Maximising Student Exposure to Unix Networking using FreeBSD Virtual Hosts

Grenville J. Armitage
Centre for Advanced Internet Architectures. Technical Report 030320A
Swinburne University of Technology
Melbourne, Australia
garmitage@swin.edu.au

Abstract- **A Remote Unix Lab Environment (RULE) is being developed at Swinburne University of Technology, allowing students access to networked unix hosts for their coursework and research projects. This paper describes our first generation solution using FreeBSD's "jail" functionality to emulate many FreeBSD hosts on a small handful of physical machines in a rack. Our primary constraint is to minimise the incremental infrastructure cost. The student front-end to the unix hosts will leverage pre-existing Windows-based PC labs scattered around campus and inter-connected by a 100Mbit/sec IP network. The FreeBSD hosts themselves are mini-ITX motherboards on a rack in a small room or closet, minimising their impact on scarce University lab space. This paper will describe our requirements, trade-offs, available tools, and how specific FreeBSD features are being utilized to create multiple virtual hosts on each physical machine. Our current implementation is based on FreeBSD 4.7.**

Keywords- Teaching, IP, Networking, FreeBSD, Unix, Virtual Hosts, Students

I. Introduction

Towards the end of 2002 our Telecommunications and Networking group faced the challenge of providing more hands-on IP networking experience for our students while working within the confines of a pre-existing, strongly Windows-centric environment. We already had special labs established for components of CCNA, CCNP, and MCSE certifications (a substantial investment in Cisco equipment and Microsoft Windows-based PC labs). Unfortunately this provided our students with a fairly specific experience in IP networking and IP client/server environments.

We also wanted our students to get their 'hands dirty' installing and using free, unix-based server, client, and middlebox applications. For example, we wanted to expose them to open-source web servers like Apache [1], alternative Windows file server such as Samba [2], DNS servers such as named, web crawlers/indexers, web proxies,... the list goes on. Not only would our students learn how to use these applications, they would be able to modify and rebuild the applications they were learning about.

In common with many small universities we work with less-than-ideal facilities and funding constraints. Our existing PC labs run Microsoft's Windows operating system, and are frequently booked solid for classes run by a variety of departments. Our plan was to avoid the additional cost (in time and salary) of re-imaging/rebooting machines between Windows and a unix system just for our IP networking classes. Building a dedicated unix lab (machines, desk space, seating) was considered an expensive last-resort.

Our solution is the Remote Unix Lab Environment (RULE). RULE provides multiple networked unix hosts, but does not require additional dedicated unix lab space. The existing campus PC labs are used as terminals through which students access their assigned RULE hosts. Because access is via the campus IP network, students can also engage in project work from home or from their laptops via our campus 802.11 network. RULE itself is housed in a regular 19 inch rack and tucked away in a corner of a small room, meeting our goal of minimal additional infrastructure cost (Figure 1).

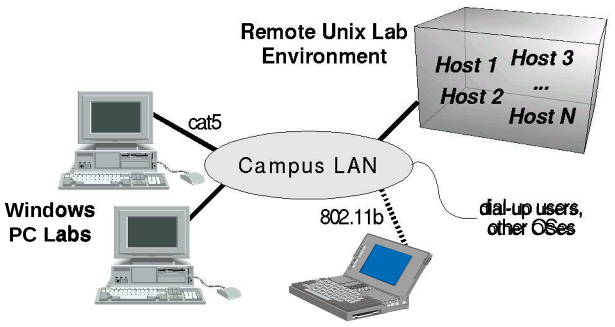


Figure 1 Remote Unix Lab Environment accessible from Windows machines around the campus network

The most interesting and critical part of RULE is our use of FreeBSD [3]. It is a robust, well-support and freely available implentation of unix (making it quite attractive from a recurring costs perspective). Most importantly, FreeBSD has kernel support, through the "jail" functionality, for instantiating multiple virtual unix hosts on a single PC motherboard. This multiples the number of students we can support with a limited set of physical hardware (or conversely, FreeBSD allows us to keep RULE small and hidden in the corner of a room). Our first generation of RULE is based on FreeBSD 4.7 (the version current in late 2002).

