USENIX Association

Proceedings of the BSDCon 2002 Conference

San Francisco, California, USA
February 11-14, 2002

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

A FreeBSD Based Low Cost Broadband VPN Router for a Telemedicine Application

Gunther Schadow
Regenstrief Institute for Health Care, Indianapolis, IN

Abstract

The author developed a small low cost broadband networking router using FreeBSD to support a telemedicine application. Our router design provides IPsec based virtual private networking (VPN) and quality of service (QoS) arrangements, simultaneously supporting two-way real-time video and audio, camera control, streaming video replay and medical record access over the public Internet using Cable-modem links from physician's homes to the Internet. These routers have been critical for the implementation of the telemedicine project after several other attempts with proprietary "solutions" have failed. FreeBSD's integrated IPsec code, QoS facility, firewall, and its ability to squeeze the entire system down into a file system image of less than 8 MB, had been instrumental for the development of our routers. Above all, however, this project would not have been possible without the availability of the system's source code, the ability to investigate bugs, rapid turn-around in the user-support community, and the ability to add missing features into existing software, which is made possible by the open source software development paradigm.

1 INTRODUCTION

We developed a small low cost broadband networking router using FreeBSD to support a telemedicine application, allowing physicians at their homes to interview patients in a nursing home over videoconference. We are currently conducting a study to test the hypothesis that audiovisual interaction with the patient through videoconferencing may enable the physician to make better decisions and reduce unnecessary referrals to the emergency room. We describe this study elsewhere in greater detail [1].

In this paper we present the design and development of our broadband VPN router that provides IPsec based virtual private networking (VPN) and quality of service (QoS) arrangements, simultaneously supporting two-way real-time video and audio, camera control, streaming video replay and medical record access over the public Internet using Cable-modem links to the Internet. We present this as a case study of a real-world application that requires many still relatively new functions of the Internet suite of protocols and the operating systems implementing it. We include some recommendations that we find would make the pieces fit together better. We also hope that the reader may find useful the description of our technical approach and the references we give.

.

1.1 Why a Custom Router?

Although funds had been allocated to use T1 connection to the 5 participating physicians' homes, our T1 provider did not install the ordered T1 lines to the physicians' homes for over a year after the contracts had been signed. We therefore had to use the public Internet instead. We found that Cable-Modem Internet service in our area had a sufficient bandwidth (2 Mbit/s downstream) even after the ISP throttled uplink bandwidth from 2 Mbit/s down to 100 kbit/s. We also realize that costs would inhibit scaling up T1 line use beyond the limited scope of this study. Being a very application-oriented project rather than genuinely a networking research and development project, the use of self-designed and open-source operating system driven devices was not initially planned; but soon became as important as to rescue the whole project.

Initial approaches were made with Cisco's PIX firewall in combination with Intel's PRO/100 S network interface cards (NIC) capable to offload encryption from the end-system's CPU. (Offloading was critical because of the high CPU and I/O load from the videoconferencing application.) However it turned out that the software kept crashing and the IPsec modes (tunnel vs. transport) did not allow interoperation with our Cisco PIX firewall. Another attempt with Cisco's broadband access router with IPsec capability disappointed because our configurations and control over the device was taken away by the Cable-Modem head-end as soon as a Cable-Internet link had been established. Other third party solutions did not provide for our specific needs (e.g., QoS.) Thus a self-developed device became reasonable.

1.2 Requirements

Our broadband VPN router addresses the following requirements:

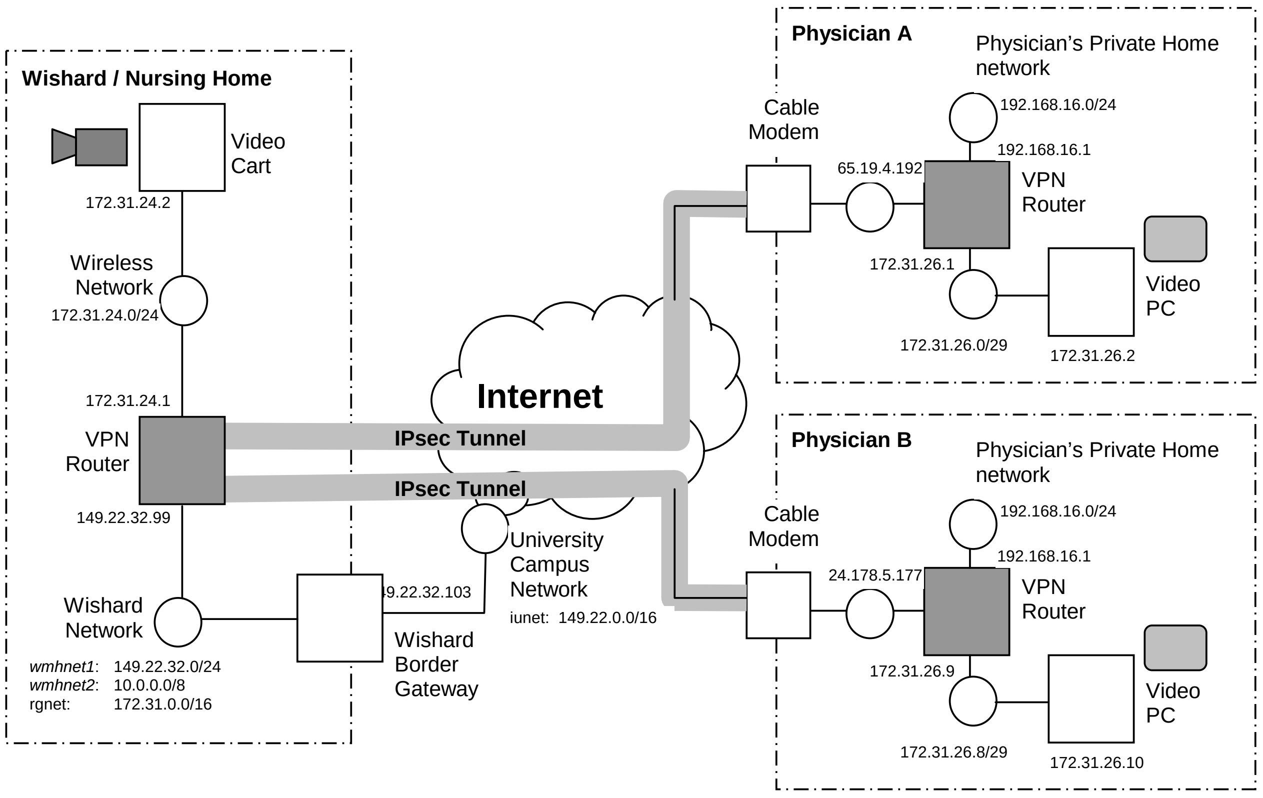

Figure 1: Topology of the Telemedicine Virtual Private Network. IPsec tunnels securely connect remote networks and establish a virtual address structure on top of public IP networks (tthe public IP addresses are disguised.)

