success will be evaluated by the status after the next set of commands are executed.
- Some commands need to possibly call a cleanup function after certain other steps are executed which may leave the build environment dirty (lock files, MFS mounts, **vnd(4)** devices configured, incomplete tar balls, etc.), which they do not yet.
- The original Soekris256 scripts are interactive while my new make_dir* scripts are not, consistency should be addressed.
- The scripts should be consolidated
- *cf.conf* which globally defines all of the variables all four scripts

Outstanding Problem Reports (PRs) Associated with Project:

Send-pr #	*Description*	*Status*
bin/14563	syslogd binds udp sockets on all interfaces	open
pkg/33755	Kernel lacks a VFS/Disk IO Stats API for pkgsrc/net/net-snmp mib module: ucd-snmp/diskio (snmptable diskIOTable)	closed
misc/33758	checkflist.sh fails when MKSHARE=no on -rnetbsd-3 due to flists	open
bin/34733	tcpdump(8) requires default snaplen > 68 for pflog(4)	open

The Scripts

make_dir_mdrd.sh

make_dir_mdrd.sh mimics the series of steps undertaken in

make_dir_cfrd.sh

make_dir_cfrd.sh prepares the file system hierarchy that will
```

the NetBSD sysinst install process. The resulting *rd_root/* directory contains the */(slash)* directory of an installed and configured system minus */usr*.

The steps are:

- **pax(1)** in the contents of **$DESTDIR** minus */usr*
- prune selective contents (necessary evil)
- build */dev/** with MAKEDEV.SH
- make directory anchors (*/usr /cf /shadow /tmp*), create temporary MFS mounts
- make relative symbolic links of localized configs in */etc* to *./shadow/etc* MFS mount
- set the time zone with a sym link (*/usr/share/zoneinfo/**)
- generate SSH RSA/DSA host keys with **ssh-genkey(8)**
- rebuild **aliases(5)** with *"sendmail -bi"*
- patch files in */etc*, install localized additions to */etc/rc.d*
- any other systems-specific changes to */var* or */(slash)*

rd_image.sh

rd_image.sh takes the contents of the *rd_root/* directory and creates a UFS/FFS in-kernel RAMDisk image. using **vnd(4)** which can be called as an argument to **mdsetimage(8)** and installed into a **md(4)** enabled kernel file:

- prompt for the kernel file
- prompt for the image size from MEMORY_DISK_ROOT_SIZE= in the kernel configuration
- prompt for the *rd_root/* path
- **dd(1)** the image file size from */dev/zero* into *image_rd*
- **vnconfig(8)** the image file into */dev/vnd0*
- **disklabel(8)** it with one slice
- **newfs(8)** slice 'a'
- **mount(8)** it
- **pax(1)** in the contents of *rd_root/*
- **umount(8) && vnconfig(8) -u**
- Call **mdsetimage(8)** against the kernel file

reside on slice 'a' of the CF image, as well as assemble the contents of the target MFS

- create a *usr.tgz* from ${DESTDIR}/usr
- create a */usr/pkg* from *pkg_list.txt*
- copy local etc/*rc.d/* scripts from *share/examples/rc.d*
- symlink local configs from *usr/pkg/etc/* to *shadow/usr/pkg/etc*
- **pax(1)** append *usr_root/* to *usr.tgz*
- copy bootblock, mbr, 2nd stage boot to $cf_root
- sync, unmount, vnconfig -u

cf_image.sh

cf_image.sh prepares the two file systems that will actually reside on the CF media. Unlike the original soekris256, this model divides the CF disk into two slices, one reserved for R/O mounts, and one reserved for R/W shadow mounts (See details later).

- prompt for geometry of the CF media
- prompt for the *cf_root/* path
- **dd(1)** the a file image from */dev/zero*
- **vndconfig(8)** it
- Initialize the MBR partition table on */dev/rvnd0* using **fdisk(8)**
- **disklabel(8)** the */dev/rvnd0* into two slices
- **newfs(8)** both
- **installboot(8)** the 1st stage boot loader on slice 'a' of the **vnd(4)** disk image
- **pax(1)** *cf_root/* (*usr.tgz*, *netbsd*, *boot* etc.) into 'a' slice
- **pax(1)** *shadow_root/* into 'b' slice
- **unmount(8)** && **vnconfig(8)** -u

References

1. "Soekris on OpenBSD Running Diskless", http://256.com/gray/docs/soekris_openbsd_diskless/
2. A white paper from Sandisk Corp. addresses some compact flash life expectancy concerns: http://www.sandisk.com/Assets/File/OEM/WhitePapersAndBrochures/RS-MMC/WPaperWearLevelv1.0.pdf
3. Thanks to Ted Unangst, Hannah Schroeter, David Young, Adrian Steinmann for the help on this issue. Adrian Steinmann <ast@marabu.ch> is working on a **pivot_root()** sysctl for FreeBSD that should be easily portable to the other *BSD VFS layers.
4. Thanks to Bill Moran <wmoran@potentialtech.com> with for the help teething through the Makefiles to track down where the extra **ld(1)** "-R" flags were coming from.
5. Those projects should utilize a "halted system" architecture, which is not what this project provides.

The redesign of pkg_install for pkgsrc

Jörg Sonnenberger

October 15, 2006

Pkgsrc is a framework for building third party software on a variety of systems. It is the system of choice on DragonFly and NetBSD.

Pkgsrc was originally derived from FreeBSD ports and many features were added to that foundation. One central component is "pkg_install", a collection of small programs to install and remove packages and other related tasks. While it has been extended over time, the original code base is still mostly present, together with a number of limitations.

During Google's Summer of Code 2006 program this component was rewritten to better fulfill the needs of pkgsrc:

- Integrated archive handling.
- Full specifications of file formats and algorithms.
- Versioned, extensible meta data.
- Better integration of the install framework.

In the paper a comparison of the old approaches, the new solution and the rationale, as well as the state of integration in pkgsrc and of the conversion tools are given.

1 Introduction

The NetBSD Packages Collection or "pkgsrc" is a framework for building third party software. Over the years it was extended to support not only NetBSD, but a great variety of Operating Systems, ranging from Apple's MacOS X to Interix (Microsoft Services for Unix). Beside NetBSD, pkgsrc is the system of choice on DragonFly.

The pkgsrc infrastructure is originally derived from FreeBSD's ports framework. Many features like the wrapper system and buildlink were added over the years. One specific piece is "pkg_install", a collection of small programs to install and remove packages and manage related tasks. While it has been extended over time, the original code base is still mostly present.

Several problems have shown up with different severity, like

- use of external programs for the extraction of packages,
- use of a temporary directory during extraction, followed by moving/copying every file to the real location,
- missing documentation of file formats and precise syntax,
- redundancy of installation/deinstallation scripts,
- advanced updating facilities,
- incoherencies between packages built from source and those installed via binary packages,
- difficult interaction with high-level tools.

The Google's Summer of Code 2006 project provided an opportunity to work on redesigning "pkg_install" to fix most, if not all of the aforementioned problems.

This paper discusses the results in comparison with the older approaches and looks at the state of integration into the pkgsrc system.

2 Package metadata

2.1 Package patterns

The ability to match package names is needed in a number of situations. This includes dependencies and conflicts, but also checks for security vulnerabilities.

In pkgsrc four different pattern types are currently used:

- Plain package names form exact matches.
- Dewey patterns like "gdm>=2.14<2.14.8" consist of the package base name and relation operations. Version numbers are parsed according to a complicated rule set modeled after common practice.

- Fnmatch patterns allow shell-like wildcards ("pear-5.0.[0-9]*") and are most commonly used to match any version of a package.
- Csh-style alternatives ("sun-{jre,jdk}<1.3.1.0.2") are expanded to elementary patterns. If any of those matches, the alternative itself is matching.

All four types have at least one major limitation. Plain matches are actively discouraged, since they can't even deal with local patch versions ("estd-0.5nb1"), making them almost useless.

Csh-style alternatives are needed to handle multiple packages providing common functionality like ghostscript-afpl and ghostscript-gnu.

Dewey patterns are the most expressive pattern, but can't represent a match to all versions. Matching e.g. all sub-versions of 4.3 is a problem as well, since release candidates and patch level complicate the matter. Dewey pattern only work well, when upper or lower bound are precise. The old implementation also had some interesting validation bugs, e.g. "php<5>4" is matched by "php-4".

Fnmatch patterns have the downside of matching more than intended. If there's ever a PHP module which name starts with a digit, the common "php-[0-9]*" pattern for the PHP interpreter itself would match the PHP module as well.

The situation is complicated further as multiple patterns are sometimes used to reduce the number of matching packages. A dependency on PHP 4.x for example introduces at least two patterns: "php-4.4.*" to match the API and "php>=4.4.1nb3" to specify the ABI. The evaluation order by "pkg_add" for missing dependencies is critical. When the second pattern is evaluated first, PHP 5 would be installed and the first pattern would be unsatisfiable as PHP 4 and PHP 5 conflict with each other.

To reduce this mess a way to unify the four styles was needed. One more desirable criterion exists, which wasn't satisfied by the existing rules. "pkg_add" has to choose a package, when more than one package matches a pattern. As long as they have the same base name, a built-in rule is used (see PHP 4/5 earlier) which selects the highest available version. There's no deterministic rule for csh-style alternatives though. User interaction can be used to resolve such conflicts, but they are often either undesirable or unavailable (e.g. automatic package installation during bulk builds). The order should therefore follow explicitly from the pattern.

As most of the patterns in pkgsrc follow the Dewey-style it was useful to keep it as base. The generalized version consists of a package base name and zero or more operator/version pairs. Zero operators provide a full wildcard match and each pair is processed in order as long as they match. This means the

incorrectly parsed pattern "php<5>4" now is valid and behaves as expected. Beside the normal relational operators "<", "textless=" and so, "~" is introduced as prefix match. "php 4.4" matches "php-4.4", but also "php-4.4pl1" and "php-4.4rc1". Finally multiple of this simple patterns can be joined using "|" to form alternatives. Ordering of two matches is done by the first matching alternative first and by ordering the versions themselves if they match the same one.

While the given rules allow easy merging of two basic patterns, it gets more complicated, when alternatives are involved. As this is not typically used in pkgsrc (yet), the problems are left unresolved for now and will be revisited later. A possible solution is to consider a package version as matching only if it matches all requirements.

2.2 Dependencies, conflicts and compatibility

Packages often need other packages to function properly, e.g. because they are dynamically linked against them or call a program from them. In a similar way, some packages can't work when installed at the same file. Historically two packages has to be marked as conflicting, when the package content overlapped, as the "pkg_add' program didn't handle it as failure.

Another use case of patterns are explicit compatibility hints. In pkgsrc the buildlink framework knows two kinds of dependencies – for ABI and API. The latter are the classic way to describe that a certain (minimal) version is needed by a package, e.g. because a new functionality was added in it. ABI dependencies are more complicated though. As dependencies are normally open-ended (all later versions match), it is hard to describe properly when the interface is compatible.

To solve this a package can explicitly specify what it is compatible to. So instead of requiring "libfoo>=1.0", an exact match can be used by packages depending on libfoo. The maintainer of libfoo is now responsible for specifying what the oldest compatible version is. This can be used for ABIs as well as module interfaces in scripting languages like Python. Support for maintaining the compatibility list based on ELF "sonames" or libtool archives is planned.