The rest of this paper describes the technological tradeoffs and solutions we are pursuing to implement our vision for RULE.

II. The Remote Unix Lab Environment

RULE needs to simultaneously meet the following goals:

- Provide students with self-directed access to open-source internet applications (e.g. clients, servers, and/or proxies) that they can compile, install, trial, and modify/recompile with minial supervision.
- Allows students to access RULE from anywhere on the campus intranet.
- Protect the rest of the university from student activities inside the RULE.
- Utilize off-the-shelf components and free software and minimise reliance on closed, commercial solutions.

We chose an open-source unix platform so users can build, run, modify, and rebuild many popular and useful networked applications without necessarily needing 'administrator' rights and/or commercial compilers and debuggers. A Microsoft Windows environment does not meet this goal. Any of the Linux distributions, FreeBSD, OpenBSD, or NetBSD would be suitable. We've chosen FreeBSD for two reasons. First, FreeBSD's clean 'packages' and 'ports' mechanisms (for installing applications in pre-compiled and compiled-as-needed forms) provides students with a number of ways to experiment with hundreds of common networked-applications. Second, FreeBSD's facilities for creating virtual hosts. (As a small bonus, many application binaries compiled under Linux also run directly under FreeBSD, and most such applications can be recompiled under FreeBSD if needed.)

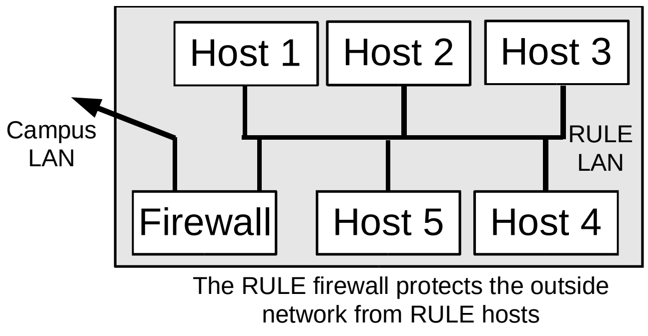

Host 1
Host 2
Host 3
Campus LAN
RULE LAN
Firewall
Host 5
Host 4
The RULE firewall protects the outside network from RULE hosts

Figure 2 RULE hosts are clustered behind a firewall to protect the outside world

Not surprisingly, RULE security is about protecting the campus network from RULE, rather than the other way around (Figure 2). The RULE firewall (another FreeBSD machine) allows external clients to initiate contact with applications (servers) inside RULE but not the other way around. For example, a student might deploy an Apache web server inside RULE and access it from their desktop or laptop. For special projects the firewall can be re-configured to allow out traffic originating from within RULE, but only if the destination is a host within the campus LAN (and excluding things like our campus web proxy to the outside world). The last thing we want are 'interesting' projects on RULE reaching out and annoying people around the Internet in an uncontrolled manner.

Secure Shell (ssh) is the remote access protocol for RULE hosts so that student's communication with their RULE host(s) are encrypted (including their initial username/password exchange). Most unixes have their own server and client implementations, and the OpenSSH consortium [4] has pointers to free and commercial implementations, including those for Microsoft Windows. PuTTY [5] is our preferred Windows SSH client, because it works well and the licensing conditions allow free use in our sort of environment. We also use Pscp, a companion to PuTTY, for secure file transfers between hosts using ssh. PuTTY requires minimal changes to the default software context of our Windows-based campus PC labs, and is easily installed by students who choose to use our campus 802.11b network from their personal laptops.