a) Route IPv4 packets between a Cable-Modem uplink and a local area network (LAN) in the physician's home.

b) Provide network address translation (NAT) to connect several machines on the LAN to the Internet via the one IP address provided by the Cable-ISP.

c) Secure all communications between the physician's dedicated video conferencing and medical record access computer and the nursing home's LAN using strong encryption at a bandwidth of 2 Mbit/s downstream and 128 kbit/s upstream.

d) Provide a reasonable firewall protection to avoid breaches into the physician's LAN and prevent unauthorized access into the VPN through one of the remote sites.

e) Provide two separate LAN segments (requiring 3 Ethernet interfaces in total) so that the physician's and his family's private use of the Internet is kept separate from office use. In particular, prevent access to the virtual private network from the private segment.

f) Compensate for the asymmetric nature of the Cable-Internet connection with only 128 kbit/s outgoing bandwidth. Coordinate available bandwidth between the essential flows: (1) video (H.261 [2] over UDP, up to 1.5 Mbit/s), (2) audio (UDP, 64 kbit/s, fixed), (3) remote camera control (TCP, low bandwidth but need for low latency and intolerant to packet loss), and (4) medical record access (TCP/HTTP).

g) Be able to route and tunnel Novell NetWare IPX packets used in our (as in many) corporate network.

h) Be small, noise-free, affordable, reliable, and remotely maintainable.

1.3 Overview of the Design

Figure 1 shows the topology of our telemedicine VPN. Physicians are connected in their homes through cable-modems. Each home has one of our VPN routers establishing an IPsec tunnel through the Internet linking to a central VPN router in the nursing home. A portable video cart is connected through a wireless network to this central router.

The VPN is designed as an "overlay network" [3], i.e. using an address space that is distinct from but mapped to the global IP address space through the use of IP aliases to hosts and tunnels between distant physical networks. For example, as Figure 1 shows, our telemedicine network uses a "private" IP address space [4] such as 172.31.0.0/16. Nodes that are only part of that telemedicine network only have addresses in this network. Nodes that are part of multiple networks have aliases in those networks. Physically distant subnetworks are connected through IPsec tunnels. The IPsec tunnels are established between VPN routers in a star-topology with one central VPN router at the nursing home facility and multiple remote VPN routers, one for each participating physician's home.

A VPN router in the physician's home has three networks attached: (1) the Internet through the cablemodem, (2) the office network that is part of the telemedicine VPN, and (3) the physician's private network that has no route to the VPN.

Packets that travel between the office network and the public Internet are not routed through the tunnel, which would only add load to the tunnel endpoints and increase the physical path lengths.

We developed the VPN routers as a custom device based on a generic UNIX operating system (FreeBSD 4.2-RELEASE) and small generic PC compatible hardware. The assumption was that FreeBSD had all the necessary facilities already integrated and all we needed to do is configure the pieces and compile them into a form that is easy to deploy and maintain. These assumptions have for the most part held true, although we found that making all the pieces play together can be a challenge. We also found bugs and missing features.

2 HARDWARE

Since the hardware market is a fast moving target, much of our writing about our hardware must remain anecdotal, and we make no effort pretending otherwise. The reader may find helpful references to resources. We also believe that most of the issues and tradeoffs we had to face still apply today (approximately 1 year after we made our hardware decisions.)

We required a moderately fast PC compatible with solid state memory of at least 4 MB and dynamic memory of at least 8 MB, 3 network interfaces, small size and simple power requirements at a reasonable price (i.e., given the relative low-end performance of the system, we set an upper limit at $350 per complete system.) Despite these very modest requirements, we could not find a suitable product for over a year.

We monitored the market by tracking the following resources: PC/104 Embedded Solutions [www.pc104-embedded-solns.com] web site and magazine, Linuxdevices.com, and the product catalogs of many of the Single-Board-Computer (SBC) vendors such as (in no particular order) Advantech, Tri-M Systems, ICP Electronics, and Diamond-Systems, and others.

Among the trade-offs between features, dimensions, and price the limiting factor was usually the requirement for built-in network devices. Most low-cost/low-end SBCs have no network interface. Many SBCs came with one network interface (usually of the 10 Mbit/s class, 10base-T); however, those would usually also include video and sound interfaces. For us, video and sound interfaces were disadvantaging factors because they would make the product more expensive, larger, consume more power and produce more heat. Furthermore, most "embedded systems" required much additional parts and assembly. Given the difficulty of finding a suitable device with one network interface, finding one with 3 interfaces is nearly impossible in the SBC market. The very few systems we found ranged in the $1000 price class.

An alternative to SBCs, however was a desktop PC-computer of small dimensions with a limited PCI-backplane that allows adding network interface cards. On the FreeBSD "small" mailing list we were hinted to FlyTech's line of small PC products, in which we found the NetPC NC-2 B62 (Intel Celeron, 533 MHz, 64 MB DRAM, 8 MB Disk-on-chip, 1 RealTek 100base-T NIC.) With 2 Intel PRO/100 network interface cards added, and at a total cost of approximately $500, this product was a viable option to implement the project for the scope of the clinical trial. We are still using this FlyTech computer for the central VPN router at the nursing home.

Of particular interest in the low-cost embedded-PC market are so called PC-on-a-chip designs such as the National Semiconductor GEODE, the AMD SC520, and the ZF Micro Devices ZFx86. These chips combine most of the electronics of a common PC motherboard in one chip which dramatically reduces the dimension and power-consumption. They can reduce cost as well, provided that the board design and manufacturing is as cost effective as the mass-production of standard PC motherboards. Having a board custom-designed was not a good option for us, since we only had funds for 30 units to be purchased initially. (The cut-off at which custom design is feasible is at about 100 units.)