2.3 Package lists

The heart of a package are the files within. The package list (plist for short) contains all the files in the package, which are supposed to be "static". For each file a checksum is stored and it can be used to detect undesired modifications. The old plist format also contains modifiers to remove directories on removal

and execute single line commands. The functionality to specify permissions or ownership existed, but was never used.

The old plists had three major issues:

- It contains some package metadata, but not all. The ability to execute commands was mentioned already. Another example is that dependencies and conflicts are listed in the plist. The on-line description, the full description, install and deinstall scripts, the package maintainer and all the other information are stored separately though.

  Checksums have been added as afterthought using special comments.

- Commands don't belong into a plist, that's what the install/deinstall scripts are for. Firstly, it increases the number of places to audit and secondly, it also provides a different environment.

- Handling of shared directories is flawed as it is often impossible or very unpractical to factor out a base package to "own" the shared directories. In the past most common directories have been created using mtree from a template and were considered sticky (e.g. never to be removed).

For the new "pkg_install", @exec and @unexec are no longer supported by unanimous consent. All non-plist rated information have been moved and the other statements have been made local to each entry. A field for checksums has been added as well as a field to tag entries to belong to specific classes. The latter allows special scripts to run on the tagged entries e.g. to register a font with fontconfig or add a texinfo page to the local index.

The second important change is the classification of entries. Inspired by the Solaris package tools, other types of plist entries beside simple files are support.

Configuration files are first-class entries. When the file does not exist at install time, it is copied from a template or created as empty file (e.g. for logfiles). On removal, the management tools can either keep it as is, remove it on user request or archive it for later use.

Similar to configuration files, volatile files have a template. They are not archived or even checked for modification, but instead assumed to be modified by the package at vim. This is useful for fixed indices like texindex's info/dir file.

Beside files directories can be contained in the plist as well. As the new "pkg_add" creates them on demand and "pkg_delete" removes them when no other package is referring to them, this is seldom needed. It is needed when empty directories should be part of a package or when special permissions are required.

Two special kinds of directories are also supported. Configuration directories can contain only configuration files and directories as entries and are a way to mark a whole directory hierarchy as containing only configuration files. They are supposed to be handled as whole (e.g. archived). Exclusive directories place a directory under the sole control of a package. No further plist entries are allowed and the system doesn't care about the content. The package is responsible for removing the content at deinstall time. This makes it possible to properly handle e.g. shared-mime-info's share/mime.

Last but not least are symbolic and hard links recorded. The former should not change its target and the latter might be converted down to a symlink if necessary, e.g. when target and plist entry are not on the same filesystem.

2.4 Essential and non-essential metadata

Some of the data attached to a package has been mentioned already – the package name, the list of dependencies and conflicts, the plist. Other items are:

- The prefix a package is installed to and which it is supposed to stay in with some exceptions,
- How to reach the maintainer of the package.
- The OS version and architecture the package was built.
- The license(s) it can be distributed under.
- The short and long descriptions, both in English and local languages.

All this data can be classified as essential or as non-essential. The former category describes what directly affects "pkg_install" and the basic user experience. Having translated descriptions is nice to have, but the English version will always be authoritative and required. Just because a field is essential doesn't mean that it has to be present though. A typical example is the license field which will be missing for most packages, but is critical for determining whether a package can be distributed.

The separation between both classes is useful as it reflects the need of correctly managing and preserving the meaning of a field. As the list of metadata will change in the future, backwards-compatibility will be needed. At the very least it must cover all the essential fields and those have be updated as easily as possible. To achieve this, each field has strong validation rules, which are relaxed for the non-essential metadata.

2.5 Package format

The old "pkg_install" just compressed tar archives containing all files in the plist and normally one file for each of short and long description, the plist, install and deinstall script, size infos. The latter set is also the metadata kept in the package database (typically /var/db/pkg or .pkg in the prefix).

For a typical installation this easily takes a few thousand inodes. To avoid the associated overhead, a format to keep them in one file was needed which doesn't compromise the extensibility. Two generic markup languages were considered, namely XML and YAML. Since white-space handling in XML is awful and YAML is also much human-friendlier, it was preferred by the author.

The serialized package content uses a shallow hierarchy which emphasizes the importance of the various fields. The package itself and the plist entries are explicitly tagged and thereby also versioned. This allows the package tools to easily detect and convert older versions when necessary.

Binary packages are still (compressed) tar archives. The content is different though. In the top level directory, there's an index file containing the serialized package description (as above). This is also required to be first entry of the archive. Signatures will be stored as second entry, but as no light-weight gpg verifying exists and X.509 certificates don't play nicely with the (current) setup of pkgsrc bulk builds, this is not finalized yet. After the index file the normal files from the plist are stored in plist order. The files are stored with the relative path under a directory named like the package. All other plist entries are synthesized during extraction.

Enforcing a strict order on the packages makes it possible to extract a tarball with minimal buffering and read the content without having to process more than the index size (up-rounded to compression blocks). The construct of using a subdirectory for the actual file allows later bundling of multiple packages into a single archive, with minimal changes.

3 The programming interface

The implementation of "pkg_install" consists of a library core and small bindings on top. The core consists of four major components: the pattern related functions, the package-related functions, the plist-related functions and the package database functions.

3.1 Pattern functions

The pattern API provide simple accessor functions for easy access in common situations. Both matching a pattern against a package name and ordering two package names with regard to a pattern are supported. The allocation and freeing of resources are kept internal.

The convenience functions are wrappers for the full implementation. Parsing of a package name or pattern is a separate task to allow later reuse. Functions to extract to the base package name or the list of matched base package names for a pattern are provided. Those are useful e.g. for a bulk build as they can reduce the quadratic runtime in the number of packages and patterns to linearly.

3.2 Package functions

The package functions deal with in-memory package description and related functions. Functions to create one from scratch or destroy it with freeing all associated resources are provided as well as functions to get or set the meta-data. Multi-value fields can be read either using a temporary array or an iterator interface.

The finished package description can be validated either for basic compliance or for the full package conformance. Descriptions which pass the latter can be serialized using a callback interface. In the same way package descriptions can be read back and parsed. A function to create a binary package from a package descriptions and the files relative to given prefix completes the interface.

Errors are classified depending on whether they are input-related or internal. For internal errors like failing memory allocations or violations of the API contract, the program can provide a callback which is called with the current package descriptions, a failure code and optional context-depending arguments. The callback is expected to terminate the application, otherwise it is abort(3)ed. For input-related and other "soft" errors, a different concept is used. The error callback has the same arguments, but can return a value to decide whether or not the processing should continued. This is a ternary value–on error the processing can continue as long as it makes sense to diagnose further problems, but the initial error is sticky. Alternatively the processing will directly bail out. The callbacks are provided on package creation or parsing, it is not yet intended to modify them.

3.3 Plist functions

The plist API allows the addition and removal of individual entries. The interface is strongly typed and each type has independent accessor functions. The implemented makes heavy use of the preprocessor to keep redundancy in code minimal. Similiar to the generic package interface, the plist access is mostly done using iterative callback interfaces.

3.4 Package database functions

The database functions are still in the progress of being revamped. The desired interface has three components:

- Functions to query the database. This should be generic enough to work with package repositories as well.
- Functions to regenerate all internal state like the hash databases of all files and the forest of packages and their relationship.
- Functions to modify the database as set of add/remove operations.

The first category is rudimentary implemented by providing an iterator interface over all packages. The requirement for generalization is important here as the same functions to decide whether a dependency is installed can be used to find the best match in a binary repository. Most query functions should work on binary package repositories as well as the package database.

The second category is implemented, but has to be moved from the standalone command into the library.

The third category is the most challenging. Single add and remove operations work, but impose a severe limitations. Updates of non-leaf packages would have to either remove all depending packages or leave the database temporarily in an inconsistent state. To solve this, complex updates should be done as sets of add and remove operations, which are atomic from the point of the package database.

The downside is that the logic for verifying whether all dependencies are resolved, no conflicts are present and the plists of all to-be-installed packages are non overlapping gets a lot more complicated. As the use of index databases is still necessary for installations with multiple hundred packages, the usage of memory to keep the changes in memory is increasing as well.

It is open whether it is possible and helpful to split such transactions into minimal blocks, which keep the database in a consistent state. It will not help when e.g. xorg-libs changes, but is useful for the generic "update-my-system" case.

4 Integration and conversion

4.1 Staged installation

The first step for the integration of the new "pkg_install" is the elimination of direct installation into the prefix. This makes it much simpler to ensure that all directories created are either requested by the administrator or handled by the framework.

Another important desire is to ensure consistent permissions as many packages don't use the pkgsrc INSTALL_* variables, but random combinations of cp, pax/tar and install.

Therefore the facilities to install into a subdirectory of the working directory were added. As pkgsrc already provided just-in-time su, it was desirable to allow full user package builds. Many packages just use default ownership for files and the aforementioned override directives can be used to provide the functionality even in the old "pkg_install". Some care had to be applied for packages which install setuid/setgid binaries as the access permissions are extracted by tar and the ownership is later changed by "pkg_add", removing the setuid/setgid bits as side effect.

4.2 Pattern conversion

The need to convert old patterns to the new style is an independent effort. Both for the integration and the conversion patterns have to be converted, but it can mostly be done on demand.

As written in section ??, esp. fnmatch patterns are often not precise. A perfect automatic conversion is therefore not possible, but the intent of most patterns can be accurately represented.

The conversion mechanism is based on type-specific rules. Csh-alternative style patterns are expanded, each expanded pattern is converted and the list joined with "|". Simple package names are converted by replacing the last hyphen with "==". Dewey patterns are unchanged as they are a subset of the new grammar. The edge cases are working as humans would expect them, so the change in functionality is justified.

The most difficult case is the conversion of fnmatch patterns. For those a number of heuristics are used. The pattern is matched against regular expressions representing common use in pkgsrc. For example, when "^(.*)-\[0-9]*$" is matched, it means that the patterns applies to any version of the captured first sub-expression. As such it is converted simply to the that sub-expression. Other cases which are handled automatically are "php-4.4.[0-9]*"

and "php-4.4.*", which are converted to "php 4.4¿=4.4". "php-4.4nb*" are "php-4.4nb[0-9]*" are converted to "php 4.4nb".

The given rules can be used to convert all but 30 patterns used by packages in the "pkgsrc-2006Q2" branch and the rest are all somewhat bogus special cases. It is not clear, whether they will end as hard-coded special cases or are left for human intervention.

4.3 The new pkg flavour

In preparation for better support of multiple packaging systems Johnny Lam refactored the package installation and creation code over the last summer. This dramatically simplifies the initial efforts needed for integrating a different "pkg_install" implementation. Using compatibility wrappers for "pkg_info" and "pkg_admin", the changes are concentrated to two places:

- mk/flavour/pkg or a copy thereof
- mk/pkginstall

The former code has to be modified to use the new calling conventions and use individual arguments for each dependency instead of a space-separated list.

The latter code provides the install/deinstall script framework. Most of the functionality has to conditionally tag corresponding items for the new "pkg_install' instead of expanding the shell scripts directly. This will be done incrementally to allow better testing.

As the interface of the package management commands is not finalized, the implementation of this code is still a work-in-progress and not part of the pkgsrc tree.