Ssh provides a security wrinkle known as 'port forwarding'. An ssh login from desktop or laptop to a RULE host can also be configured to provide one or more TCP-over-ssh tunnels from the RULE host to the rest of the campus network. Whether or not this is tolerable depends on ones goals. For now we impose on students the requirement of responsible use - port forwarding is a conscious act, and they will be traced if things go hay-wire. Port forwarding is also extermely useful for supporting X11 clients running on jail hosts. If the student runs an X11 server on their desktop or laptop, they can use ssh to automatically tunnel X11 sessions out from their RULE host to their local X11 server/display.

Given our limited budget, it is also important to build RULE out of common yet small components. We cannot afford to be 'bleeding edge' in our choice of hardware. Although RULE begins with a homogenous collection of hardware, over time the initial motherboards will become unavailable and incremental repairs will result in a heterogenous collection of hardware. FreeBSD runs on a range of x86-based hardware, from old 486-based machines up to the latest Pentium 4s, ensuring that RULE will survive motherboard upgrades and changes.

We have built the first version of RULE around VIA Technologies' EDEN embedded system processor (ESP) series, specifically the ESP 5000 released in 2002 [6]. This low-power motherboard comes in a mini-ITX form factor (170mm x 170mm), has an embedded fanless 500MHz Celeron-equivalent processor, can support up to 1GB of PC133 SDRAM, onboard 10/100 Mbit/sec ethernet interface, onboard COM, PS/2, USB, Printer, VGA, and sound ports, two ATA100/66 IDE sockets, and takes standard ATX power. They are also quite cheap (around $200AUD at the end of 2002). Adding a power supply, RAM, and hard-drive was enough for a running system. A CDROM drive is temporarily attached to the second IDE port when installing FreeBSD.

The small form factor allows us to pack a number of these devices into limited rack space. While the video, audio, printer and PS/2 interfaces are unnecessary for RULE applications, we make use of the serial ports for console server access to each motherboard, and the USB ports are used to power small “Alloy NC-05c” Ethernet hubs [7] that form part of the RULE's internal network (minimising the wiring and power supply complexity within the rack, Figure 3). Our first RULE implemention uses one ATX supply per ESP5000, but we hope to run two or more motherboards from a single ATX supply in the future to further minimise space requirements.

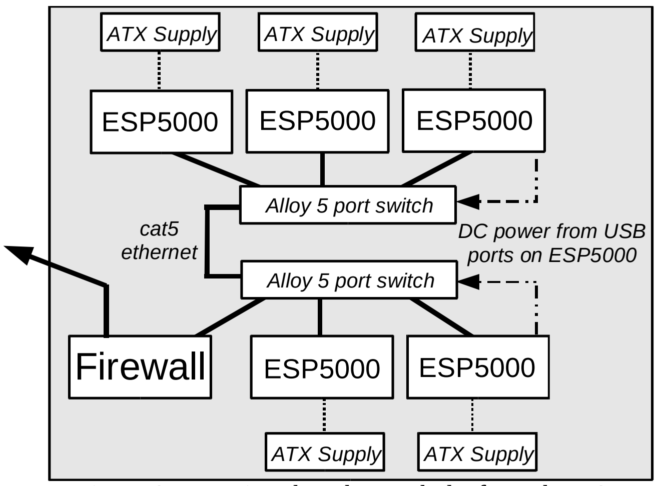


Figure 3 Powering the ethernet hubs from the USB ports of RULE hosts simplifies wiring

III. Virtual hosts using FreeBSD “jail”

Virtual FreeBSD hosts are a central part of RULE. We are developing a Jail Host Toolkit (JHT) to simplify the establishment and management of multiple virtual hosts on a single physical motherboard running FreeBSD 4.7. Each virtual host has its own IP address and can run separate instances of most user-space applications. Each JHT virtual host can have its own user accounts and passwords. If needed, users can even be given 'root' (administrative user) access inside their own virtual host without compromising any other virtual host. Once a JHT virtual host has been configured and booted, it appears just like a regular, IP-accessible FreeBSD host.