For SBC applications, a PC-architecture may not be the smallest or most cost-effective approach. Since in the first months of searching we could not find a reasonable PC hardware we investigated other platforms such as the Intel StrongARM and MIPS. These hardware platforms are supported to a certain extent by both NetBSD and Linux. So, we would have had an avenue of implementing our project.

We finally found a small engineering firm, SOEKRIS Engineering [www.soekris.com] who had designed a device, "net4501," based on the AMD Elan SC520 PC on a chip. This device has 3 100base-T network interfaces on board and a serial interface, no video- and sound. One low-power PCI and one mini-PCI connector provides the flexibility to accommodate special needs in the field. The SOEKRIS net4501 board comes with a CompactFlash socket for a solid-state memory, which turned out to be cheaper and much easier to work with than the Disk-on-chip devices that most SBC products (including our FlyTech Net-PC) use. The board needs no airflow cooling requiring no moving parts and emitting no noise at all. The hardware is available for approximately $200 including a simple enclosure and a wall-mount AC transformer (enclosure and power supply can be a major cost factor in the deployment of SBCs.)

We had the privilege of testing one of a few samples of the SOEKRIS net4501 board and were convinced that it would be close to the ideal device for our needs. The only problem of this board was that production was delayed due to the economic circumstances and we didn't have the purchasing power to jump-start production. However, a few months later the board did go into production in time for our project to use it.

3 SOFTWARE

We developed the software based on FreeBSD. We choose BSD over Linux because of the KAME IPsec implementation that is becoming a reference implementation. At that time, KAME had just succeeded and partially subsumed other open source IPv6/IPsec projects (WIDE, NRL, INRIA.) Most of these early IPv6/IPsec implementations had BSD as their primary target platform. FreeBSD's built-in support for small systems ("PicoBSD"), its large user base, and focus on stability made it most attractive among the other BSD systems, NetBSD, and OpenBSD.

We begun development on this project based on the 4.0-RELEASE of FreeBSD. We loosely track the FreeBSD RELEASE branch rather than keeping abreast of STABLE or even CURRENT, because having deployed the systems, we can not tolerate very frequent changes as we do not have the resources to test the system with every new build on a weekly or daily basis. However, we did find that the KAME IPsec code available in the RELEASE was not up to the latest critical bug-fixes, hence we usually had to apply a certain KAME "snap-kit" on top of the major FreeBSD release.

We will present the software design along the lines of the major requirements: (1) firewall; (2) network address translation (NAT); (3) IPsec-based VPN; (4) quality of service (QoS) arrangements; and (5) Novell IPX protocol routing and tunneling.

3.1 Firewall

In a VPN, firewalls are essential security elements more so than in a corporate network without VPN. This is because each remote site is mapped into the internal corporate network by means of VPN tunneling but the corporate network administration has less physical control over the remote site. Thus, a remote site can become an open door for intruders into the corporate network if not sufficiently protected against such attacks. Most importantly, the firewall must delete all incoming packets that have a source or destination address into the corporate network.

Several different firewall packages are available for open source UNIX systems. Initially, FreeBSD has a "native" IP-Firewall (IPFW) facility. Even though we found the IPFW design and integration into the system very useful, we did switch to Darren Reed's IP-Filter (IPF) [5] system early on. IPF has the advantage over IPFW of being available for systems other than FreeBSD, NetBSD in particular (as we alluded to above, we planned for migrating to NetBSD had we not found the SOEKRIS hardware in time for the project to continue.) The IPF design and implementation claimed to be more secure (e.g., it allowed updates to the firewall rules without inconsistent temporary states) and better monitored, if only because IPF had a larger user base, (including all BSDs, Linux and even many commercial UNIX systems, such as Solaris, IRIX, and HP-UX.)

To the user, the difference between IPFW and IPF is mostly the different rule syntax. IPFW has a richer set of functionality available for the filtering rules, such as dropping, NATing, forwarding, queuing/delaying. Through the "divert socket" one can tie user-defined special packet handler processes outside the kernel into IPFW (the NAT handler natd(8) being such an extra-kernel process.) Conversely, NAT with IPF are two distinct kernel facilities configured with ipf(8) and ipnat(8) using different configuration files and only a loosely related configuration syntax.

3.2 Network Address Translation (NAT)

We have to perform network address translation (NAT) on both the remote and the central VPN routers. The remote VPN routers perform NAT to allow communication between the office and private networks and the Internet to which the cable-ISP provides only one IP address. The central VPN router also performs NAT for communications of the video cart to the Internet. In addition, the central VPN NATs all communications coming out of the tunnel with destination in a public network (e.g., hospital LAN, university campus network that.)

The same choices as for the firewall exist for NAT: IPFW and IPF. We found that more complex rules could be more easily expressed with IPFW than with IPF. For example, one of our rules is to apply network address translation (NAT) only to destinations not routed through the VPN tunnel and all destinations if traffic originated on the "private network." With IPFW one can fine-tune every NAT rule exactly for each source and destination and all the other criteria available for filtering, and one can reuse the same NAT resource pool (e.g. a set of mapped port numbers) for multiple rules. Conversely, with IPF's NAT facility, ipnat(8), the criterion language is only a minimal subset of the ipf(8) filter language.

3.3 IPsec

For the beginner, IPsec comes in a confusing number of modes (tunnel/transport, configured/negotiated security associations, pre-shared keys/other authentication) [6] and various alternative ways to configure them with BSD/KAME. In addition, KAME is not the only alternative. Notably, Pierre Beyssac's pipsecd program, implements limited IPsec function using the generic tunnel pseudo-device tun(4). The tun(4) device is a standard part of BSD and Linux and represents a network interface that passes packets to a handler program, which, in the case of pipsecd, can encrypt and encapsulate the packets and pass them on. Finally, not all of KAME is equally well tested and integrated into FreeBSD. All of these factors tend to confuse the issue of getting an IPsec system operational. We will address each of them in this subsection.

ESP, AH, IPCOMP, some or all: The choice of using the Encapsulating Security Payload (ESP) protocol [7] or the Authentication Herader (AH) protocol [8] alone is simple: if encryption is part of the requirements (as it usually is), ESP is the only choice. However, an ESP security association (SA) can be configured to perform header authentication (AH) or packet compression (IPCOMP) in addition to encryption.