4.4 Converting existing packages and installations

The creating of package descriptions for testing a new implementation is tiresome and with the implementation of "pkg_create" a shell script for converting existing packages was written. This script has been extended over time to stay in sync with the feature set of "pkg_create".

The biggest missing item right now is the handling of old install scripts. Those fall in one of two categories. Either they are created from the install/deinstall script framework or they are custom rules for a specific package.

The first class is relatively easy to handle as the scripts create individual entries in the package tarball or package database. The metadata can be extracted from the bottom of each file to handle appropriately.

The second class is more involved as there are no fixed marker in the scripts to annotate the beginning or ending of the common fragments (which are already handled). A second problem is that the scripts work both as pre-installation and post-installation scripts and the calling convention has to be emulated. It is an open question how far and in which an automatic conversion can be successful.

5 Conclusion

The redesign of "pkg_install" allowed fixing many of the problems of the old implementations. The installation of packages can be done in-place. The toolchain itself is much more self-contained, typically not requiring external programs, but for additional features. Building blocks for better high-level update mechanisms are provided. The modular architecture will allow further improvements and extensions with minimal redundancy and in a straight-forward fashion.

As side-effect of this work, pkgsrc itself has been improved in a number of ways. During the development of the pattern conversion tools, many bogus dependencies have been fixed. The staged installation has been desired for years and allows catching up with OpenBSD's ports system in that area. Beside the ability to build packages entirely as normal user, it will allow pkgsrc to sub-packages as well.

How to Make Sure That Nobody Will Ever Use My Excellent Software (Twice)

Benedikt Stockebrand

Abstract

One of the greatest successes that a coder can achieve is writing a piece of excellent software and at the same time ensuring that nobody will ever use it—or nobody will ever use it twice.

This talk presents a number of real-world highlights that show how minimum effort can achieve maximum devastation, teaching even the most stubborn user or administrator a lesson it deserves—a lesson to stay away from my software.

Understanding the Enemy

As in any battle the first step to victory is a thorough understanding of the enemy. For our purposes it pays to distinguish four different subspecies: The end user, help desk staff, administrator and management; they all have different vulnerabilities and should therefore be treated specifically.

The End User

The most clueless victim is the end user. It has the least possible grasp of technology and notoriously refuses to read documentation of even error messages, usually either complaining of "techno babble" or pointing out that it doesn't understand English documentation. The end user comes in two varieties: Private and business end user.

The private end user is easy to handle; if we just frustrate it enough it gives up using our software quite quickly.

The business end user is usually told by its management what to do and can't just give up on our software. Dealing with it directly can be fun because it can't really defend itself, but usually we can only teach it to be terrified of our software, but can't really make it stop using it.

The Helpdesker

The business end user is usually supported by a help desk team. The helpdesker usually has a better grasp of the English language, sometimes enabling it to read and understand end user documentation. It is technically marginally clueful and tends to gather quite a bit of experience from dealing with all the problems its end user has.

Like the end user, the helpdesker generally doesn't have much of a choice of the software it has to support. Unlike the end user, the helpdesker is a notorious job hopper, so it is feasible to convince it to hop its job even faster than usual if we write our software accordingly.

The System/Network Administrator

Beyond the help desk hides the system and network administrator. It is at least semi-clued; in some cases it may even have developed basic coding skills. It tends to understand even technical documentation, even if it is written in English.

The admin is usually overworked and has to deal with a wide range of systems. It usually doesn't have time to get into all the details of all the systems it take care of and therefore spends a ridiculous amount of time searching documentation over and over again for details it needs to solve the problem at hand.

A major part of its working hours the admin spends troubleshooting. This gives it an enormous routine to stay cool even in an "exceptional failure" situation—it is just daily business to the admin. A good admin works with great care even in a major emergency situation where every minute costs dearly (but bad admins won't).

At least the senior admin may occasionally be trusted to decide which software to use by its management. So at least in some cases we must consider it a direct target throughout our efforts to prevent it from using our software.

Management

The ultimate target however is management. The Manager isn't even proper business end user by qualification because it has its underlings to do the business end using. But it is the one who is "in charge" and in many cases it will decide for an entire company to stay away from our software.

This is a blessing for us. Since the manager doesn't have the technical competence to realize that our software is excellent, we can achieve our goal without sacrificing any of the excellence of our software.

The only problem with management is that it is hard to reach. But once it gets alarmed it is bound to make some heavy-handed decisions against using our software, no matter what its technical qualities are.

The Software Lifecycle: A Monetary View

The key strategy to deterring the prospective enemy from our software is hitting it where it hurts it most: at its wallet. If we hit hard enough and at the right moment, then we can't but succeed in our mission.

Hitting hard is a matter of the tactical weapon we use; we'll take a look at our armory in the next section. But hitting in the right moment is even more important than the way how we hit. This leads to a brand new sort of software lifecycle, one that focuses on what happens when the software gets shipped to the enemy and how it relates to its money.

On a side note: There is a common strategy pursued by certain "consulting companies" which "do the IT projects" for an IT-illiterate customer. Then run up such a huge bill that the customer's management can't possibly admit to the shareholding public that the "consulting company" delivered an entirely useless heap of junk, so they declare the project a "success" for "political reasons". The result is almost exactly the opposite of what we want to achieve: They make their customers use disastrous software, while our intention is that we want to make the enemy *not* use our *excellent* software. Both strategies do have one thing in common, though: Both will drive the admin crazy and quite likely away from that company.

Evaluation

When a data center intends to install new software, it usually starts with an evaluation of the various alternatives available.

At this point it is generally easy to deter our prospective enemy: Bad documentation and possibly some wanton flaming in the support mailing lists explicitly set up for the new enemy usually do the trick.

At this point it is also surprisingly difficult to deter the prospective enemy in the long run: There is little money and time involved yet, so the enemy may decide to give our software another try later on.

Installation and Configuration

At the end of the evaluation phase the enemy chooses a software. During the following installation and configuration phase we can apply a variety of attacks.

But generally this is not the best time for an attack since there is still too little money involved. Instead we should keep a low profile, try to appear helpful and give the enemy the impression that there won't be any problems with our software. The installation and configuration phase is the last chance for the enemy to retreat without major losses.

Only if the installation and configuration phase is managed as part of a software project with the customarily infeasible deadlines, then some stealthy tactics to delay the installation and configuration may be worth a thought.

Production

After the initial installation and configuration phase the enemy finally places itself at our mercy.

Sometimes it realizes this and tries to negotiate its surrender through a "pilot phase" that it doesn't consider regular operations. Often the administrator is particularly vigilant when then it puts a system into production. But hurting the enemy is primarily about hitting its wallet, so at this point the enemy can't possibly escape our wrath any longer.

There are three basic attack lines relevant to production systems: We can make regular operations excessively expensive, create havoc during upgrades including security patches, and lay some fatal traps that only bite the enemy when it already struggles with another problem.

Regular Operations

To attack the enemy during regular operations we must make operations as expensive as possible. If we make the software difficult to use, then the end user is the first to give up, causing excessive workload to the helpdesker which needs additional manpower which costs the management additional money.

Similarly, we can make administration more expensive than necessary. Any software that makes daily operation a tedious routine job or requires extensive knowledge and experience in widely different and unrelated areas will be considered an excessive expense by our involuntary accomplice in management, which calls itself "controlling". While this may sound fairly unexciting to a software developer, the sums involved can be large enough to

drive the enemy manager to despair and therefore away from our software.

Another useful strategy related to regular operations are excessive requirements for service downtimes. While these can be scheduled in advance, they still put a burden on many installations. Again, the financial losses during downtimes can be substantial enough to convince th enemy manager to convert to an inferior software.

Finally, problem hiding is a way to drive the enemy to despair: If it never really knows if everything works as expected or if some problem is quietly creating some yet unseen havoc, then it will quickly decide that it doesn't want to use our software anymore: The admin is permanently stressed from worrying if everything works and the manager considers the software as well as the admin unreliable, which is reason enough for either one to look for an alternative to our software.

Upgrades

The enemy is even more vulnerable during upgrades. Of course it knows about this and avoids upgrades whenever possible. But if growing systems, increasing reliability requirements, exciting new features, support for new hardware or ending support for older software versions don't convince the enemy to face the upgrade, then an exploitable security hole will.

The only defense it has is extensive preproduction testing. A reasonably experienced enemy will avoid haphazard upgrades and rather spend time and money on preliminary tests. But duplicating an entire datacenter environment doubles the expenses, so in many cases the enemy only has a limited test environment used for multiple software installations in turn. So a pre-upgrade test works like this: Somebody decides that an upgrade is necessary, then a time slot for the test environment is allocated, a large number of tests are done in short time and afterwards the enemy will still worry if it missed the one crucial thing to test that'll blow up in its face. If we don't leave it a chance to revert to the old version this situation will make the enemy grow old way before its time.

With desktop machines the situation is different: A test environment doesn't need as many desktop machines as there are desktop machines in use, which saves some money on the test hardware. But rolling out an upgrade to several thousand machines running a large variety of applications is quite similarly scary.

The obvious strategy of throwing a monthly batch of updates plus the occasional "extra urgent" security fix at the enemy is well-known and has proven useful over and again. It does however require an unacceptably low level of software excellence to be applicable, so it is less useful to deter the enemy from using our excellent software.

Still, there are other means to send the enemy to the upgrade hell; we'll see examples below.

System Failure

Finally, there is the ultimate victory scenario, the one scenario that strikes the very heart of the enemy.

The new junior admin, which was just hired three weeks ago, is on weekend call duty for the first time. On Sunday morning, 03:00, its mobile phone rings. The admin answers, only to hear "machine XYZ doesn't work, come here and fix it immediately". Of course, it has never ever heard of machine XYZ, but it drives to the data only to be welcomed by an unscheduled meeting of the management board with the words: "Do you actually realize that every minute of this costs us 50 000 €?"

Of course, the very same scenario applies to senior admins in large enterprises if the 50 000 € are substituted by a more adequate number. The sky is the limit here: Occasional rumors claim that the Deutsche Bank will be bankrupt within 24 hours if their entire IT breaks down. For the recent two-hour failure of the Spanish top level DNS domain no numbers seem to be available, but at a national economic scale the losses caused by this might just possibly exceed 50 000 €/minute, or 3 000 000 €/hour—by a few orders of magnitude.

This scenario is so valuable for a variety of reasons. Of course, the immediate economic impact is obvious: we hurt the enemy where it hurts most—at its wallet. Besides that, we score multiple severe psychological hits: The end user gets upset because it can't work. The helpdesker gets upset because it has to answer the phone calls of the end user while it doesn't really know what's going on. The admin feels seriously embarrassed because it appears incapable of keeping its systems up and running. The manager feels helpless because it has no idea what's going on or how long it will take to fix, but a reasonably precise idea of the huge losses per minute.

Even in situations far less dramatic than those mentioned it is quite simple to make our software so expensive to use that the enemy will eventually give up and leave our software alone; we just need to make system failures expensive. We can make failures happen often, and we can make them last long. Beyond that we can make it impossible to repair the system completely by introducing inconsistencies during the failure; this will make it necessary to roll back to the last working backup, losing all data changes since then. All three approaches can be combined and put enough economic pressure on the enemy to convince it quickly to use some other, less excellent software than ours.

If we want to keep the level of excellence of our software, we can't resort to causing the problem within our software; instead we must ensure that the "cause" of the problem lies outside, usually either within the system environment or the user or admin operating it.