Because JHT virtual hosts are implemented using the FreeBSD kernel's `jail(8)` functionality, we refer to them as *jail hosts* and the machine in which jail hosts reside as the *primary host*. Jail hosts are replicas of the FreeBSD user space environment. Each VIA Technologies ESP 5000 motherboard is a primary host.

Students are only given access to jail hosts.

What does a jail host look like?

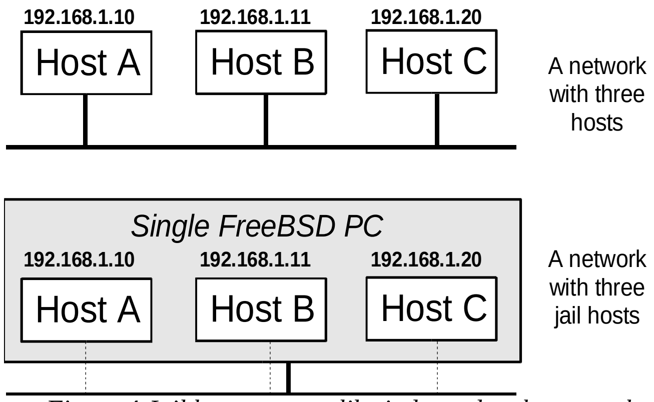


Figure 4 Jail hosts appear like independent hosts on the network

Figure 4 attempts to convey the notion that, from the perspective of other people on an IP network, jail hosts look very similar to regular FreeBSD hosts. If there existed three hosts on the 192.168.1 network that could be ping'ed and otherwise accessed over the network, three jail hosts on a single FreeBSD-based PC would approximate the same behavior.

A jail host runs its own instances of ssh and other remote login daemons, has its own password files, and has its own copy of the FreeBSD filesystem independent of the other jail hosts sharing the underlying PC. A jail host supports all conventional, kernel-mediated TCP and UDP based communication. User-space applications you can run on a regular FreeBSD machine will generally run unchanged inside an equivalent jail host.

However, a jail host does not completely replicate the environment of a regular FreeBSD host. The limitations primarily relate to the jail host's networking and kernel functionality.

- Networking
  - A jail host has a single network interface and a single IP address (no multi-homing, and no local routing table access)
  - The localhost address of 127.0.0.1 is silently mapped to the jail's real IP address
  - Sockets bound to a wildcard address are actually bound to the jail's real IP address
  - A jail host cannot get raw access to the network interface (e.g. for network sniffing with tcpdump, building custom UDP frames for traceroute, sending/receiving ICMP packets for ping, etc....)
  - Sockets cannot be bound to non-IPv4 protocols (up to and including FreeBSD 4.7 there is no jail support for IPv6)
- Kernel
  - File systems cannot be mounted or unmounted from within the jail host
  - Special devices and/or loadable kernel modules cannot be added from within a jail host
  - Kernel system variables cannot be modified from within a jail host
  - Access to physical devices is seriously constrained
  - Access to System V IPC primitives are blocked by default (because their namespace is common to all processes, potentially allowing jail host processes to interfere with each other and primary host processes).

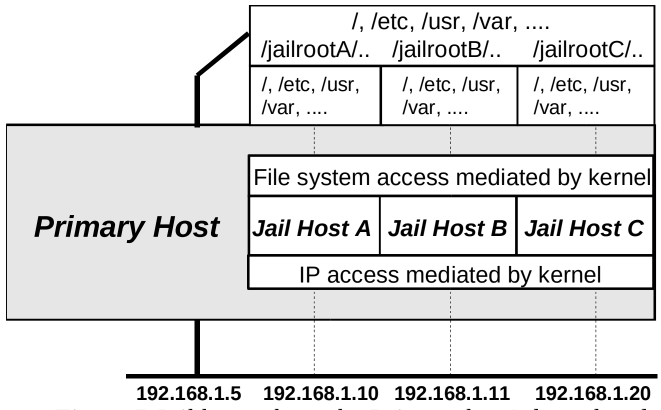
A jail host provides the usual files and directory structures (`/`, `/etc`, `/usr`, ...) and will happily support applications that can live within the constraints listed above. For things like software development (e.g. compilers, IDEs, etc..) or servers (e.g. web, ftp, DNS, Samba, etc...) these constraints shouldn't generally be a problem. Some applications (e.g. X11 authentication) need tweaking to handle the fact that localhost (traditionally 127.0.0.1) is silently mapped to the jail host's actual IP address.

