ISI-TR-2003-579

NetFS: Networking through the File System

Joshua Train, Joseph D. Touch, Lars Eggert, Yu-Shun Wang
{train, touch, larse, yushunwa}@isi.edu
USC Information Sciences Institute[◊]

Abstract

NetFS provides a platform-independent file system interface to the network stack of operating systems. It unifies and integrates different existing APIs that each control parts of the network stack. NetFS offers a common, familiar interface that supports both existing functionality, as well as new capabilities for fine-grained access control, user-based virtual views, and remote access. A subset of NetFS was implemented using Perl and named pipes. The system is currently being implemented as FreeBSD 5.0 kernel extension.

1 Introduction

As operating systems and networks have evolved, the interface to the networking components has become increasingly complex. Most UNIX flavor OS's have separate commands to configure network interfaces, set routes, and view current connections, and require further cryptic configuration via unfamiliar interfaces directly accessing OS structures, namely *ioctls sockopts*, and *sysctls*. Having this variety of network configuration interfaces is confusing and makes it difficult to manage access control. NetFS replaces this complex constellation of interfaces with a single API based on a familiar file system paradigm which further provides fine-grained access control and process-specific views.

User level configuration commands are confusing because they lack a common API. `ifconfig` configures interfaces, the `route` command adds routes, and the `netstat -r` command shows routes – with little in common. Using different commands requires consulting a plethora of `man` pages, and the variety of syntaxes and configuration options can quickly overwhelm. The usage of each command also varies between OSs. For example, the LINUX `ifconfig` command has a very different syntax than BSD or Solaris `ifconfig` [3] [5] [8]. Many networks consist of systems with a variety of OSs, thus the task of configuring networks can be increasingly complex.

The variety of methods, such as *ioctls*, *sockopts*, and *sysctls*, used to communicate the configuration information back to the kernel is complex both because of the variety of parameters and their lack of organization. Programmers often have difficulty determining which method is required to configure a particular parameter (Figure 1). In some cases, multiple interfaces are required for complete access to configuration control, *e.g.*, TCP connections can require configuration *via* the top four APIs shown. In other cases, there are competing alternative methods, for example, routing table entries can be configured by *ioctls,* the *routing socket* in-band API, as well as by the command line *route* and *netstat* commands.

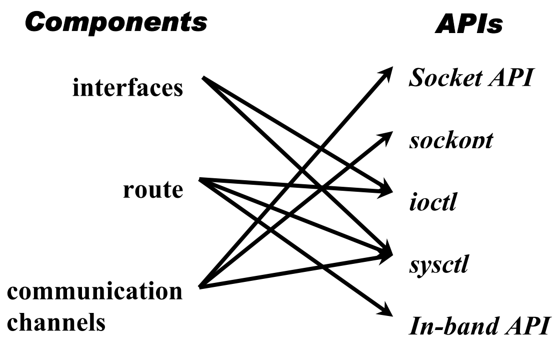

Figure 1 Intertwined network control

◊ This material is based upon work supported by the National Science Foundation under Grant No. <NETFS: ANI-0129689> . Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

There is further no simple way to provide fine-grained access control to network configuration. Network configuration commands use system calls that require the user process to be executing in root mode. The ability of the user to configure the network boils down to whether the user has root access or not.

On systems where multiple processes must configure the network (*e.g.*, X-bone [9]), different processes need to be granted different configuration permissions, yet blindly giving root-level access poses risks to the system's integrity. One process may need to configure the address of a virtual interface, but should be prevented from configuring other interfaces or making other network configuration changes. Granting processes root access for the sake of configuring one component of the configuration is like giving away a combination to a bank vault; the entire vault is now open. It would be preferable to give compartmentalized, fine-grained access, akin to the keys to a single safe-deposit box.

Previous attempts have tried to overcome these challenges to network configuration, but have not gained wide acceptance due to their complexity of design and the addition of yet another unfamiliar API. NetFS uses the file system as an interface to network configuration, providing a familiar API and capitalizing on the fine-grained access control it already provides.