Masterpieces of Deterrence

Now that we understand the enemy and its vulnerabilities, we can understand and assess the various tactics available to us. After so much theory we now follow a more practical approach and take a closer look at successful examples of enemy deterrence.

Documentation

Documentation-based deterrence measures are quite common, often because documentation isn't really considered part of the software and thus exempt from the goal of excellence. But even if we consider documentation part of the software, there are some excellent tactics available.

The most obvious tactic related to documentation is used by various low-budget hardware vendors: Write the documentation in whatever language we are unfamiliar with, then run it through babelfish to translate into Albanian, and then have a native Chinese speaker translate the result into English. As long as we don't consider documentation part of our software, this approach is generally known to work as expected. Unfortunately, the enemy will quite likely notice this during evaluation, so the impact is very limited.

A more effective, diametrally different approach involves a native English speaker linguist specializing in classic English literature polishing the documentation to unsurpassed beauty: While the language of Shakespeare, Melville and Thoreau may be most elegant and stylish, it is impossible for a non-native speaker with a more limited grasp of English to understand any of this.

The Debian[1] project came up with a nice way to deal with missing documentation: There is an `undocumented(7)` man page that the package maintainers generously use as a substitute for non-existing man pages. The subtle psychological effect of this is quite remarkable: The enemy will almost invariably interpret this helpful note as "I know I let you down; sue me."

Beyond that, Debian Woody made generous use of the Linux-specific `ip` command to configure its network, rather than the `ifconfig`, `route`, `arp` and various other commands commonly used with Unix. The documentation available came in three variants: LaTeX source code with more than 80 chars per line to make it less readable on a text console, DVI intermediate output and PostScript. The subtlety of this is brilliant: During the evaluation, installation and regular operation it is quite likely that the enemy will simply use this documentation. Only when a problem occurs that drops the enemy into text mode only will it realize that it can't any longer read the documentation it desperately needs.

FreeBSD 6.1 installs with a file `IMPLEMENTATION` in `/usr/share/doc/IPv6` which states right at the beginning that it doesn't relate to the KAME IPv6 stack integrated with NetBSD 1.5.1, but might still be useful. Following that the table of contents has an entry "7.2 Multipath Routing Support". Except of course that chapter 7 covers coding style and there is no section 7.2. So if the enemy doesn't bother to test for multipath routing support during the evaluation period but relies on the assumption that it can use multipath routing later on because it was mentioned in the table of contents, then the enemy will stumble over this only when it tries to use multipath routing when the system is hopefully already in production.

The man page for `dig` with FreeBSD 6.1 shows a date of June 30, 2000. During the last six years a variety of changes to the `dig` command have found their way into the source code. It takes a very close look to realize that only the date hasn't been updated in the man page but everything else has. In a high stress situation like troubleshooting this tiny little lapse will easily extend the downtime by several minutes; the average enemy admin tends to be overly careful when working in production machines and won't really trust this man page until it has verified that its contents is actually up to date.

The man page for `cvs` starts with this note: "This man page is a summary of some of the features of `cvs` but it may no longer be kept up-to-date. For more current and in-depth documentation, please consult the Cederqvist manual (via the `info cvs` command or otherwise, as described in the SEE ALSO section of this man page)." Again this is a gem: During evaluation and installation the introductory style, texinfo based manual is generally preferable over a man page. But when a problem occurs, then a more concise, reference-style man page is needed, not a lengthy tutorial-style info file. And we have shown all the goodwill the enemy could ask for: We have provided a reference man page as well—if it assumes that the man page is unreliable, well, that's its own decision.

Support

Similar to documentation, support is often not considered part of the software proper and therefore lends itself well to various tactics. Some of them

[1] OK, "Debian GNU/Linux", if you insist.

are more commonly seen with commercial software vendors, but they are still inspiring enough to be mentioned.

It is common to discontinue support for old software versions as soon as possible. Together with promises of new features, this is a common way to lure the enemy into upgrade hell and no particular surprise. Several years ago SUN support vastly improved this tactic: Whenever the enemy tried to open a problem call they would first demand that it updated the system with the latest "recommended patches". With this strategy they managed to force the enemy to deal with a system failure while in upgrade hell at the same time. Unfortunately, the enemy eventually realized developed a counter-strategy of demanding a written guarantee that the new, untested "recommended patches" wouldn't affect the system adversely but actually helped to solve the problem at hand.

Open source projects offer another line of tactics: Since support usually isn't paid for, it is easily possible to repel the enemy by simply telling the truth: "If you can't read the source and figure it out yourself, then get lost. I can't be bothered to write documentation and I definitely can't be bothered to tell anyone how to use my excellent software." If we do this only after the enemy has reached the problem-solving state, then a few well-chosen insults will quickly deter it from using our software once and forever.

Wanton Limitations

Another set of tactics relate to imposing wanton limitations on the software or the system it runs on.

Limitations that relate to the software itself are usually a sign either of bad software or unscrupulous money-making: The various size limits for IDE hard disks imposed on various generations of BIOSes are all signs of bad software design; selling different kinds of system phones to different PBX systems has been used by phone manufacturers to force the enemy to replace not only the PBX but also all the phones in a company as soon as the company grew beyond the capacity of the old PBX. Either way, while these tactics are proven to be effective they can't be applied to excellent software too easily.

A more useful and less conspicuous way to impose artificial limitations involves interference with other software. Back in the Good Old Days[TM] it was impossible to install both what later became FreeBSD/NetBSD and DOS on the same hard disk, simply because BSD didn't support PC-style partition tables. Since this will usually be noticed during evaluation, the impact on the enemy isn't too exciting, but it quite effectively scares it away. Microsoft has refined this to the "Windows DLL hell": If different programs need different versions of a certain dynamically linked library (DLL, the Windows equivalent of a shared library), then they can't run on the same machine.

Excessive Dependencies and the Autoconf Trick

Even more useful and less conspicuous are excessive dependencies.

Solaris 10 offers "zones", the Solaris equivalent of FreeBSD jails (but with IPv6 support). These zones can't be used without installing resource pools, which need a Java runtime, which need X11 even on a server with a serial console and no video hardware.

Again, this can be improved. The autoconf-generated configure scripts commonly used by open source projects can be easily used to create excessive dependencies that we can blame the enemy for: If it finds KDE, why not use it? And GNOME too, and a Java runtime and SSL and SNMP and an OpenOffice programming interface. Now if the enemy builds our software on its desktop machine, then it has no choice but to install all these things on the final destination machine, too. We can't possibly be blamed because our software doesn't really "depend" on these dependencies, it just makes use of them if they are available anyway. And next time the enemy builds our software again, chances are that it has installed yet some more software on its desktop machine and if it installs the newly built software on the destination machine it will fail because the enemy hasn't installed the additional software there, yet. Beautiful.

All these tricks also open the path for additional fun with respect to security: If our software uses some insecure dependency, like the intrinsically insecure SNMP, or anything with a less-than-impressive security history, like a certain commercial web browser, then we can use these dependencies to force the enemy into upgrade hell more often than it can handle.

Configurability

To make software excellent we must make it configurable to the enemies needs. So if we want to preserve the excellence of our software, we can't use configurability as a weapon against the enemy—or can we?

The Asterisk IP telephony software offers a highly configurable "dialplan" configuration which defines the behaviour of the software. Its lines must be numbered like ancient BASIC programs. Unlike BASIC programs the line numbers must be consecutive. Errors cause a jump to the current line

number plus 101, so if an error in line 7 occurs, line 108 will be executed next. A local Asterisk expert summarized this in this way: "If you want to write a dialplan, then do it in a single day. If you don't, then on the second day you won't understand what you have done the day before and start from scratch." This makes any configuration change later on a nightmare to the enemy; the first time a change is necessary, it takes all day and breaks some previously functional features will quite likely encourage the enemy to look for an alternative solution.

Another example is the traditional `sendmail.cf` to configure Sendmail. Besides the syntax, which is annoyingly difficult to understand, a Sendmail configuration has to be fairly large. Writing a `sendmail.cf` from scratch is quite demanding and requires detailed understanding of the SMTP protocol to keep the result standards-compliant. Unfortunately somebody made the huge mistake to write an m4 macro package that generates a `sendmail.cf` from a fairly short and readable configuration file.

The traditional `sendmail.cf` offered an excessive degree of configurability that virtually nobody needs anymore today. DECNET, BITNET and UUCP are effectively dead, but still Sendmail has everything necessary to support them. Again, if it wasn't for the m4 macros this would serve quite well to make the enemy switch to Microsoft Exchange without looking back.

Even with just a few configurable parameters it is possible to make configuration tedious and error-prone simply by using bad default settings. Solaris uses a default prefix length ("netmask") setting of /128 for IPv6 addresses even though RFC 4291 and its predecessors explicitly state that the prefix length for all but a few special address ranges is always /64. Configurations which appear to be correct, but aren't, can easily confuse the enemy for some time.

The Solaris installer creates an `/etc/hosts` file which assigns the name `loghost` to the IPv4 loopback address. In an environment that uses a central log host and a name server, this leads to an inconsistent configuration until the enemy fixes either the resolver configuration or the `/etc/hosts` file. If it has also enabled IPv6 support on the system, then the same procedure repeats with `/etc/inet/ipnodes` for the IPv6 loopback address.

Another tactic is generously applied by most Unixen: The hostname is often spread all over `/etc`, making it a tedious job to just rename a single machine.

Configuring a Fedora Core 2 box as an IPv6 router shows how to use an inconsistent configuration syntax to confuse the enemy: In `/etc/sysconfig/network` the lines

```
NETWORKING_IPV6=yes
IPV6FORWARDING=yes
IPV6_ROUTER=yes
```

show how mixing underscore and no-underscore notation and prefix and postfix category naming can be easily applied to confuse the enemy admin. Of course, if the file was read by a proper parser, then it could flag parsing errors if the keywords were misspelled. But this file is a shell script, so if one of the variables is misspelled there won't be any visible problems—except that the router doesn't work as expected.

Finally, Solaris 10 introduced the "service management facility" (SMF) as a substitute for `init` and the SysV-style init scripts. It speeds up the boot process, deals with dependencies between services and is far nicer to handle than traditional init scripts—until the enemy needs to change the settings for a service: Then it will face a configuration "data base" that keeps binary representations of lengthy XML-based "service manifests". Editing an XML file with `vi` is great fun to watch the enemy do, especially if it is sitting at a VT100 terminal trying to solve a problem that just brought a system down.

The beauty of all these tactics is that they appear nothing more than a slight awkwardness. If the enemy loses time because of them, possibly at 50 000 € per minute, then that isn't our fault, really. And it *will* lose time because of them. Lots of time.

Change handling and Upgrades

All tactics so far deal with a "static" software that doesn't change over time. In practice, virtually all software is continuously changed, updated and extended. We can use this to hit the enemy when it hurts a lot: during upgrades.

Again we can learn from various hardware vendors who sell different components under the same name. If you have ever seen the enemy trying to replace a broken network card with a new card of "the same model" you know the beauty of this plot: Of course the new card doesn't work, so the enemy assumes that something else is the problem. After replacing virtually everything else in the machine it might finally "re-install the driver" from the CD-ROM shipped with the new card and surprise, afterwards the machine works again—until the next major upgrade is mass-deployed to all machines and this one with the "same model" network card loses its network connectivity in the process. The loss of service may sum up to several days in the long run.