The primary host and its jail hosts

It is important to recognize that jail hosts are not virtual machines. All jail hosts and their processes run in a common process space of a single FreeBSD kernel. This single instance of a FreeBSD kernel mediates access to shared resources- primarily disk, network, and general I/O ports.

Figure 5 shows the relationship between jail hosts and their primary host. The primary host is the FreeBSD kernel and user space that governs and controls the environment within which each jail host operates.

Jail host processes are regular processes inside the primary host's process space, additionally marked as 'jails' defined by the jail's assigned IP address and the path to the jail's private file space. Processes belonging to a particular jail host (i.e. forked from another process inside the jail host) inherit the jail's restricted context (IP address and private file space). Filesystem accesses by jail host processes are silently re-interpreted by the kernel relative to the jail's private file space. Network access is limited by the kernel to connections relating to the jail host's specific IP address.



/, /etc, /usr, /var, ...
/jailrootA/..
/jailrootB/..
/jailrootC/..
/, /etc, /usr, /var, ...
/, /etc, /usr, /var, ...
/, /etc, /usr, /var, ...
File system access mediated by kernel
Primary Host
Jail Host A
Jail Host B
Jail Host C
IP access mediated by kernel
192.168.1.5
192.168.1.10
192.168.1.11
192.168.1.20

Figure 5 Jail hosts share the Primary host's kernel and resources

The primary host is assigned multiple IP addresses, one for itself and an aliases for each jail host. Jail host IP addresses must belong to one of the subnets to which the primary host's IP address(es) belong.

Consider Figure 5. The primary host's IP address is 192.168.1.5, and it has aliases on the underlying network interface for 192.168.1.10, 192.168.1.11, and 192.168.1.20. The latter three IP addreses are assigned to jail hosts A, B, and C. The primary host has also allocated three directories as the root of each jail host's filesystems, `/jailrootA`, `/jailrootB`, and `/jailrootC`. Each jail host's own FreeBSD filesystem is replicated under their assigned root directory.

For example, when a process inside jail host A tries to access `/etc/hosts` it will actually be accessing `/jailrootA/etc/hosts`. Likewise, a process in jail host C accessing `/usr/local/etc/` will actually be accessing `/jailrootC/usr/local/etc`. The jail host processes are totally unaware of the deception.

Another example, if a process in jail host A tries to `bind()` a UDP socket to listen on the wildcard IP address '*' at port 9000 (*:9000) the actual socket will be bound to listen on 192.168.1.10:9000. A similar wildcard bind() on jail host B would have its socket bound to listen on 192.168.1.11:9000.

These restrictions are enforced by the primary host's kernel to ensure jail host processes never see (and cannot access) processes and files outside their constrained context.

Primary host processes have no such restrictions (aside from the usual restrictions pertaining to user and group ID). They can access the entire file system (including the sub-directories assigned to each jail host), bind to TCP and UDP sockets with any valid IP address associated with the primary host (including those assigned to jail hosts), and are allowed raw socket access to the underlying network interface(s). Kernel run-time modifications (loading/unloading modules), creation of special devices, mounting/unmounting filesystems, etc, are all performed while logged into the primary host.