2 File System Interface

One of the key components of general purpose OSs is an organized system with which to read and write stored data, namely the file system. File access methods have provided the canonical API; it is thus useful to consider how they can be used for more than storing static data, *e.g.*, for network configuration.

Users are familiar with the file system API. One of the most elementary tasks of computing is reading and writing files; it is one of the first APIs taught, and one of the most ubiquitous. File operations are just data reads and writes based on object names; this process can likewise be used for network configuration.

Users are also familiar with the organizational structure a file system provides. Directory hierarchies and aliases are powerful tools for organizing files, and can be applied to configuration data as well.

2.1 Uniform Interface

Networking has introduced a wide variety of different objects such as sockets, routes, and interfaces. These objects have few similarities, but the file system can provide a common interface for identification and access.

2.1.1 Order Through Subdivision

File systems allow users to subdivide different objects by types and instances. The "documents" directory usually holds documents, and a "pictures" directory typically holds pictures. Although pictures are very different than documents, they are organized, handled, and accessed in the same way.

As illustrated in Figure 2, NetFS uses this same methodology to organize the different networking components of the operating system.

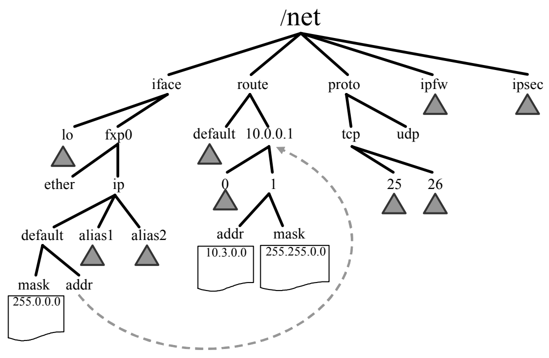

Figure 2 NetFS directory structure

The */net* directory contains directories that describe networking components such as *iface*, *route*, *socket*, *etc.* Inside each of these general category directories exist directories and files that give each object further context. For example, the */net/iface* directory includes directories named *l0* (loopback) and *fxp0* (an Ethernet interface), which are the names of typical interfaces on a FreeBSD machine. The */net/route* has directories named with network addresses such as *10.0.0.0* use to describe the network to which the route leads. The directory tree is a series of context descriptors that eventually leads to the object that it describes.

The directory structure provides a context by which to locate an object, and the contents of that directory consists of files and directories that describe that particular object. For example, the */net/iface/xl0* directory may contain files such as: *media* – the description of the interface's physical media, and *status* – indicating the current availability of the interface.

A folder may contain sub-directories, such as shown for the *address* directory. Folders are used to separate addresses by family, each requiring a subdirectory in which to store multiple addresses.

The */net* directory structure provides a uniform organization and easily-browsed structure to the set of network configuration parameters. All configurable parameters of an interface are subfolders or files under that interface's directory in */net/iface*. Routes are in the */net/route* directory. Locating configuration parameters is thus less mystical.

2.1.2 Unified Access Method

Not only does the file system provide organizational structure, but it also provides a standard API: the VNODE interface [7]. This interface provides, among other things, a standard way the user can create directories, read directory contents, write files, and read files. Regardless of the information system that is represented by the file system, be it a hard drive, memory-mapped, or a server across the network; because of this standard interface, the user can access this information in the same way. Looking beyond the scope of hard drives and memory, this interface provides a standard way to map operations performed on files to respective functions that are defined by the programmer. NetFS uses this methodology to use files to tie into a series of functions that perform networking tasks.

Using the file system, network properties can be determined by reading the corresponding file. The subnet mask assigned to an interface can be shown by typing[1]:

```
?] cat /net/iface/xl0/subnetmask
```

[1] Throughout, input is shown after the "?]" prompt (wrapped and indented as needed); output is shown without a prompt, on the following line.