The same can be done with software. The beauty of the BSD ports collections is that they usually obtain the source files from the original

source of the software. If a source file isater on updated there, then the ports collection makefiles will reject that source file as "broken" since it doesn't match the stored checksum. To rebuild a working system the enemy will first have to upgrade its ports tree and then recompile all and everything. If the timing is right, there will be new releases of KDE, GNOME, OpenOffice and a variety of other large packages, causing all sorts of minor problems that are tedious and therefore expensive to fix.

The BIND name server package did a very sneaky trick with the upgrade from 8.2.2 to 8.2.3: They called 8.2.3 a "maintenance release" that didn't change any functionality but just contained bugfixes. Which was perfectly true, except that they changed the way `dig` handled the `-x` option with IPv6 addresses: Instead of looking up the address in the (by then deprecated) bitlabel format, they changed it to use the (by then re-established) nibble format. The results to various shell scripts using `dig` to query the DNS could have been most satisfying; unfortunately it was way too early for pulling this trick, so only a single IPv6 advocate writing a book on the topic was seriously affected.

In a project that I was involved with some years ago, the closed-source software vendor forced the enemy to upgrade to a new release. The software was meant as a micro-billing system but the enemy used it as a means to gather statistical data only. The upgrade involved various reorganizations of the data bases which the vendor claimed not to require any additional disk space. The script they supplied to the enemy to do the reorganization was expected to run two weeks, during which the system couldn't gather any additional billing data. Most unfortunately, the enemy employed a single better-than-average application admin which managed to first split the script into about twenty separate steps. Taking data base dumps after each step it was able to revert to the results of the previous step after the occasional failure. In one case it temporarily doubled the disk space by acquiring a second external storage array before continuing. The entire upgrade took almost five weeks; afterwards said admin left the project. If the enemy had actually used this system for its billing, then this plot would have sent it straight into bankruptcy.

As these examples show, it is both feasible as well as worthwhile to drive the enemy into upgrade hell. Watching it squirm, trying to delay the upgrade while it fully realizes that it can't possibly escape it in the long run, is one of the most satisfying experiences in every software developers career.

Security Aspects

Of course, excellent software is also secure. The Sony "no other notebook brands are affected" line concerning their exploding batteries is about as inapplicable as the as Microsoft's "our primary goal now is to improve security" while they still ship their Internet Explorer with ActiveX support and Office with VBA. So how can security aspects help to deter the enemy?

There is a technical tactic that is applicable: We can add user-configurable support for inherently insecure features, like support for SNMP "set" operations; this leads back to the configurability topic.

As a psychological trick we can simply deny all alleged security problems in a manner that shows the enemy that we feel personally offended by its allegations and no, there is no security issue at all. Then if there ever *is* an issue, then the enemy will deeply distrust the security of our software.

Incompatibilities

Most of todays datacenter environments are highly heterogeneous. Incompatibilities, ranging from straight non-interoperability down to slight variations between different Unixen make life more difficult to the enemy while there is no single Unix to blame.

Several years ago I witnessed the enemy buying three PCI cards of different types to be put in a single machine. Any two cards worked together flawlessly, but all three together rendered all available test machines unbootable. All three card vendors blamed the others on the problem and refused the enemy a refund for their respective card.

Back in 1998 a Solaris NFS server and a Linux NFS client would manage to transmit approximately 100 kB/s between each other. Solaris-Solaris or Linux-Linux setups would easily do 1–3 MB/s. Unfortunately the enemy found ways to tune the Linux client in a way that raised the throughput quite significantly, but still the performance was less than satisfying.

Simply using different option letters for the same command makes scripting difficult to the enemy and can wreak the occasional havoc that the enemy admin fears: The `netstat` command with both Solaris and the BSDs uses an option `-f inet6` to request IPv6-related information; with Linux it is `-A inet6`. Various Solaris commands like `df` use the `-h` option not to display a short command synopsis but to render their output in a "human readable" style.

So far these incompatibilities are mere nuisances to the enemy. But a Solaris box in a mostly BSD-derived environment will eventually embarrass the enemy admin with a gem that looks basically like

```
# /usr/sbin/shutdown -g 10 -i 5 -y \
    Shutting down in ten minutes
System going down in 10 seconds
```

The ultimate incompatibility example is the `killall` command. With Linux and the BSDs it kills all processes with a given program name but with Solaris and other true SVR4 Unixen it will do what its name suggests—it will kill *all* processes, shutting down the system the hard way. Linux has a command `killall5` which behaves like the SVR4 `killall` and Solaris has a command `pkill` which behaves similar to the Linux/BSD `killall` command. Eventually the enemy will learn to be very careful about the `killall` command; usually it will learn the hard way.

Error Handling

When an error occurs within our software, be it an internal problem or bad input or a problem with the environment it runs in, like a full file system, then we can apply some more tactics.

If we didn't care about the excellence of our software, then we could simply send a cryptic error message of the "General Protection Fault" style to the enemy and then ask it to decide if it wants to "Abort, Retry or Continue?" Especially if we don't define a default decision for the enemy it will feel completely lost.

We can also write all logging entries into a single file, mixing debugging information, notices and more or less serious error conditions into a single file. If we don't include time stamps and use multi-line messages that aren't even separated by newlines, then the enemy will have great trouble to monitor this log file. If we use standard block-buffered I/O for this file "for performance reasons" and "forget" to open the file in append-only mode, then the enemy can't add periodic timestamps to the file, the most recent entries are either missing or incomplete due to buffering and if the writing process crashes, then the most vital last entry is effectively unavailable for cleaning up the mess. The billing software mentioned above has proved that this approach is both applicable and effective; unfortunately the effects on the excellence of our software usually forbid this tactic. Excellent software uses the `syslog(3)` API and even sets the priorities of each log message to a reasonable value.

Similarly, using the `assert()` macro or simply dumping a Java backtrace is generally considered bad style. The `rsync(1)` command tries to flood the enemy with information it can't use; a full file system at the destination first provides an "XXX write failed, filesystem is full" message and then follows up with seven additional lines that the average enemy can't understand, including the source file names and line numbers where the related secondary errors occurred.

My personal favourite in this category is Debian and how it deals with the IPv6 configuration in `/etc/network/interfaces`. As mentioned above the prefix length is effectively always 64 bits. So a configuration like

```
iface eth0 inet6 static
        address 2001:db8:fedc::1
```

provides all the information necessary to configure interface `eth0` for IPv6. Still, the `ifup` command will complain that

```
Don't seem to be have all the
variables for eth0/inet6.
Failed to bring up eth0.
```

Only after adding a line `netmask 64` to the configuration will the interface configure. The artful combination of a grammatical mistake in the error message, explicitly demanding a constant to be configured, using an IPv4 term "netmask" for an IPv6 prefix length and finally denying any hint at the problem in the error message will make it absolutely plain to the enemy that it isn't welcome to use this software.

High Risk User Interfaces

Finally, the ultimate weapon against the most stubborn enemy is a high risk user interface. This is the software equivalent of a gun without a safety catch.

This tactic is commonly cloaked by asking the enemy for explicit confirmation for even the most simple operations: "Do you really move this file to trash?" asked for every single file out of five hundred will quickly teach the enemy to confirm whatever the system asks. Not only will this result in the same net effect as not asking for any confirmation at all, it will also annoy the enemy and, most importantly, it will make it confirm whatever really dangerous operation it accidentially invokes. Exposing the enemy to Microsoft Windows will quickly make it reach this "whatever it asks, just hit Return" mentality.

Solaris 10, 06/06, first made the excellent zetta file system (ZFS) available to the public. ZFS manages multiple file systems within a storage pool. To remove such a file system from a pool, the command

```
# zfs destroy ⟨file system⟩
```

will destroy a file system without further confirmation. But there's more to it: ZFS supports snapshots, which are named ⟨file system⟩`@`⟨snapshot⟩. To remove a snapshot, the command is

```
# zfs destroy ⟨file system⟩@⟨snapshot⟩
```

and again it doesn't ask for confirmation. It is only a matter of time until the enemy wants to release a snapshot but accidentially nukes an entire file system. This example more than compensates the lack

of even the remotest hint of subtlety with the enormous degree of devastation it can cause.

The beauty of high risk user interfaces is obvious: Whatever happens, it is the enemies fault, not ours or that of our software.

Summary

We have seen that there is a wide choice of low-effort, devastating-impact tactics to discourage even the most stubborn enemy from using our excellent software. Most of them can be made to appear "accidential" or "slightly awkward" rather than intentional and malicious.

With these weapons available and properly understood we can easily teach the enemy never to use our excellent software (twice).

About the Author

Benedikt Stockebrand is a BSD-biased "generic Unixer" with a strong background in system administration and large-scale data center design and operation. He is working as a freelance trainer, author, IT journalist and consultant with a current focus on IPv6 operations.

He has been repeatedly charged with offensive sarcasm but so far escaped conviction.

How the FreeBSD Project Works

Robert N. M. Watson
rwatson@FreeBSD.org

FreeBSD Project

Computer Laboratory
University of Cambridge

1 Introduction

FreeBSD is a widely deployed open source operating system. [3] Found throughout the industry, FreeBSD is the operating system of choice for many appliance products, embedded devices, as a foundation OS for several mainstream commercial operating systems, and as a basis for academic research. This is distinct, however, from the FreeBSD Project, which is a community of open source developers and users. This paper discusses the structure of the FreeBSD Project as an organization that produces, maintains, supports, and uses the FreeBSD Operating System. As this community is extremely large, I approach this from the perspective of a FreeBSD developer. This necessarily captures the project from my perspective, but having had the opportunity to discuss the FreeBSD Project extensively with many people inside and outside the community, I hope it is also more generally applicable.

2 Introduction to FreeBSD

FreeBSD is an open source BSD UNIX operating system, consisting of a kernel, user space environment, extensive documentation, and a large number of bundled third party applications. It is widely used as an ISP server platform, including at well-known providers such as Yahoo!, Verio, New York Internet, ISC, Demon, and Pair. It is also widely used in part or in whole for appliances and embedded devices, including Juniper's JunOS, Nokia's IPSO, and for commercial operating system products, such as VXWorks and Mac OS X. The product of one of the most successful open source projects in the world, FreeBSD development work has focused on the areas of storage, networking, security, scalability, hardware support, and application portability.

The highly active FreeBSD development community centers on services offered via FreeBSD.org, which include four CVS repositories and a Perforce repository. These represent the life-blood of the development and documentation work of the Project. There are over 300 active developers working in CVS, which hosts the official development trees for the base source code, Ports Collection, projects tree, and documentation project. Significant project work also takes place in Perforce, which supports a heavily branched concurrent development model as well as guest accounts and external projects.

Another defining feature of the FreeBSD Project is its use of the liberal Berkeley open source license. Among features of the license are is remarkable simplicity (the license can be fully displayed in an 80x24 terminal window) and its ability to support derived works that are closed source, key to commercial and research adoption of FreeBSD.

3 What do you get with FreeBSD?

FreeBSD is a complete, integrated UNIX system. The core of FreeBSD is a portable multi-processing, multi-threaded kernel able to run on a variety of hardware platforms including Intel/AMD 32-bit and 64-bit processors, Intel's Itanium platform, and Sun's UltraSparc platform. FreeBSD is also able to run on several embedded platforms based on i386, ARM, and PowerPC; a MIPS port is also underway.