Resource management between jail hosts is not entirely complete. The kernel does not provide any specific mechanisms for preventing one jail from starving other jails of shared diskspace, CPU time, or network bandwidth. In RULE we enforce disk sharing by placing each jail host's file system within its own disk partition. However, CPU time cannot (currently) be similarly protected. Jail host processes compete for CPU time with each other, just like any other process inside the primary host. Network resources can be managed by using `ipfw` and `dummynet` to mediate link bandwidth consumption to each jail host's IP address.

IV. Jail Host Toolkit

Jail Host Toolkit (JHT) is a set of scripts for building a jail host (`makejail`), preconfiguring a jail host with various software packages (`newjail`), booting/killing a jail host (`bootjail` and `killjail`), and re-starting jail hosts whenever the primary host is rebooted.

Each of our ESP5000 primary hosts has 512MB of RAM and an 80GB IDE drive, and supports four jail hosts (in this configuration Figure 3 instantiates 20 distinct FreeBSD hosts for student use). We allocate 16GB of disk space for each jail host and the primary host. Because JHT assigns each jail host to its own FreeBSD partition a typical disk layout would look like:

- `ad0s1    16GB primary host`
- `ad0s2a   16GB jail host 1`
- `ad0s2e   16GB jail host 2`
- `ad0s2f   16GB jail host 3`
- `ad0s2g   16GB jail host 4`

Slice 1 (the first BIOS partition) is used for the primary host's own FreeBSD installation. Slice 2 is given the majority of the disk and further subdivided into four FreeBSD partitions, one for each jail host.

JHT's `makejail` looks after compiling and installing a clean, user-space FreeBSD tree onto each jail host's disk partition, and mounting this partition into the primary hosts filesystem. (A default FreeBSD installation takes up around 130MB of the 16GB available to each jail host.)

In practice most interesting student projects can be achieved inside jail hosts with 4GB or less of disk space, making it quite feasible to build primary hosts out of machines having 20GB or smaller drives.

Jail hosts are 'booted' from within the primary host by causing the `jail(8)` command to execute an instance of `/etc/rc` and associating this instance with the jail host's IP address and allocated part of the primary host's filesystem.

For example, to start a jail host whose filesystem ad0s2a is mounted on /home/jail1, give it the hostname 'jail1' and IP address 192.168.50.1, we would use:

- `/usr/sbin/jail /home/jail1 jail1 192.168.50.1 /bin/sh /etc/rc`

JHT's `bootjail` makes sure the jail host's disk partition is mounted, and that an appropriate IP address is aliased to the primary host's ethernet interface before the jail host's /etc/rc is started. /etc/rc then spawns the usual FreeBSD daemons and startup processes, unaware that they are confined to the jail's context (in this example, the file system under /home/jail1 and an IP identity of 192.168.50.1). JHT's `killjail` handles the reverse task, running the /etc/shutdown script inside a jail host before killing the jail host's processes.

V. Disk-friendly Access to FreeBSD Packages

Once a jail host is running people can login into it (using ssh) from anywhere on the campus network. They can pull in new software packages using ftp, fetch, wget or similar commands. FreeBSD's “pkg_add -r” can be used to download and install precompiled FreeBSD packages over the Internet. However, there's no way for users (even the root accounts) to mount CDROMs (or any other file systems) from inside their jail host. On the face of it this makes it hard for us to give students access to local copies of the 5000+ packages freely available on the FreeBSD CDROM set.

Our solution has two parts. We NFS-export copies of the FreeBSD CDROMs from the primary host, and then NFS mount the exported CDROM filesystems multiple times back into the primary host's file space such that one copy appears in each jail host's filesystem. The nett effect of these loop-back mounts is that each jail host sees the entire FreeBSD CDROM set in their filesystem, but no additional hard disk space is consumed. By exporting the filesystems “read-only” not even the root user in each jail can modify the files in what appears to be 'their' copy of the FreeBSD CDROMs.