The output would be:

```
255.255.255.0
```

Similarly, the MTU of a route is shown by:

```
?] cat /net/route/127.0.0.1/MTU

1500
```

The subnet mask can be changed as follows:

```
?] echo "255.255.240.0" >>
   /net/iface/xl0/subnetmask
```

One common network configuration task is to view what objects are available. Through the file system, this task is done by simply reading the contents of a directory. NetFS's *ls /net/iface* command is much simpler and more obvious than the current *ifconfig –a*. directory. Similarly, routes can be shown by *ls /net/route*.

For example:

```
?] ls /net/iface

xl0 fxp0 fxp1
```

Creating a new instance of an object, such as adding a new route or virtual interface, is achieved by creating the appropriate subdirectory. Consider the following command, which is the first step in creating a route to the 192.168.0.0 network:

```
?] mkdir /net/route/192.168.0.0
```

The next step would be to create a file *16* in that directory, and the contents of that file would indicate the next-hop router address or the outgoing interface (if a LAN).

An advantage to using a file system for configuration and control of an OS networking stack is portability. Programs that modify network parameters require only a common hierarchical file system, which is highly portable. The remainder of the functionality, including OS-dependent operations, are implemented underneath, inside the NetFS implementation on a specific OS.

There are two obvious considerations – choice of the directory structure and consistency. First, the file system does not impose an *a-priori* structure to the network configuration parameters; NetFS provides that structure as part of its API specification. There is substantial leeway in determining a useful API, and the examples shown here are not final. It is expected that the interface will evolve with experience. Note that using the file system provides a unique opportunity to retain past APIs; the alternate views are just aliases or symlinks of the current API.

Consistency checks are part of the NetFS system. If the name passed to the *mkdir* command did not make sense in the directory that it was being created in, it would be rejected. For example, the following command would result in an error if there were no interface called *timbuktoo*:

```
?] mkdir /net/iface/timbuktoo

error: no device "timbuktoo"
```

NetFS provides for a framework where all configuration tasks can be performed in a standard fashion. Configuring an interface or a route is functionally equivalent. Each different component may involve different features, but the method to configure those features is the same.

2.2 Fine Grained Access Control

Current operating systems do not provide a convenient mechanism for partitioning the configuration of different networking components. Typically a user must have root access to perform network configuration tasks such as setting an IP address or configuring a route. On systems such as the X-Bone [9] and other multi-user systems, different users need to have specific configuration permissions. However, current configuration commands do not provide permission control.

By configuring networking components through the file system, access control is as simple as setting the permissions on files and directories. To configure all interface cards, a user would be given write permission in the entire */net/iface* directory. Limiting a user to configuring a particular network interface, *e.g.*, xl0, involves giving write access to only */net/iface/xl0*.

Unix file systems provide separate owner, group, and global permissions. This coarse control can be further augmented with Access Control Lists (ACLs), providing fine-grained access [10].

2.3 File System Features

Along with providing a standard API and access control, NetFS uses the file system's existing support for aggregation and virtualization to provide augmented network configuration capabilities.

2.3.1 Remote Networking Configuration

File systems can be remotely mounted, *e.g.*, *via* NFS [7]. By using remote access, NetFS seamlessly supports remote network configuration and monitoring, again using the same file system API. All operations, except those affecting the remote mount, can be just as easily performed remotely.

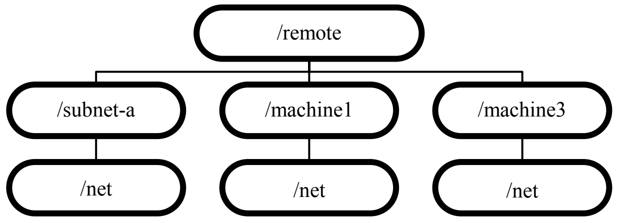

Figure 3 Remote view of /net

Consider the directory structure shown in Figure 3. This directory structure is constructed by remote mounting the */net* directory local to each of the remote machines to be configured, or even aggregating them *via* a combination of mounts and symlinks (*e.g.*, *subnet-a*). Remote file system access control replaces the need for separate login procedures.

As with all remote configuration of networking components, the remote administrator must be careful not to lock himself out by changing the network information that is used to provide the remote mount capability. As with consistency control, this can be managed by the NetFS system on the remote machine, such that entries used by the mount protocol are locked to remote users.