To decide which protocols and algorithms to use we measured the impact of many of the choices on throughput. We found that the combination of encryption (e.g., with 3DES-CBC) and header authentication (e.g., with HMAC-SHA1) reduced the throughput to less than 1 Mbit/s on a Pentium 120 MHz PC. So we decided to only use encryption because we have to support videoconferencing bandwidth of up to 1.6 Mbit/s in each direction. With random keys of 256 bit length, encryption alone provides a sufficient level of packet authentication. For encryption we found that the Blowfish-CBC algorithm is the fastest available with KAME, hence we use Blowfish-CBC with 256 bit random keys.

Configured SAs vs. ISAKMP The IPsec protocols have been designed with the Internet Security Association and Key Management Protocol (ISAKMP) [9] in mind. A security association (SA) is identified by a security parameter index (SPI) and specifies a set of algorithms and keys which two endpoint hosts use to encrypt (or authenticate) their exchange. These parameters are negotiated by the ISAKMP agents on demand, or, alternatively, can be configured statically.

Using ISAKMP can be the only available option, if KAME is to interoperate with certain commercial IPsec implementations, such as Microsoft Windows 2000, Intel PRO/100 PacketProtect, and the Cisco PIX firewall (Cisco's IOS can use static configured SAs.)

The ISAKMP agent for KAME is called 'racoon'. Racoon appears to be one of the less mature components of the KAME suite and it has not been made part of the standard FreeBSD release as of 4.4-RELEASE. We found that in order to compile and use racoon at all, we had to work from a recent KAME snap-kit. Racoon as part of the FreeBSD ports collection was mostly out of date and racoon as part of the KAME snap-kit would require all of that snap-kit. The only alternative to racoon is isakmpd as developed for OpenBSD and although isakmpd can in theory be used on other BSDs and with KAME, we gather that isakmpd is not much more mature.

Since at the beginning of this project, stability and performance of racoon was questionable, we decided to use statically configured SAs initially. Configured SAs are set up typically at time of system boot using the setkey(8) shell command as follows[1]

```
setkey -c <<END
  add $myend $hisend esp $spi_out
    -E $algo $key;

  add $hisend $myend esp $spi_in
    -E $algo $key;
END
```

Each direction from *myend* to *hisend* and vice versa has its own SA with its unique security parameter index (SPI). The SPI is part of the IPsec packet and allows the host to match an incoming packet with its SA. When setting up a multi-way VPN system with configured SAs assigning unique matching SPIs *spi_out* and *spi_in* on each VPN router node is important. If SPIs don't match up, the packets cannot be processed and are lost. In order to be sure that SPIs do match up, we let each end calculate the SPIs using a common formula that combines the overlay network addresses and the role of the endpoint as central (server) or remote (client).

[1] For all code examples, we assume the Bourne shell language sh(1), and we use shell variables as addresses for brevity and readability.

Tunnel vs. transport mode IPsec in tunnel mode is the approach of choice when implementing a VPN for a heterogeneous network. When some machines are not natively IPsec capable, using IPsec in end-to-end transport-mode is not an option. Also, when certain systems may be located behind NAT components, transport mode is likewise not possible, because ESP transport mode encrypts the port numbers which are required for the commonly used port-based NAT.

The normal way to set up an IPsec tunnel only requires defining security policies (SP). If ISAKMP is not used, a pair of SAs is also needed as discussed above. The SPs are set up with the same shell-command setkey(8).

```
setkey -c <<END
  spdadd $mynet $hisnet -P out ipsec
   tunnel/esp/$myend-$hisend/require;

  spdadd $hisnet $mynet -P in ipsec
   tunnel/esp/$hisend-$myend/require;
END
```

The X-Bone [3] and many online tutorials about implementing IPsec tunnels with KAME [10,11], however, suggest implementing IPsec tunnels using IP-in-IP tunneling in combination with IPsec in transport mode. Both approaches result in the same wire format, the difference is only how they are managed [12].

```
ifconfig gif0 $meonhisnet $hisnetmask \
              tunnel $myend $hisend link1

route add $hisnet -interface gif0

setkey -c <<END
  spdadd $myend $hisend -P out ipsec
    transport/esp//require;

  spdadd $hisend $myend -P in ipsec
    transport/esp//require;
END
```

Separating the tunnel configuration from the IPsec processing makes it easier for the user to test and debug the tunnel before enabling IPsec processing where SPIs, protocols, keys, etc. must all match up properly.

Because with IP-in-IP tunnels the VPN overlay network *hisnet* is an entry in the routing tables (even without the explicit 'route add' command shown in the example), packets originating from the tunnel-endpoint and with destination to *hisnet* will be routed into the tunnel. Conversely with only the IPsec tunnel policies, packets originating on the tunnel host will not match the tunnel policy because their source address (*myend*) is not in the space of VPN overlay addresses, but in the space of global IP addresses. Thus one usually needs 4 machines to fully test and debug an IPsec tunnel.

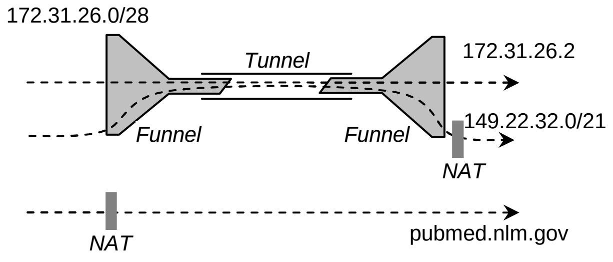

Figure 2: Tunnel remotely links two private subnets. Additional policies act like a funnel routing flows between other networks through the tunnel.

The IPsec tunnel implementation before June 2001 suffered from a severe bug which would cause failures from inbound ESP packets not being properly processed all the way to sudden kernel-panic conditions when more than one pair of tunnels was configured. Fortunately, this bug was fixed by Jun-ichiro (Itojun) Hagino after we could sufficiently isolate the conditions causing this bug to become manifest. This bug affects all FreeBSD releases up to and including 4.3-RELEASE.

Tunnels and Funnels Each physician's home site has two SAs that establish a tunnel to the central nursing home VPN router. The tunnel connects the physician's telemedicine network to the nursing home's telemedicine network. However, the tunnel is not only used for traffic between these two VPN overlays, but for all traffic between the physician's office network and the hospital network and adjacent networks. This provides and extended shell of security wherein other applications, such as the web-based Electronic Medical Record system can be secured as well. Many corporate IPv4 networks today face similar issues, where several IP address spaces exist that need to be included into the VPN without renumbering the entire network.