For example, assume we have two jail hosts (jail1 and jail2) on the primary host, and the primary host is exporting the first FreeBSD 4.7 CDROM as /mnt/freebsd47. The primary host is running as both an NFS server and NFS client. We want each jail host to see this CDROM at /mnt/freebsd47 in their own filesystems. The solution is:

- NFS Export /mnt/freebsd47 read-only
- `mount localhost:/mnt/freebsd47 /home/jail1/mnt/freebsd47`
- `mount localhost:/mnt/freebsd47 /home/jail2/mnt/freebsd47`

Users can now access FreeBSD packages under `/mnt/freebsd47/packages/All` in their own jail host's context.

This technique allows RULE to be completely self-contained, enabling students to access the packages collections without needing to mount or unmount physical CDROMs. (Note that the primary host does not need physical CDROMs either - we use the 'vn' vnode driver to create exportable filesystems from binary ISO images of the CDROMs.)

JHT's `newjail` uses these loop-back mounts to pre-install a range of useful tools (e.g. X11 clients and libraries, acrobat reader,) from the FreeBSD packages collection when initializing a jail host recently created with `makejail`.

VI. What Can Students Actually Do?

Jail hosts do impose practical limitations on students, both in what they can do and what they can damage.

Students can compile and run any program that accesses the network using unix `socket()` facilities for conventional TCP or UDP communication. This includes almost all client/server scenarios (such as http, ftp, DNS, X11, and similar). Students can run graphical code development environments (such as Kdevelop [8]) on their jail hosts if your RULE firewall allows X11 clients to access external X11 servers.

However, students cannot run programs that require raw access to the underlying IP or Ethernet layers. This precludes the use of some common commands such as `ping` and `traceroute`. Students cannot access or modify the routing tables of the underlying primary host, nor can they rebuild the kernel, or modify the running kernel's state (even when logged in as root within their jail host). For course and project work focusing on network applications these limitations can usually be tolerated

On the upside, students cannot irretrievably mangle a jail host. RULE administrators can make regular `tar`-file backups of each jail host's disk partition, and restoration of a jail host becomes as simple as un-tarring the tar-file back onto the jail host's disk partition. Students can be given hands-on experience of having root access within their jail host (managing group and user accounts, access levels, system daemons such as inetd, etc) and yet suffer only minor inconvenience if they accidentally mangle their jail host's filesystem. The primary host remains isolated - root processes within each jail cannot reach out and modify the primary host environment.

Being able to tar/un-tar entire jail hosts also simplifies the administrative task of sharing RULE between classes at different times of the week. Each primary host can automatically switch around jail host filesystems according to class schedules, rather than requiring a staff member to manually re-image physical machines.

VII. Open issues and Future Directions

RULE is in its early days, and we still have a lot to learn and try. Resource management within primary hosts is still somewhat crude and jail hosts do not allow students full control of the host's networking functionality.

Resource management schemes must be resilient against attacks from students who have root access inside their jail hosts. Assigning distinct primary host disk partitions allows us to insulate each jail host from disk-hungry (or buggy) applications running on other jail hosts. However, CPU time, memory space, swap space and network interface bandwidth is still shared across jail hosts. CPU time is not considered a critical factor - jail host processes share the CPU just like regular primary host processes, and any student projects that need high performance computing probably shouldn't be on the RULE anyway.

JHT will use ipfw (a FreeBSD firewall module) and dummynet (FreeBSD's network bandwidth controller) in the primary host's kernel to enforce bandwidth limits on the IP traffic flowing in and out of specific jail hosts. (For example, we might limit each jail host to 1Mbit/sec in each direction - sufficient for most student projects while ensuring no one jail host can starve the others of network access.)