2.3.2 Virtualization

File systems also allow per-process customization. Different parts of */net* could look and behave differently for different processes, much as "~/" is interpreted local to a login. One user or process would use the following command:

```
?] ls /net/iface

lo xl0
```

Another user may see:

```
?] ls /net/iface

lo em0
```

The root user would see the entire set of interfaces:

```
?] ls /net/iface

lo xl0 em0
```

The file system not only virtualizes the hierarchy, but it can be extended to virtualize functionality. For example one user may see:

```
?] cat /net/route/default/gateway

192.168.1.1
```

Another user may see:

```
?] cat /net/route/default/gateway

10.10.10.1
```

In this example the */net/route/default/gateway* files is the same file but points to two completely different routing entries. This would be useful if the first user could attach (open sockets on) only the 192.168.0.0/16 interfaces, thus his default route would be 192.168.1.1. The other user sees a different default route because his context – the interfaces he can attach to, *etc.* – differs.

The ability to virtualize file system views and functions provides yet another opportunity in customizing the information that is presented to different users. Other virtual file systems such as Procfs [2] use this feature to simplify how processes check their own state. */proc/current* contains information particular to the process that reads it. Processes can monitor their state without needing to know their process identifier.

These file systems virtualization abilities provide a unique opportunity to further the process of virtualizing network connections and configuration.

2.4 Costs of Using File System API

There is essentially no performance cost in accomplishing network configuration through a file system. Previous virtual file systems such as Procfs [2] claim no slow-down *vs.* using conventional command-line APIs. Because network configuration operations are not a high-performance task anyway, this is somewhat moot.

Note that NetFS is not an exclusive API. Previous APIs and commands can be supported, either underneath or on top of NetFS. This supports gradual migration to the new API, backward compatibility, and incremental upgrades of commands and control programs.

3 Implementation Issues

As useful as a file system API has been shown to be, it presents implementation challenges. Issues such as directory organization, internal consistency, and functional atomicity need to be addressed. Two implementation methods are examined: the first, using Perl and named pipes, has been completed and exposed some of these issues; the second, involving a loadable kernel module (FreeBSD KLM) is under design. The KLM approach is more flexible and powerful, though it is more complicated to implement.

3.1 Hierarchy

File systems offer the ability to organize information hierarchically. This organization is necessary to simplify the network interface, yet organizing the components is not a trivial process and presents design challenges.

As noted earlier, a variety of hierarchies can be considered, depending on the level of detail desired. For example, in organizing network interface devices, BSD machines use a 2-5 letter device type identifier followed by a number identifying the instance of card, *e.g.*, xl0, xl1, fxp0, em0, em1, *etc.* The hierarchy can be configured in a variety of ways, two of which are shown in Figure 4 and Figure 5.

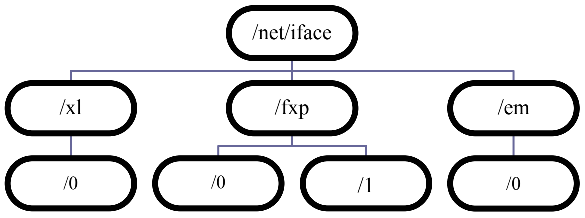

Figure 4 Hierarchy by inteface class

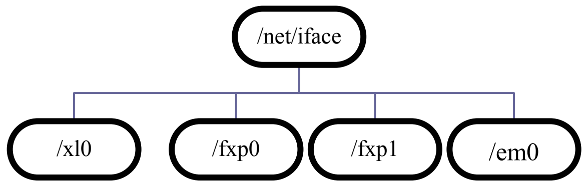

Figure 5 Hierarchy by interface

Figure 4 allows permissions to be given based on interface class, whereas Figure 5 requires per-interface permissions. Conversely, Figure 5 provides less nested directories and appears less complex. As noted earlier, both can be provided using symbolic links or aliases; the number of different hierarchies and specific hierarchies supported are part of the specification of the NetFS API.