Thus one tunnel between the central VPN router and each remote site must have multiple ingress and egress rules that "funnel" the packets between the physician's office network and the extended networks through that tunnel.

```
funnel="$wnet1 $wnet2 $rgnet $iunet"

for i in $funnel; do
 setkey -c <<END
  spdadd $mynet $i -P out ipsec
   tunnel/esp/$myend-$hisend/require;

  spdadd $i $mynet -P in ipsec
   tunnel/esp/$hisend-$myend/require;
END
done
```

Corresponding policies need to be set up on the central server in the nursing home. These policies must direct the flows between the extended networks and the physician's office network through the tunnel.

```
for i in $funnel; do
 setkey –c <<END
  spdadd $i $hisnet –P out ipsec
   tunnel/esp/$myend-$hisend/require;

  spdadd $hisnet $i –P in ipsec
   tunnel/esp/$hisend-$myend/require;
END
done
```

Interactions between IPsec, NAT, and firewall Both IPsec policies and firewall or NAT packet filters are based on the same kind of criteria, i.e., source, destination address protocol and port numbers, etc. In absence of a common framework of filtering rules, it becomes a problem in what order IPsec and firewall filters are executed. No standard exists for the order in which IPsec and packet filters are applied. For KAME IPsec in combination with the IPF firewall/NAT suite the following procedures apply:

a) Incoming packets are first processed by filter and NAT rules and then handed over to the IPsec policies.
b) Outgoing packets are first processed by IPsec and skip outgoing filter or NAT rules.

These procedures are useful because outgoing packets, once processed by IPsec rules, do not require additional NATing or filtering while all incoming packets are first subjected to the firewall rules for intrusion enforcement. This requires enabling passing-rules for the IPsec protocol numbers 50 and 51 for AH and ESP respectively on the inbound leg, but no corresponding rules on the outbound leg. However, with other firewalls, the order may be different and filter rules may be needed for outgoing IPsec or for the incoming packets after IPsec decryption.

3.4 Quality of Service

With coast-to-coast Internet bandwidth in the range of several megabit per second, the need for quality of service arrangements is easy to overlook until one encounters a problem that can only be solved with QoS. In our case we had a bottleneck at the uplink of the remote location, where the cable-modem limits outgoing bandwidth to approximately 100 kbit/s. Such bottleneck can make a H.323-based video-conferencing system unusable. Given that each application loses packets at the bottleneck in proportion of their bandwidth, the more critical but less bandwidth-intensive applications will be impacted more than the highly-redundant video. In our case, audio was delayed by up to 10 seconds and cut out frequently so as to become useless; remote camera control was irresponsive.

The goal of a QoS arrangement under such circumstances is to allocate different bandwidth to the outgoing flows according to the need of the overall application rather than the greed of the individual part. An H.261 video system will try to use as much bandwidth as can be delivered without packet loss; on the other hand, the H.261 codec can cope with lower bandwidths and is automatically adjusted when packet drops are detected. The task of the QoS facility is therefore to drop many packets of the video stream forcing it to lower its output. Conversely, a 64 kbit/s MP3 audio codec will always need those 64 kbit/s of bandwidth to function. TCP-based camera control (typically based on a terminal-server type circuit) will in theory adjust to decreased bandwidth, but not without delays intolerable for a control-feedback circuit. Thus, our QoS arrangement must allocate 64 kbit/s for audio, a small but prompt 9.6 kbit/s for camera control, and must confine the video signal to a budget of approximately 10 kbit/s (discounting other TCP flows that also were given priority over the outgoing video signal).

The only reliable way of discerning the three flows (session control, video and audio) from our video conferencing application (VCON) is by the type of service (ToS) IP-header field.

FreeBSD provides two alternatives to such a QoS scheme, DUMMYNET and ALTQ. DUMMYNET is part of the IPFW suite. Initially invented to simulate realistic network behavior, it can be used for allocating different bandwidth limits and delays for different flows. The three main reason not to use DUMMYNET/IPFW for QoS is if (a) we don't use IPFW but IPF, (b) IPFW had no way of filtering packets by ToS field, and (c) DUMMYNET is too inflexible with bandwidth allocation.

The KAME suite of "Next-Generation-Internet" (NGI) protocols includes Kenjio Cho's Alternate Queuing Framework (ALTQ). ALTQ provides a traffic flow definition language, and several queuing disciplines, including FIFO, priority queue, Weighed Fair Queuing (WfQ) and Class-based Queueing (CBQ). For most applications with a few defined flows, CBQ appears to be the queuing discipline of choice.

After allocating 70 kbit/s for audio, 10 kbit/s for camera control and 15 kbit/s for miscellaneous use (e.g., web-based medical record access), only 5 kbit/s were available for video. This is an impossibly small budget through which one can not expect any reasonable video. In this situation, CBQ allows bandwidth to be borrowed from parent to child classes if the bandwidth available for the parent class is not fully used. This allows excess-bandwidth allocated to audio and unused camera control bandwidth to be used for video. In practice we can use between 10 and 30 kbit/s for the video channel. This video signal would not be useful for examining patients, but is enough for providing an image of the physician to the patient, which the patient usually appreciates.

Queuing is most effective at the ingress of a bottleneck, in other words on an outbound direction of an interface. However, ALTQ can also measure, mark and drop packets on an inbound direction of an interface. We use this feature to reduce excess load of data into the router, avoiding excess encryption of data that has no chance of being forwarded.

Interaction between ALTQ and IPsec Since any QoS scheme needs to inspect the outbound traffic to discern flow classes, it would be impossible to use QoS effectively on encrypted traffic. Since most bottlenecks are at the ingress of or within public networks, IPsec and QoS could almost never be used were it not for the optional "ECN-friendly" behavior [13] that KAME implements in its IPsec or IP-over-IP tunnels. ECN-friendly means for outbound packets that the tunnel agent copies most ToS bits of the payload packet into the ToS bits of the tunnel IP header. On inbound traffic the congestion notification bit of the tunnel IP header is copied into the ToS bits of the payload packet after decryption (hence the name "Early Congestion Notification", ECN). ECN-friendly behavior is the only way to let ALTQ discern traffic classes in an encrypted tunnel.

Since the ToS field through ECN-friendly behavior is the only way to use ALTQ on an encrypted tunnel, one has to reflect all traffic flows by a specific marking in the ToS bits. This can be done by using the ALTQ on the inbound direction such that characteristics as source and destination address protocol and port numbers can be used to classify the unencrypted traffic. Each packet is then marked with ToS bits indicating its flow class. At the outgoing interface IPsec ESP traffic will be classified only by ToS bits and queuing can proceed as usual on the encrypted traffic.