FreeBSD implements a variety of application programming interfaces (APIs) including the POSIX and Berkeley Sockets APIs, as well as providing a full UNIX command line and scripting environment. The FreeBSD network stack supports IPv4, IPv6, IPX/SPX, EtherTalk, IPSEC, ATM, Bluetooth, 802.11, with forthcoming support for SCTP. Security features include access control lists (ACLs), mandatory access control (MAC), security event auditing, pluggable authentication modules (PAM), and a variety of cryptographic services. FreeBSD ships with both workstation/server and embedded development targets, and comes with extensive user and programmer documentation.

FreeBSD also ships with ports of over 16,000 third party open- and closed-source software packages, providing programming and user interfaces such as X11, KDE, Gnome, OpenOffice, and server software such as Java, MySQL, PostgreSQL, and Apache.

4 The FreeBSD Project

The FreeBSD Project's success can be measured by the extremely wide deployment of FreeBSD-based systems. From root name servers to major web hosts, search engines, and routing infrastructure, FreeBSD may be found at most major service providers. FreeBSD is also the foundation for a number of commercial operating systems. The FreeBSD Project is more than just software, or even software development: it includes a global community of developers, port maintainers, advocates, and an extensive user community. Central to this community are the FreeBSD.org web site, FTP site, CVS repository, and mailing lists.

Several papers and studies have been written on the topic of the FreeBSD Project and its development process, including a papers by Richards [7], Jorgensen [4], and Dinh-Trong [1].

5 The FreeBSD Foundation

The FreeBSD Foundation is a non-profit organization based in Boulder, CO. By design, the Foundation is separate from the FreeBSD Project. When the Foundation was created, it was not clear that a non-profit supporting open source development was a viable concept. As such, it was important to the founders that the Foundation be a separate legal entity that would support the Project, but that the Project not be dependent on the long-term viability of a Foundation. It was also important to the founders of the Foundation that there be a differentiation between the people managing the monetary, legal, and administrative matters and those administering the software development work in the project. In practice, the Foundation has proved financially and administratively successful, and plays an important role in supporting the daily operation and long term success of the Project.

The FreeBSD Foundation is responsible for a broad range of activities including contract development (especially relating to Java), managing of intellectual property, acting as a legal entity for contractual agreements (including non-disclosure agreements, software licensing, etc), providing legal support for licensing and intellectual property issues, fund-raising, event sponsorship (including BSDCan, EuroBSDCon, AsiaBSDCon, and several FreeBSD developer summits a year), providing travel support for FreeBSD developers and advocates, negotiating collaborative R&D agreements, and more.

The FreeBSD Foundation is currently managed by a board of directors, and has one part-time employee who is responsible for day-to-day operation of the Foundation as well as sitting on the board. The board also consists of four volunteer members drawn from the FreeBSD developer community. The FreeBSD Foundation Board is in regular communication with other administrative bodies in the FreeBSD Project, including the FreeBSD Core Team.

The FreeBSD Foundation is entirely supported by donations, and needs your help to continue its work!

6 What We Produce and Consume

The FreeBSD Project produces a great deal of code: the FreeBSD kernel, user space, and the Ports Collection. But the FreeBSD Project does not produce "just source code". FreeBSD is a complete software product, consisting of software, distribution, documentation, and support:

- FreeBSD kernel, user space
- Ports collection, binary package builds
- FreeBSD releases
- FreeBSD manual pages, handbook, web pages, marketing material
- Architecture and engineering designs, papers, reports, etc
- Technical support, including answering questions and debugging problems
- Involvement in and organization of a variety of FreeBSD user events

This would not be possible without support of a larger community of users and consumers, who provide certain necessary commodities:

- Beer, wine, soda, chocolate, tea, and other food/beverage-related vices in significant quantity.
- Donated and sponsored hardware, especially in racks at co-location centers, with hands to help manage it.
- Bandwidth in vast and untold quantities.
- Travel grants, developer salaries, contracts, development grants, conference sponsorship, organization membership fees, etc.
- Thanks, user testimonials and appreciation, good press.
- Yet more bandwidth.

None of these has a trivial cost–by far the most important resource for the project is developer time, both volunteered and sponsored.

7 Who are the Developers?

FreeBSD developers are a diverse team, made up of members from 34 countries on six continents. They vary in age between 17 and 58, with a mean age of 32 and median age of 30; the standard deviation is 7.2 years. FreeBSD developers include professional systems programmers, university professors, contractors and consultants, students, hobbyists, and more. Some work on FreeBSD in a few spare hours in the evening once a week–others work on FreeBSD full time, both in and out of the office. FreeBSD developers are united by common goals of thoroughness and quality of work. Unlike many open source projects, FreeBSD can legitimately claim to have developers who have worked on the source base for over thirty years, a remarkable longevity that would be the envy of many software companies. This diversity of experience contributes to the success of FreeBSD, combining the pragmatic "real world problem" focus of consumers building products with the expertise of researchers working on the cutting edges of computer science research.

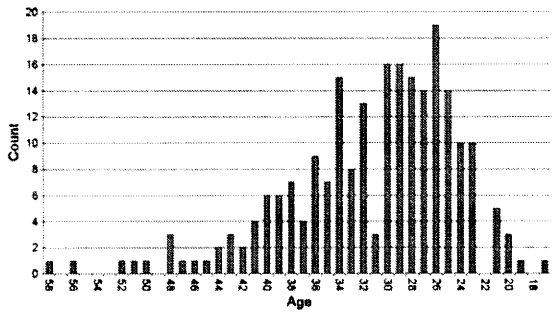


Figure 1: Age Distribution of FreeBSD Developers (2005)

8 FreeBSD Processes

The FreeBSD Project is successful in significant part because it encapsulates not just many experienced and highly competent individuals, but also because it has a set of well-defined development processes and practices that are universally accepted and self-sustaining.

- *Committer life cycle and commit bits* - The process by which new developers are inducted into the community and mentored as new members of the community is well-defined and successful.

- *Core Team* - Project leadership is selected and renewed via regular elections from the developer team as a whole, insuring both continuity, continued engagement, and fresh voices lead the project over time.

- *Mailing lists* - Through extensive and courteous use of mailing lists for almost all project communications over many years, consensus is almost universal in project decision making, and there is relatively little "stepping on toes" for a project that spans dozens of countries and time zones.

- *Web pages and documentation* - A well-designed and extremely complete set of web pages and documentation provide access to both the current condition and history of the project, from tutorial content for new users to detailed architectural information on the design of the kernel.

- *Groups/projects* - A hallmark of FreeBSD's success is the scalable community model, which combines the best of centralized software development with project-oriented development, allowing long-term spin-off projects to flourish while maintaining close ties and involvement in the central project.

- *Events* - The FreeBSD Project exists primarily through electronic communication and collaboration, but also through in-person developer and user events occurring continuously throughout the year. These include developer summits and involvement in both BSD-specific and general purpose conferences.

- *Honed development and release cycle* - With over ten years of online development and release engineering experience, the FreeBSD Project has pioneered many online development practices, combining professional software engineering approaches with pragmatic approaches to volunteer-driven open source development. One of the key elements of this approach is effective and highly integrated use of software development tools and revision control, including the use of multiple revision control systems, CVS and Perforce.

- *Centralized computing resources* - Also key to the success of the project has been the use of several globally distributed but centrally managed computing clusters, organized and maintained by project donors and a highly experienced system administration team. The FreeBSD.org infrastructure "just works", providing flawless support for the daily activities of the project.

- *Conflict resolution* - In any development project, but especially in widely distributed organizations, effective management of technical disagreements and conflicts is critical; the FreeBSD Project's history is full of examples of successful conflict resolution leading to both good technical and social outcomes.

8.1 FreeBSD Committers

A FreeBSD committer is, in the most literal sense, someone who has access to commit directly to the FreeBSD CVS repository. Committers are selected based on four characteristics: their technical expertise, their history of contribution to the FreeBSD Project, their clear ability to work well in the FreeBSD community, and their having made the previous three extremely obvious. Key to the induction of new committers is the notion of a mentor: this is an existing committer who has worked with the candidate over an extended period and is willing to both sponsor their candidacy and also act in a formal role in introducing them to the project. The mentor proposes the candidate to one of the Core Team, Port Manager, or Doceng, who respectively approve commit rights for the src tree, the ports tree, or the documentation tree. A typical proposal includes a personal introduction of the candidate, a history of their background and contribution, and volunteers to mentor them.

Once approved, typically by a vote, the new committer is given access to the FreeBSD.org cluster and authorized access to CVS. Mentorship does not end with the proposal: the mentor and new committer will have a formal ongoing relationship for several months, in which the mentor works with the new committer to review and approve all commits they will make, helps them circumnavigate the technical and social structure of the project. This relationship often continues informally in the long term, beyond the point where the mentor has "released" the new committer from mentorship. Typically, there is significant technical interest overlap between the proposing mentor and the new committer, as this will be the foundation on which familiarity with their work, as well as competence to review their work, will have been formed.

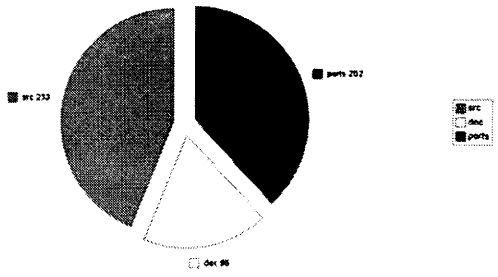


Figure 2: Number of FreeBSD committers by commit bit type (2005)

Committers often begin working in one of the various trees, and gradually spread to working in others. For example, it is not uncommon for documentation committers to expand the scope of their work to include source development, or for src developers to also maintain a set of application ports. Some of FreeBSD's most prolific and influential kernel developers have begun life writing man pages; "upgrading" a commit bit to allow access to new portions of the tree is a formal but lightweight process, in which a further proposal by a potential mentor is sent to the appropriate team for approval. As with an entirely new committer, a formal mentorship will take place, in which the new mentor takes responsibility for reviewing their commits during their earlier work with their new commit bit.

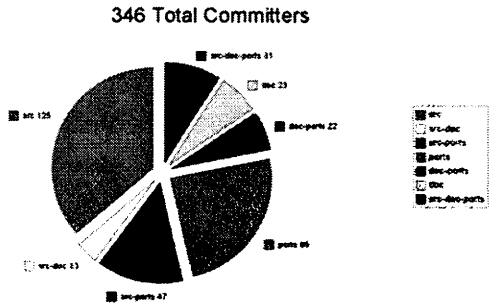


Figure 3: There is significant overlap, with many committers working in more than one area of the source tree. (2005)

8.2 FreeBSD Core Team

The FreeBSD Core Team is the nine-member elected management body of the FreeBSD Project, and is responsible for a variety of administrative activities. Historically, the Core Team consisted of a self-selected set of the leading developers working on FreeBSD; however, in 2000, the model was changed to an elected model in order to adopt a more sustainable model. Every two years, nominees from the FreeBSD committer team volunteer to be placed on the role, and a one month online election is held. The FreeBSD Core Team then appeals for and selects a volunteer to act as Core Secretary.