One of the NetFS's primary goals is to simplify the interface to the networking components, yet "simplifying" can be context-dependent. Debugging kernel network data structures would benefit from direct access in NetFS, but that further complicates the directory structure for users not doing debugging. The primary hierarchy should be as complete as possible, but should consider frequency of access and utility in the organization.

NetFS makes use of the features of the file system to solve some of these issues. The root user will be different than the view that a standard user has. The different views that can be established for every user can be solved by setting up policies as to different groups need to see different information.

3.2 Internal Consistency

As noted earlier, there are consistency checks that are already part of the current network component configuration interface. When an interface is added, its corresponding default is also added; when the interface is removed, the route is removed as well. When these operations are translated into file system operations, they need similar consistency checks.

Suppose a user has permissions to change an interface address, but does not have permissions to create a new route *per se*. The system could either allow the address change and implicitly permit the route, or prohibit the address change exactly because the route change is prohibited. It is to NetFS's credit that this example is easily considered and alternatives implemented, depending on policy. In either case, the constancy checks occur during the creation of files or directories, just as the file system prevents the creation of files with illegal names or removes hard links when primary files disappear.

Under current APIs these issues do not arise, because root access is required for any modification, and thus any corollary operation would be permitted. If the user were changing an interface address, he had root access which further gave him the ability to change a route. However, by having permissions on the different components there is now a choice of what operations to allow, and these are the type of issues that arise when designing NetFS.

These component interrelationships bring about the need for new policies. The needs of one system may differ from the needs of a different system; therefore user defined policies are needed to determine the desired method to handle conflicts like these.

3.3 Atomicity

Discussions about file systems often bring about with them concerns about the data integrity and functional atomicity. NetFS uses existing file locking mechanisms and read-through properties to ensure the integrity of the data represented. Other issues of atomicity are relevant, however. Notably, when internal consistency mechanisms require new subordinate files and directories, the entire set must appear as an atomic operation.

Further, locking must be considered. As with any set of resources, unconstrained locking can result in mutual starvation; this can be avoided with ordered locks. Additional timeout mechanisms can be used to ensure that processes can not hold configuration information and starve other processes attempting to change that configuration.

Because there are multiple APIs to change network configuration, the information represented through the file system must accurately represent the current kernel configuration. Because NetFS files do not hold any static information in of themselves, but instead serve as windows to the underlying information represented in the kernel, synchronization is ensured.

4 Implementation Experience

The implementation of NetFS is proceeding in two stages. The first stage working prototype of NetFS uses Perl and named pipes. The second stage uses a loadable kernel module (KLM, on FreeBSD). The initial development was performed on FreeBSD 4.7 and the KLM is being developed for FreeBSD 5.0

4.1 Perl and Named Pipes

Perl and named pipes provide a simple environment for prototyping NetFS. A series of Perl scripts are run just after the OS boots, which script built the */net* directory based on the existing hardware and running configuration, and starts daemon processes behind named pipes as interfaces to the OS data. Named pipes look and behave like files, but are actually a communications channel to hidden, daemon-like processes. Both sides – the daemon process and the user process accessing the named pipe – interact using file system operations (reads, writes, *etc.*). When the daemon process opens a named pipe for reading it waits until a writer arrives to provide data. Similarly, if a daemon process opens a named pipe for writing it will hold and wait until a user process has opened that file for reading. Depending on whether the named pipe is used for output or input, NetFS initiates daemon processes waiting to provide that pipe with the corresponding functionality.

For example if a user opened the named pipe */net/iface/xl0/ip/address_out* by typing:

```
?] cat /net/iface/xl0/ip/address_out
```

NetFS would run `ifconfig xl0`, parse the outputted data, and print the IP address to the named pipe and the user would see:

```
192.168.1.1
```

This NetFS implementation redirects information between the named pipes and the underlying network configuration commands. Operations on named pipes are translated to command-line operations, and vice versa (Figure 6).