3.5 Novell NetWare IPX Tunneling

In a corporate networking environment, one frequently has to deal with certain non-IP protocols, most likely IPX. We use FreeBSD as a router to shield the wireless network from excess traffic, and hence we have to route IPX as well as IP. We also, experimentally, support IPX tunneling through the VPN.

FreeBSD includes some support for IPX protocols including multiple Ethernet frame types (Ethernet-II (by default) and 802.2, 802.3 and SNAP through the pseudo-device "ef"), IPX forwarding, and bindary-mode Novell NetWare protocols (file system, printers, login.) IPX routing requires nothing else than enabling the respective IPX forwarding kernel option with sysctl(8) and running the IPX routing agent IPXrouted(8).

For IPX tunneling, we now use Boris Popov's pseudo-device "nwip" (if_nwip.c) that mimics an Ethernet device and encapsulates the packets through UDP/IP packets to a certain tunnel peer. This UDP/IP traffic is then subject to our IPsec tunneling.

As FreeBSD evolves, Boris Popov's nwip code is, however, exceedingly outdated. We had to modify the code to work properly with FreeBSD 4.2-RELEASE, and we could not make it work with 4.4-RELEASE yet. Since this nwip code doesn't appear to be actively maintained any more, we plan to develop a new nwip facility outside the kernel using the generic Ethernet tunneling pseudo-device tap(4).

4 DEVELOPMENT, DEPLOYMENT

FreeBSD comes with a development suite for small stand-alone systems, called "PicoBSD". The FreeBSD installation process itself depends on such small systems to bootstrap a new machine. One can fit a limited system on a single floppy disk, including kernel, programs and configurations (for another PicoBSD adaptation to the SOEKRIS net4501 see http://sourceforge.net/projects/thewall.)

Squeezing a stand-alone operational system on a 1.4 MB floppy disk is possible due to three key functions: (a) the FreeBSD boot loader that can load a compressed image of a memory-resident file system from the diskette and mount it as the root file system; (b) the boot loader that can decompress gzip-compressed a kernel image; and (c) the crunchgen(1) utility that combines multiple programs into one statically linked binary file. This spares many copies of library code that would otherwise be linked repeatedly to each of the programs without the overhead of shared libraries.

The PicoBSD development suite allows the user to generate small BSD system disk images controlled by configuration files in combination with a menu-driven interactive tool src/release/piocbsd/build/build.script. We borrowed the principle approach to developing our system images from the PicoBSD suite, but we have modified the configuration and workflow significantly.

The menu-driven system did not lend itself well to automatically producing several images of the same kind with some minor variations. We therefore replaced the menu-driven system with a straight-forward Makefile approach. Different configurations of images are built by setting different options when make(1) is invoked. The major options are HOST and MODEL. The HOST determines the statically configured host name, IP addresses, IPX addresses, and VPN tunnel configurations, etc. The MODEL determines on what kind of hardware this host is to be instantiated. Our supported models are (1) the FlyTech NetPC with 8 MB Disk-on-Chip and (2) the SOEKRIS net4501 with 16 MB CompactFlash device.

Conserving Memory Because we have to disable swapping to solid state flash memory, we must reduce memory usage by reducing the kernel to the bare minimum and keeping the memory-resident root file system small. Our root file system uses only 44% of 2 MB compared to the FreeBSD install floppy "mfsroot" that uses 78% of 2.88 MB.

The reduction in size of the root file system is possible because unlike the install floppies, we have a relatively generous 8 or 16 MB flash-ROM available to store most of the program binaries. On the root file system itself, we only use 688 kB for a crunched binary (1822 kB on FreeBSD's mfsroot.) Our root binary only contains init(8), sh(1), mount(8), and fsck(8). When the system boots, init is started that uses sh to execute the autoboot script /etc/rc. This first mounts the flash-ROM disk to /flash and copies a field-editable set of configuration files to /etc. Then a script /etc/rc.real performs a subset of the usual autoboot tasks.

To further reduce memory use, the bulk of the programs are distributed over 3 other crunched binaries. (1) A *setup* crunch file (800 kB) contains only those programs that are needed during initialization (e.g., ifconfig, route, setkey, etc.) (2) A *system* crunch file (900 kB) contains all daemon programs and those programs that run most of the time, and thus, whose code segments would be paged into physical memory anyway. (3) A *user* crunch file (1.9 MB) contains those programs that are only needed when an administrator logs in for remote maintenance, and includes most of the essential UNIX system management and productivity tools (e.g., ps, netstat, ping, telnet, fetch, grep, sed, vi, find, etc.) For each available program the memory-based root file system contains a symbolic link from the usual directories /bin, /sbin, /usr/bin, /usr/sbin, and /usr/libexec to the respective crunched binary.

Robustness and Maintenance The flash file system is mounted in read-only mode so that the router can be powered down without concern to corrupting the boot file system (we don't actually require fsck(8) in the root binary.) The flash file system is only written to at two occasions: saving a field-customized configuration and upgrading the whole system image. For saving customizations applied in the field, the flash file system is remounted writeable and contents of the /etc directory of the running system written to a compressed tar file. This file is reloaded into the running system on rebooting just before the normal /etc/rc autoboot script is executed.

System image upgrade is even simpler: we simply use fetch(1), a command-line HTTP client, to copy the new system image directly to the flash-ROM of the running system, then the system is rebooted. Replacing a mounted flash-ROM file system on-the-fly has been a very reliable process and has caused no problems. Only when the download is aborted incompletely can the system become unusable and requires physical access to recreate. To reduce this small risk in the future, we plan to save a compressed system image on the memory filesystem first, before we copy it to flash-ROM. This is the main use where a memory file system with ample free space is invaluable.

Having worked with both kinds of flash-ROM media, Disk-on-Chip and CompactFlash, we find CompactFlash much easier to work with. Being a "consumer product," CompactFlash is much less expensive (generally less than ½ of the Disk-on-chip prices.) But most importantly, the Disk-on-chip devices emulate disk geometry very disadvantageous if not erroneous, causing the boot process to fail on Disk-on-Chip if the compressed kernel and root memory file system image uses blocks above the first 2 MB (approximately) of the disk.