While the process of selecting the Core Team is well-defined, the precise responsibilities of the Core Team are not, and have evolved over time. Some activities are administrative in nature: organizing successive elections, assisting in writing and approving charters for specific teams, and approving new FreeBSD committers. Other activities are more strategic in nature: helping to coordinate developer activity, making sure that key areas are being worked in by cajoling or otherwise convincing developers they are important, and assigning authority to make significant (possibly contentious) architectural decisions. Finally,

the FreeBSD Core Team is responsible for maintaining and enforcing project rules, as well conflict resolution in the event that there is a serious disagreement among developers.

8.3 Ports Committers, Maintainers

The FreeBSD Ports Collection is one of the most active areas of FreeBSD work. At its heart, the ports tree is a framework for the systematic adaptation of third party applications to FreeBSD, as well as a vast collection of ported applications. In 2005, there were 158 ports committers working on 16,000 application ports. In addition to ports committers, the notion of a ports maintainer is also important: while ports committers are often involved in maintaining dozens or even hundreds of ports themselves, they also work to funnel third party porting work by over 1,500 ports maintainers into the ports tree. Particularly prolific maintainers often make good candidates for ports commit bits. With an average of 100 ports per committer and 11 ports per maintainer, the ports work is critical to the success of FreeBSD.

The Port Manager (portmgr) team is responsible for administration of the ports tree, including approving new ports committers as well as administering the ports infrastructure itself. This involves regression testing and maintaining the ports infrastructure, release engineering and building of binary packages across half a dozen hardware platforms for inclusion in FreeBSD releases, as well as significant development work on the ports infrastructure itself. Regression testing is a significant task, involving large clusters of build systems operating in parallel; even minor infrastructure changes require the rebuilding of tens of thousands of software packages.

8.4 Groups and Sub-Projects

The FreeBSD Project is a heavily structured and sizable organization with many special interest groups working in particular areas. These groups focus on specific technical areas, support, advocacy, deployment and support of FreeBSD in various languages and in different countries. Some sub-groups are formally defined by the project, and in some cases, have approved charters and membership. Others exist more informally, or entirely independent of the central FreeBSD.org infrastructure, shipping derived software products.

8.5 A FreeBSD Project Org Chart

While the concept of an organizational chart applies somewhat less well to a loose-knit volunteer organization than a traditional company, it can still be instructive.

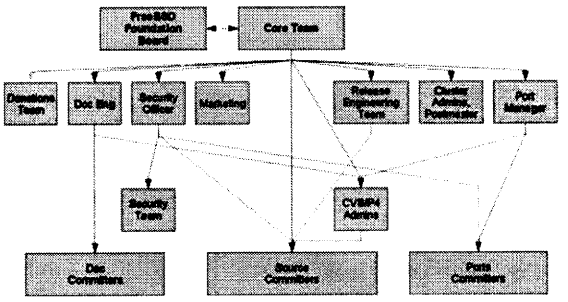


Figure 4: Lines in this FreeBSD Project Org chart represent more than just downward delegation of authority found in commercial organizations.

In a traditional organization chart, arrows would represent delegation of responsibility. In the FreeBSD Project organization chart, this is only partially true: typically arrows represent delegation of authority: i.e., the FreeBSD Core Team, the elected management body of the project has assigned authority, by means of voting to approve a written chart, for security advisory and other Security Officer activities to the Security Officer and Security Officer team. As the organization is volunteer-driven, delegation of of responsibility occurs up as much as down: the larger body of FreeBSD committers select a Core Team to take responsibility for a variety of administrative activities.

8.6 Derived Open Source Projects

FreeBSD provides components, and in some cases the foundation, of a large number of derived open source software projects.

- FreeSBIE, a FreeBSD-based live CD image
- m0n0wall, an embedded FreeBSD-based firewall package
- pfSense, an extensible firewall package based on m0n0wall
- PC-BSD, a workstation operating system based on FreeBSD
- Darwin, the open source foundation of the Mac OS X operating system, which includes both portions of the FreeBSD kernel and user space
- DesktopBSD, a workstation operating system based on FreeBSD
- DragonflyBSD, a FreeBSD-derived research operating system project

- FreeNAS, a FreeBSD-based network storage appliance project

In addition, FreeBSD code may be found in an even greater number of projects that software components developed in FreeBSD; this includes open source projects such as OpenBSD, NetBSD, and Linux systems.

8.7 Mailing Lists

Mailing lists are the life-blood of the project, and the forum in which almost all project business takes place. This provides a long term archive of project activities. There are over 40 public mailing lists hosted at FreeBSD.org, as well as a number of private mailing lists associated with various teams, such as the Core Team, Release Engineering team, and Port Manager team. Mailing lists serve both the developer and user communities. A great many other mailing lists relating to FreeBSD are hosted by other organizations and individuals, including regional user groups, and external or derived projects.

8.8 FreeBSD Web Pages

Web sites are a primary mechanism by which the FreeBSD Project communicates both internally and with the world at large. The main FreeBSD.org web site acts as a distribution point for both FreeBSD as software and documentation, but also as a central point for advocacy materials. Associated web sites for the mailing lists and mailing list archives, bug report system, CVSweb, Perforce, and many other supporting services are also hosted as part of the FreeBSD.org web site.

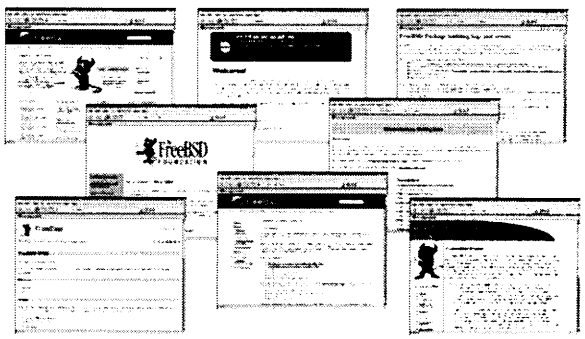


Figure 5: Web sites play an integral role in how the FreeBSD Project communicates with both users and contributors.

In addition, there are a number of project-specific web sites for FreeSBIE, TrustedBSD, PC-BSD, DesktopBSD, and others, which are linked from the main FreeBSD.org web site, but are separately authored and hosted.

8.9 Events

While electronic communications are used as the primary method of communication for most on-going work, there is no substitute for meeting people you are working with in-person. The FreeBSD Project has a presence at a great many technical workshops and conferences, such as USENIX and LinuxWorld, not to mention a highly successful series of BSD-related conferences, such as BSDCan, EuroBSDCon, AsiaBSDCon, NYCBSDCon, MeetBSD, and a constant stream of local user group and developer events.

As these conferences bring together a great many FreeBSD developers, there are often Developer Summits occurring concurrently, in which FreeBSD developers meet to present, discuss, hack, and socialize. Summits typically consist of a formal session containing both presentations and moderated discussion, and information activities, such as hacking and gathering at a bar or pub.

8.10 FreeBSD Development Cycle

FreeBSD is created using a heavily branched development model; in revision control parlance, this means that there is a high level of concurrent work occurring independently. The central FreeBSD src CVS repository contains a large number of branches; the main of these is the HEAD or CURRENT branch, where new features are aggressively developed.

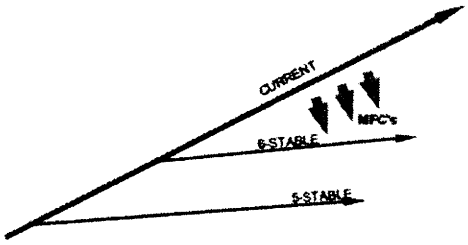


Figure 6: Branching is a key element of the FreeBSD development model: simultaneous work on several complete versions of FreeBSD at once allows changes to be merged from one branch to another as they gain stability, exposing them to successively wider testing and use.

A series of STABLE branches contains more conservative development, one per major release series, with changes being trickled from the CURRENT branch to

other branches as they stabilize; this process is referred to as "Merged From Current", or MFC. Minor releases are cut from STABLE branches at regular intervals, typically three to six months. Major releases are cut around every 18 months, although sometimes less frequently, and involve the creation of a new STABLE branch; this allows extremely large features, inappropriate for merge to a STABLE release series, to be released as part of new major (.0) releases.

In addition to the CURRENT and STABLE branches, RELEASE branches are used for release cycles as well as for security and errata patches following release.

7-current	cutting edge development
6-stable	active development with releases
5-stable	legacy branch with releases
4-stable	legacy branch

Branched development is also used extensively during early feature development. Due to limitations in CVS, discussed later, this work typically occurs in branches in the FreeBSD Perforce server.

8.11 FreeBSD Releases

Release engineering is one of the most tricky aspects of running any large software project, let alone a large-scale, volunteer-run open source project. The release team (RE) is responsible for the coordinating the combination of technical and technical engineering necessary to bring a FreeBSD release to fruition. With membership approved by the Core Team, RE is given significant leeway to steer the FreeBSD development process, including placing administrative limits on development in the tree (code slushes, freezes), performing CVS branching and tagging operations, not to mentioning begging and cajoling developers into doing that which is necessary to make a release possible.

As FreeBSD development is centered on revision control, the revision control operations involved in a release are important to understanding how releases occur. Releases occur in release branches, which are typically branched from a -STABLE development branch. In preparation for a release, development on the -STABLE branch is slowed to a more conservative set of changes in order that existing new work can stabilize. First a "code slush" occurs, in which new features are eschewed, but bug fixing and refinement occurs largely unhindered; any significant changes for the release require approval by the Release Engineering team during this period. After a period of slush, a "code freeze" is started, after which point commits to the tree may only occur with the specific approval of the release am. This change in process increases the level of review taking place for changes, as well as allowing the Release Engineering team to manage risk for the release as a whole.

A series of beta test releases will be made during the code freeze, in which major and minor problems are incrementally identified and corrected. Once the Release Engineering team is satisfied with the quality of the tree, branching of the release branch may occur, which can allow more active development on the -STABLE branch to resume. A series of release candidates is used to continue to refine the release, with successively more broad testing, especially of the install procedure, which sees less exposure during normal development. Once a final release candidate is created, the release itself may occur, and the release is tagged.

Coordinated with this process for the base tree is both a release process for the ports and documentation trees. Final third party package builds occur prior to the release candidate series, ensuring testing and compatibility after significant changes have been completed in the base source tree. The Port Manager team also places a slush and freeze on the ports tree, allowing testing of the packages together rather than in isolation. The documentation tree is likewise tagged as part of the release process; an important aspect of the release is preparation of the release documentation, including the release notes identifying changes in FreeBSD, finalization of translated versions, and updates to the web site and documentation to reflect the release.

The release branches continue to serve an important role after the tagging and release of a FreeBSD version. Once the Release Engineering team believes that there is no risk of a re-roll of the release due to a last minute issue, it will transfer ownership of the branch to the Security Officer team, which will then maintain security patches against the release in that branch. The Release Engineering team may also coordinate the addition of errata patches to the branch for major stability or functional problems identified after the release. Freezes requiring approval of the Release Engineering or Security Officer teams are not released on release branches.