FreeBSD allows per-user resource consumption limits to be specified through an `/etc/login.conf` file in each jail host, but there's the challenge of making this file unalterable by root users inside each jail. FreeBSD's concept of four 'secure levels' may help here. We can write the /etc/login.conf files from the primary host level, mark them as *immutable*, and run the kernel at securelevel 1 (which disallows subsequent attempts to modify immutable files, even by root). Another approach would be to NFS mount read-only copies of /etc into each jail host (although this would make it impossible for the student's to modify other useful files such as /etc/inetd.conf, etc). This is still an open question.

We also plan to increase the number of primary hosts so that students can have their own dedicated FreeBSD machines in RULE (enabling raw packet access to the network interface and the ability to control or rebuild the kernels they run).. Small clusters of three primary hosts per student project group will enable quite flexible IP networking (routing and traffic analysis) experiments. We are evaluating solutions for remote power-cycling/cold-rebooting of individual primary hosts (for those times when students completely jam their machines) and a FreeBSD-based console server so that students can watch their machines rebooting after making kernel modifications. Unfortunately, giving students complete access to a primary host guarantees that they will (whether by accident or design) scribble all over the hard drive at some point. We are still considering options for remotely restoring a primary host to pristine, pre-student condition. Ideally the solution will not require staff to physically access the RULE rack space (so as not to limit the times of day or week that RULE can be made available to students).

JHT and RULE is also being used as the basis for an 802.11b/IP mobility testbed. FreeBSD jail hosts can be instantiated over any of the primary host's IP interfaces, including wireless devices, IP-IP tunnels, and VLAN devices. We have already created jail hosts over an 802.11b-enabled primary host (using a Ricoh PCMCIA-PCI bridge and Lucent 802.11b card on an ESP5000 motherboard) and hope to publish a description of this testbed later in 2003.

Finally, there are always going to be quirks in mating the operating system and compact motherboards, especially as RULE evolves and becomes heterogenous. For example, the VIA ESP5000's in-built ethernet device relies on FreeBSD's 'vr' driver. We have already discovered a bug in the vr driver triggered by repeated "`ifconfig alias`" commands. The bug-fix is being folded back into FreeBSD 4.7's source tree. Aside from this, the ESP5000s are proving quite useful and convenient.

VIII. Conclusions

In order to provide increased student access to unix development environments at our Windows-centric university we have developed a FreeBSD-based Remote Unix Lab Environment (RULE). RULE hosts are remotely accessed using secure shell (ssh) from existing Windows-based PC labs around campus, leveraging an existing institutional investment in physical infrastructure and minimising our group's incremental costs. Students are also able to access RULE hosts from wirelessly equipped laptops and dial-up hosts (whether running Windows or some other operating system), which provides flexible opportunities for them to do their work.

To further reduce costs, we utilize FreeBSD's `jail` functionality to implement *jail hosts* - virtual RULE hosts that provide each student their own FreeBSD user-space environment to manage and explore. A set of scripts, our Jail Host Toolkit (JHT), is being developed to automate many aspects of jail host creation and management. Our initial implementation of RULE uses 5 mini-ITX ESP5000 motherboards from VIA Technologies to create 20 RULE hosts. RULE and JHT will evolve to include more physical hosts, better resource management between jail hosts, and the ability for students to control entire physical hosts for certain classes of educational IP networking experiments.

References

[1] "The Apache Software Foundation," http://www.apache.org, Dec 2002

[2] "Samba," http://www.samba.org, Dec 2002

[3] "FreeBSD," http://www.freebsd.org, Dec 2002

[4] "OpenSSH," http://www.openssh.org, Dec 2002

[5] Simon Tatham, "PuTTY: A Free Win32 Telnet/SSH Client," http://www.chiark.greenend.org.uk/~sgtatham/putty/, December 2002

[6] "VIA Technologies, Inc.," http://www.viavpsd.com

[7] "NS-05c 5 port NWAY 10/100Mbps mini switch," http://www.alloy.com.au/products/ns05c.htm, Dec 2002

[8] "KDevelop," http://www.kdevelop.org