Currently our VPN routers start from a statically configured system image created by the development environment just described. Should configuration parameters change, the system administrator can log into the remote router and manually modify some configuration files and save them back to flash-ROM. Alternatively, changed configurations can be branded on new images and then reinstalled.

The need for manual interaction with the router systems is kept low, and indeed, we have not laid a hand on most of the deployed systems for several months. We have added some preliminary web-based monitoring facility running the simple-http web server that is part of the special PicoBSD source code. We also run a DHCP server (ISC DHCP3) on both the office and private LAN segments so that machines connected to these segments need no manual IP configuration.

5 CURRENT WORK IN PROGRESS

We are currently working some important upgrades to our VPN routers with the objective of making configuration and deployment even simpler and more flexible. These changes include: (1) have outside IP addresses dynamically assigned the ISP's DHCP mechanism, (2) make all devices load their configuration parameters through the network rather than from flash-ROM, and

(3) have all tunnels established dynamically rather than with statically shared keys.

With this redesign of the autoboot process we have removed all configuration parameters that we anticipate being subject to change from the static configuration files (e.g., /etc/rc.conf, /etc/resolv.conf, /etc/dhcpd.conf, and others). A booting system first sends a DHCP request to the outside interface connected to the ISP. When an Internet-link is operational, the system queries all its configuration parameters through DHCP. This includes IP and IPX network addresses and masks, IPsec tunnel and funnel policies, QoS parameters, DHCP server configurations, etc.

In order to query for individualized parameters, each system needs to know its own identity. The only source of identity for a VPN router system is its X.509 certificate and matching private key. Any two system images are exactly the same except for a unique key and certificate file loaded onto the flash-ROM. When the system boots it extracts its own DNS name from the certificate and can then query all configuration parameters as that DNS-name's sub-domains.

The public key infrastructure and certificate is implemented using OpenSSL and kept very simple. Besides the necessary public key information, our certificate profile only include the issuer distinguished name (DN) and a subject alternate name ("general name") of type DNS. Because OpenSSL's "ca" tool does not work for our simple certificates without subject DN (which is a valid certificate as per the PKIX specification [14]), we only use the "req" and "x509" tools and we had to modify parts of OpenSSL. We also added a function into the x509 tool that lets us extract specifically the DNS subject alternate name from a certificate.

In order to set up tunnels dynamically, racoon supports authentication via X.509 certificates and a "passive mode" where security policies are dynamically generated from the client ISAKMP proposal (generate_policy on). Notice the difference between dynamically negotiated security associations (SA) and dynamically established tunnel security policies (SP). Every ISAKMP agent can negotiate SAs, but usually the tunnel SPs still need to be statically configured. However, as we turn to having the ISP assign our VPN routers' outside IP addresses through DHCP, we cannot establish fixed tunnel policies, because that would require knowing the remote tunnel endpoint address.

On the client side, tunnel policies are set up through the DNS based remote configuration. With racoon's *generate_policy* option enabled on the central VPN router, the server's racoon will add a pair of SPs that match the SPs on the client that triggered the initial contact between client and server. However, we found that this alone will not support our funnel policies. The problem is that after the SA is established, ISAKMP will not be involved in any further data exchanges through the tunnel. The client assumes that the tunnel is established and funnels other traffic through the tunnel, however, on the server side no funnel policies exists apart from the one policy that took effect for the first contact. We have modified racoon and added support for funnels. One can now configure racoon to use a certain shell script when an initial contact between tunnel client and server is made. That shell script will then query the DNS-based configurations to determine and establish all the funnel policies. (We borrowed the idea of a shell-escape in a daemon process from the ISC's DHCP client.)

In the future we will use the hardware encryption option based on the HiFn 7951 that is available from SOEKRIS as a mini-PCI module for the net4501 or a full size PCI module for other systems. At a cost of $80 this chip can significantly reduce the CPU cost of encryption and public-key cryptography. At this time, however, this chip is supported on OpenBSD but not yet on FreeBSD.

6 DISCUSSION

Our task was generally to install and configure existing open source components to form one functional and maintainable system, not to develop these components from scratch. Thanks to the work done by others we have been quite successfully in our project. Although we did have to use kernel debugging techniques and modify some components, there were very few problems that others did not fix promptly or that we could not fix ourselves. In this section we list the major issues we found.

6.1 Maturity of the IPsec and KAME

While the KAME IPsec implementation has an excellent track record of interoperability testing [www.vpnc.org] we found several bugs and issues that apparently are manifest only in more advanced and complex scenarios that we had to work with to integrate our system into an existing networking infrastructure. We had network blockage and kernel crashes related with multiple SAs and SPs. We were very pleased with the rapid bug fixes we received once our problems had been isolated. However, our experience suggests that the testing scenarios in interoperability tests may be too simplistic to make those tests valid for "real-world" applications.

Stale SAs, an IPsec/ISAKMP robustness issue. We are not entirely confident yet whether the use of

ISAKMP and racoon in particular will be robust enough. One possible problem case is that the server's and the client's SA data may not be always in synchronization. For instance, when the server is rebooted its SAs are lost, while the client still holds on to the now stale SAs with the server. The client will continue to send IPsec packets to the server using the stale SA, whereas the server does drops those IPsec packets because they don't match any of the server's SAs. The client has no way to notice that the tunnel is broken and the server does not reinitiate an ISAKMP negotiation for a new SA.

This appears as a flaw in the IPsec/ISAKMP protocol design: IPsec with ISAKMP depends on a state (the SA database) that two peers negotiate and must maintain synchronized, but it has no way of promptly recovering from one peer loosing its state. Obviously, the fact that ISAKMP negotiated SAs expires after a certain time (usually several hours) will cause this problem to be solved automatically at time of SA expiration. However, the protocol is still not robust if network downtimes can happen, even if only for several minutes. Also, the ISAKMP negotiation is too costly both in terms of elapsed time and CPU time to expire SAs very frequently. We believe that the IPsec/ISAKMP protocol needs to be amended to provide for detecting and resolving the problem of stale SAs promptly.