The FreeBSD 6.1 release process is fairly representative, in that it contained the typical snags and delays, but produced a very technically successful and widely deployed release:

25 Jan 2006	Schedule finalized
31 Jan 2006	Code freeze begins
5 Feb 2006	Ports schedule, announced
5 Feb 2006	6.1-BETA1
19 Feb 2006	6.1-BETA2
23 Feb 2006	Ports tree frozen
3 Mar 2006	6.1-BETA3
6 Mar 2006	Doc tree slush
14 Mar 2006	6.1-BETA4; ports tagged
5 Apr 2006	RELENG_6_1 branch
10 Apr 2006	6.1-RC1
17 Apr 2006	Doc tree tagged, unfrozen
2 May 2006	6.1-RC2
7 May 2006	Release tagged
7 May 2006	Build release
8 May 2006	6.1-RELEASE released

Major (.0) releases occur in a similar manner to minor releases, with the added complexity of creating a new -STABLE branch as well as a new release branch. As this occurs quite infrequently, often as much as several years apart, the process is more variable and subject to the specific circumstances of the release. Typically, the new -STABLE branch is created after a long period of code slush and stabilization in the -CURRENT branch, and occurs well in advance of the formal release process for the .0 release. Critical issues in this process include the finalization of application binary interfaces (ABIs) and APIs for the new branch, as many ABIs may not be changed in a particular release line. This includes library version updates, kernel ABI stabilization for device drivers, and more.

Incremental releases of FreeBSD, such as the 6.1 and 6.2 releases, largely require appropriately conservative strategies for merging changes from the CURRENT branch, along with some amount of persuasion of developers to address critical but less technically interesting issues. Typical examples of such issues are device driver compatibility issues, which tend to rear their heads during the release process as a result of more broad testing, and a few individuals bravely step in to fix these problems.

Larger releases, such as 3.0, 4.0, 5.0, and 6.0, require much more care, as they typically culminate several years of feature development. These have been handled with varying degrees of success, with the most frequent source of problems the tendency to overreach. While the FreeBSD 4.0 and 6.0 releases were largely refinements and optimizations of existing architecture, the FreeBSD 3.0 and 5.0 releases both incorporated significant and destabilizing architectural changes. Both resulted in a series of incremental releases on a STABLE branch that did not meet the expectations of FreeBSD developers; while these problems were later ironed out, they often resulted from a "piling on" of new features during an aggressive CURRENT development phase.

The success of the FreeBSD 6.x release series has been in large part a result of a more moderated development and merge approach, facilitated by the heavy use of Perforce, which allows experimental features to be maintained and collaborated on without merging them to the CVS HEAD before they are ready. Prior to the use of Perforce, experimental features were necessarily merged earlier, as there were not tools to maintain them independently, which would result in extended periods of instability as the base tree ceased to be a stable platform for development. The more mature development model leaves the CVS HEAD in a much more stable state by allowing a better managed introduction of new features, and actually accelerates the pace of development by allowing avoiding slowdowns in concurrent development due to an unstable base.

8.12 Revision Control

Most major technical activities in the project are centered on revision control. This includes the development of the FreeBSD source code itself, maintenance of the tends of thousands of ports makefiles and metadata files, the FreeBSD web site and documentation trees (including the FreeBSD Handbook), as well as dozens of large-scale on-going projects. Historically, FreeBSD has depended heavily on CVS, but has both extended it (via cvsup), and made extensive use of Perforce as the project has grown. The FreeBSD Project is now actively exploring future revision control options.

8.12.1 Revision Control: CVS

CVS, or the Concurrent Versions System, is the primary revision control system used by the FreeBSD Project, and holds the authoritative FreeBSD source trees, releases, etc. [2] This repository has over twelve years of repository history. The FreeBSD CVS repository server, repoman.FreeBSD.org, actually holds four separate CVS repositories:

/home/ncvs	FreeBSD src
/home/pcvs	FreeBSD ports
/home/dcvs	FreeBSD documentation
/home/projcvs	FreeBSD project

The FreeBSD Project supplements CVS in a variety of ways; the most important is cvsup, which allows high-speed mirroring and synchronization of both the CVS repository itself, as well as allowing CVS checkouts without use of the heavier weight CVS remote access protocol. This permits the widespread distribution of FreeBSD, as well as avoiding concurrent access to the base repository, which with CVS can result in a high server load. Most developers work against local CVS repository mirrors, only using the central repository for check-in operations.

Over time, the technical limitations of CVS have become more apparent; cvsup significantly enhances the scalability of CVS, but other limits, such as the lack of efficient branching, tagging, and merging operations have become more of an issue over time.

8.12.2 Revision Control: Perforce

While CVS has served the project extremely well, its age is showing. CVS fails to offer many key features of a distributed version control system, nor the necessary scalability with respect to highly parallel development. To address these problems, the FreeBSD Project has deployed a Perforce server, which hosts a broad range of on-going "projects" derived from the base source tree. [6] The most important feature that Perforce brings to the FreeBSD Project is support for highly branched development: it makes creating and maintaining large-scale works in progress possible

through lightweight branching and excellent history-based merging of changes from parent branches to children.

Currently, most major new kernel development work is taking place in Perforce, allowing these projects to be merged to the base tree as they become more mature, avoiding high levels of instability in the CURRENT branch. Perforce also makes collaboration between developers much easier, allowing developers to monitor each other's works in progress, check them out, test them, and modify them. Projects that have been or are being developed in Perforce include SMPng, KSE, TrustedBSD Audit, TrustedBSD MAC, SEBSD, superpages, uart, ARM, summer of code, dtrace, Xen, sun4v, GEOM modules, CAM locking, netperf, USB, ZFS, gjournal, and many others. CVS remains the primary and authoritative revision control system of the FreeBSD Project, with Perforce being reserved for works in progress, but it plays a vital role in the growth of the project, so cannot be ignored in any serious consideration of how the project operates.

8.12.3 Revision Control: The Future

The FreeBSD Project is in the throes of evaluating potential future distributed version control systems as a potential successor to CVS and Perforce, with the goal of subsuming all activity from both into a single repository. The Project's requirements are complicated, both in terms of basic technical requirements, as well as being able to support our development processes and practices. Primary of these requirements is that the entire current CVS repository and history be imported into the new repository system, a task of non-trivial complexity, and that it support the new branched development model used heavily in Perforce. Another important consideration is continued support for the cvsup infrastructure for the foreseeable future.

8.13 Clusters

The FreeBSD Project makes use of several clusters scattered around the world, typically located at co-location centers. These clusters are possible due to the generous donations of companies using FreeBSD. One of the most important aspects of these donations is that they are not just significant donations of servers or rack space, but donations of administrative staff time and expertise, including hands to rearrange and handle new and failing hardware, reinstall and update systems, and help troubleshoot network and system problems at bizarre hours of the day and night.

8.13.1 FreeBSD.org cluster

While there are several FreeBSD Project clusters, The FreeBSD.org Cluster is hosted in Santa Clara by Yahoo!, and is home of many of the most critical systems making up the FreeBSD.org domain.

Mail servers	hub, mx1, mx2
Distribution	ftp-master, www
Shell access	freefall, builder
Revision control	repoman, spit, ncvsup
Ports cluster	pointyhat, gohans, blades
Reference systems	sledge, pluto, panther, beast
Name server	ns0
NetApp filer	dumpster

All of these systems have been made available through the generous donations of companies supporting FreeBSD, such as Yahoo!, NetApp, and HP. The systems are supported by remote power, serial consoles, and network switches.

8.13.2 Other Clusters

The FreeBSD.org cluster hosted at Yahoo! is not the only concentration of FreeBSD Project servers. Three other major clusters of systems are used by the FreeBSD Project:

- The *Korean ports cluster* hosted by Yahoo! in Korea provides a test-bed for ports work.
- *allbsd.org* in Japan provides access to many-processor Sun hardware for stress and performance testing.
- The *Sentex cluster* hosts both the FreeBSD Security Officer build systems, as well as the Netperf cluster, a network performance testing cluster consisting of a dozen network booted systems with gigabit networking. This cluster has also been used to test dtrace, hwpmc, and ZFS.
- The *ISC cluster* hosts half of FreeBSD.org, as well as a large number of ports building systems, the FreeBSD.org Coverity server, test systems, and more.

8.14 Conflict Resolution

Conflict resolution is a challenging issue for all organizations, but it is especially tricky for volunteer organizations. FreeBSD developers are generally characterized by independence, a good sense of cooperation, and common sense. This is no accident, as the community is self-selecting, and primary criteria in evaluating candidates to join the developer team are not just technical skills and technical contribution, but also the candidate's ability to work successful as part of a larger global development team. Conflict is successfully avoided by a number of means, not least avoiding unnecessary overlap in work areas and extensive communication during projects that touch common code.

Despite this, conflicts can and do arise: some consist purely of technical disagreements, but others result from a combination of the independence of spirit of FreeBSD developers and the difficulty of using solely

online communications to build consensus. Most conflicts are informal and self-resolving; on the rare occasion where this is not the case, the FreeBSD Core Team is generally responsible for mediating the conflict. For purely technical disagreements, reaching a decision by careful consideration (and fiat) is often successful, relying on the elected authority of the Core Team to make a final decision. As technical disagreements are often only the trigger in more serious conflicts, the Core Team typically selects a mediator (usually a Core Team member) to help work to improve communications between the disagreeing parties, not just pick a "right" technical solution.

8.15 Bike sheds

"Bike sheds" are a very special kind of conflict found, most frequently, in technical communities. First described by Parkinson in a book on management, the heart of the issue of the bike shed lies in the observation that, for any major engineering task, such as the designing of a nuclear power plant, the level of expertise and investment necessary to become involved is so significant that most contributions are productive; however, the building of a bike shed is something that anyone (and everyone) can, and will, express an opinion on. [5] Strong opinions prove easiest to have on the most trivial details of the most unimportant topics; recognizing this problem is key to addressing it. Bike sheds, while not unique to FreeBSD, are an art-form honed to perfection by the project. Since they have become better understood, they have become much easier to ignore (or dismiss once they happen). This terminology has now been widely adopted by many other open source projects, including Perl and Subversion.

9 Conclusion

The FreeBSD Project is one of the largest, oldest, and most successful open source projects. Key to the idea of FreeBSD is not just software, but a vibrant and active online community of developers, advocates, and users who cooperate to build and support the system. Several hundred committers and thousands of contributors create and maintain literally millions of lines of code in use on tens of millions of computer systems. None of this would be possible without the highly successful community model that allows the FreeBSD Project to grow over time, as well as permitting other projects to build on FreeBSD as a foundation.

References

[1] DINH-TRONG, T. T., AND BIEMAN, J. M. The FreeBSD Project: A Replication Case Study of Open Source Development. *IEEE Transactions on Software Engineering 31*, 6 (2005).

[2] FREE SOFTWARE FOUNDATION. cvs - Concurrent Versions System. http://www.nongnu.org/cvs/.

[3] FREEBSD PROJECT. FreeBSD Project home page. http://www.FreeBSD.org/.

[4] JORGENSEN, N. Putting it all in the trunk: incremental software development in the FreeBSD open source project. *Information Systems Journal 11*, 4 (2001), 321–336.

[5] PARKINSON, C. N. *Parkinson's Law; or, the Pursuit of progress.* John Murray.

[6] PERFORCE SOFTWARE. Perforce, the Fast Software Configuration Management System. http://www.perforce.com/.

[7] RICHARDS, P. eXtreme Programming: FreeBSD a case study. In *UKUUG Spring Conference and Tutorials: Conference Proceedings* (2006), UKUUG.

Organized by:

WillyStudios.com

I.T. Consulting, Web & VoIP Services

I-20040 Carnate, Milano, Via Carducci, 9
Tel. (+39) 02 44417203 - Fax. (+39) 02 44417204
P.IVA – VAT ID: IT04696330960
info@willystudios.com