6.2 Excessive IP Packet Matching

Most network components such as (1) router, (2) packet filter, (3) NAT, (4) IPsec policy engine, (5) IPsec association matcher, (6) traffic shaper, (7) traffic conditioner, (8) raw IP handler, and (9) IP-in-IP tunnel handler all use rules that operate on some characteristics of IP packets, typically of the IP header. Even though the task of matching packets with criteria is common to all of these components, each uses its own syntax for the criteria and/or its own code to compare actual packets against those criteria. Not only will users have some added difficulty mastering the many criteria languages, the duplication in poorly optimized (linear search!) matching code contribute to "kernel bloat" and increase the CPU time spent on each packet flowing through the system. We feel that the packet processing components and their configuration could be better coordinated perhaps using a more efficient packet matching algorithm such as the Berkeley Packet Filter (BPF) [15].

6.3 Automation-Friendly Tools

UNIX is well-known for its large set of tools that "do one thing and do it well" [McIlroy in 16]. Together with pipes and the shell command interpreter, one can build powerful automated scripts quickly. We do this extensively creating most of the networking components' configuration files from templates and generators at system boot time. In a few cases we saw no choice but to create some of our own tools.

IP address calculations. We missed a tool that would allow us to do some common address calculations on the fly. This was necessary because we wanted to reduce the number of manually maintained configuration parameters but the various configuration files required many parameters that could be derived from fewer parameters. For example, some components (e.g., dhcpd(8)) require netmasks (e.g., 255.255.255.0) for specifying subnets while others (e.g., setkey(8)) would take only network prefix lengths (e.g., /24). So we had to have a tool that could convert between both. Some would require a host-independent network address (e.g., setkey(8) or route(8)) that could be derived from an IP address in the network (e.g., 172.31.24.1/24 has network address 172.31.24.0). For these and other address calculations we developed *ipac*, am IP Address Calculator. The ipac tool can increase or decrease host numbers, or network numbers, test if one network subsumes another network and convert addresses in different representations (hexadecimal, dotted decimal, simple decimal). This tool simplified many of our configurations and we believe it could be useful for a broader community. (The only shortcoming being that it only supports IPv4 addresses at this time.)

Querying the DNS. While there are many tools to query the DNS, such as nslookup or dnsquery, all of the tools we found seemed to be geared to interactive use or at least human interpretation of their output. For our DNS-based remote configuration mechanism, we required a tool that could query the DNS for various resource record types (e.g, A, TXT, PTR) and return the resulting data in a simple non-verbose form. Thus we developed *dnsq*, a DNS Query tool that is based on the standard BSD resolver and uses Dave Shield's resparse library for parsing the DNS results.

7 CONCLUSIONS

This work has shown that FreeBSD is a very useful platform to develop custom networking solutions that offer at a very low cost a host of functionality that we could not otherwise implement using commercially available products even for prices 5 to 10 times higher. Self-designed network infrastructure equipment such as ours can be economically feasible even for such institutional users who do not consider themselves hardware systems developers. The effort it takes to acquire skills in designing custom FreeBSD-based solutions probably is not much higher than what it takes to successfully implement a commercial product: instead of wading through amounts of sales brochures hunting for the information to figure out if a commercial product will meet one's needs, instead of having endless phone calls to figure out the pricing options with sales representatives who cannot answer technical questions, instead of spending hours on waiting loops with customer service call centers to figure out that the commercial product has one little missing feature or bug that will fail the entire implementation or require awkward workarounds; that same time can be more productively spent on developing a custom solution with standard UNIX-based tools that can be made to work even if it does not initially work out of the box. The open source community around FreeBSD and KAME have provided excellent "customer support" to us: we received one critical kernel patches within a month and another one in just 24 hours, few questions remained unanswered on e-mail lists, and those that didn't get answered could eventually be answered by studying the source code.

We plan to contribute our development environment and tools back to the open source community once we have finalized our upcoming major version. The code will be available from http://aurora.regenstrief.org.

ACKNOWLEDGEMENTS

This work was performed at the Regenstrief Institute for Health Care and supported by the National Library of Medicine contract NLM-99-106/RMC.

REFERENCES

1 Weiner M, Schadow G, Lindbergh D, Warvel J, Abernathy G, Dexter P, McDonald CJ. Secure Internet videoconferencing for assessing acute medical problems in a nursing home facility. Proc AMIA Symp 2001. pp. 751-756.
2 International Telecommunication Union. Video codec for audiovisual services at p × 64 kbit/s [Recommendation H.261]. Geneva, 1990. The Union.
3 Touch J. Dynamic Internet overlay deployment and management using the X-Bone. In Proceedings of the 8th International Conference on Network Protocols, Osaka, Japan, November 14-17, 2000. pp. 56-68. Available from: http://www.isi.edu/touch/pubs/icnp2000.
4 Rekhter Y, Moskowitz G, Karrenberg D, de Groot GJ, Lear E. Address allocation for private Internets [RFC 1918]. Network Working Group, 1996.
5 Conoboy B, Fichtner E. IP filter based firewalls howto. Available from: http://www.obfuscation.org/ipf.
6 Kent S, Atkinson R. Security architecture for the Internet protocol [RFC 2401]. Network Working Group, 1998.
7 Kent S, Atkinson R. IP encapsulating security payload [RFC 2406]. Network Working Group, 1998.
8 Kent S, Atkinson R. IP authentication header [RFC 2402]. Network Working Group, 1998.
9 Maughan D, Schertler M, Schneider M, Turner J. Internet security association and key management protocol (ISAKMP) [RFC 2408]. Network Working Group, 1998.
10 Rushford JJ. Setting up a FreeBSD IPsec tunnel. FreeBSD Diary, June 2001. Available from: http://www.freebsddiary.org/ipsec-tunnel.php
11 Tiefenbach J. FreeBSD IPsec mini-howto. DaemonNews, 2001 Jan. Available from: http://www.daemonnews.org/200101/ipsec-howto.html
12 Touch J, Eggert L. Use of IPSEC transport mode for virtual private networks [Internet draft-touch-ipsec-vpn]. Nov 24, 2000.
13 Floyd S, Black DL, Ramakrishnan KK. IPsec interaction with ECN [Internet draft draft-ietf-ipsec-ecn-02.txt]. December 1999.
14 Housley R, Ford W, Polk W, Solo D. Internet X.509 public key infrastructure certificate and CRL profile [RFC 2459]. Network Working Group, 1999.
15 McCanne S, Jacobson V. The BSD packet filter: a new architecture for user-level packet capture. In Proceedings of the USENIX Winter Conference, San Diego, CA, January 25-29, 1993. pp. 259-270.
16 Salus PH. A quarter-century of Unix. Addison Wesley 1994.