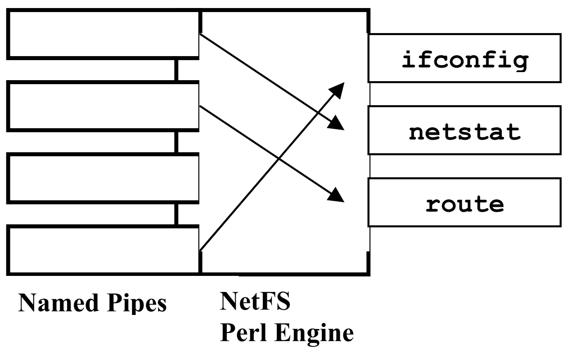

Figure 6 Conversion between named pipes and command-line operations

There are a few challenges and limits to this technique. Most implementations of named pipes prohibit using the same named pipe descriptor for both output and input. The interim solution, as noted above, was to create a pair of corresponding pipes, suffixed with *_in* and *_out*; although awkward, this provides a sufficient solution.

Creating new objects such as routes or interfaces creating a new directory proved to be a more difficult task. Named pipes sit beneath file-like descriptors, not directory-like descriptors; it is infeasible to have a directory-like named pipe to intercept *vnode* creation and create entire subdirectories instead, as desired.

Instead, a separate, true daemon process polled the */net* directory structure and effected needed consistency corrections. Virtualization proved to be beyond the capabilities of the named pipe version.

The primary reason for these difficulties was that NetFS is creating a new file system with custom semantics; named pipes affect only unidirectional file semantics. Instead, it would have been more useful to completely remap *vnode* operations, to provide completely custom, bidirectional (read and write on the same node) semantics. As a result, after a proof-of-concept named pipe NetFS implementation was completed, development focused on a KLM to provide new *vnode* operations.

4.2 Loadable Kernel Module (KLM)

The current goal of the NetFS project is to develop loadable kernel modules for a variety of operating systems. Kernel modules provide the most powerful way of implementing the NetFS *vnode* semantics. Other virtual file systems such as Procfs and Kernfs have been developed as loadable kernel modules [2] [4]. There are several advantages to this approach:

- Direct access to kernel data dtructures and functions. NetFS will not have to run other applications (such as `ifconfig, netstat, routed`) or use external APIs (*sysctl, ioctl, sockopt)* to obtain and set information.

- All *vnode* operations can be supported and customized. The current virtual file system (VFS) provides 32 *vnode* operations that can be modified; named pipes were limited to file open, read, and write only.

- Policy implementation can be integrated with the ACLs [10].

Developing a KLM has a higher learning curve than writing Perl scripts as named pipes. Work has begun on developing the NetFS KLM for FreeBSD 5.0 and features will be added incrementally. Support for other OSs take longer using this technique, but the additional capabilities warrant the cost.

5 Future Work

NetFS is focused on network configuration of parameters of hosts and routers. Future work expands NetFS to accommodate all network operations. For example, NetFS will implement socket similar to that of Plan9, *i.e.*, the */net/tcp* directory [6]. Additionally, other OSs targeted include Linux and MacOS X.

Because NetFS is an interface to the OS networking structures, it will need continual maintenance, as do most KLMs. As new features are added to the network, these will also need to be added to NetFS. VPN configuration, and support for truly virtual networks, such as X-Bone, are readily supported (and provided one motivation for) NetFS.

6 Related Work

NetFS is a combination of ideas from several sources. It closely follows the functional model of Procfs and Kernfs. It adopts the socket directory structure of Plan9 and further focuses them towards the networking components of UNIX like machines. It has the same security goals as Jail and TrustedBSD.

6.1 Procfs and Kernfs

The process file system, *procfs* (/proc), was developed as part of Eighth Edition UNIX as a mechanism to support process debugging [4]. The file system paradigm provided protection that allowed users to debug their own programs without requiring complicated controlled access to kernel data structures. By mapping processes onto the file system model, "*the most obvious security loopholes are plugged by the file system itself.*" [4]

These systems first noted that file interface is more familiar than kernel data structures, and simpler to code to. Even in its early early versions, *ps(1)* ran four times faster when coded to read the /proc files than conventionally (reading kernel data structures directly). The early version of *procfs* provided the process's memory space as a memory-mapped file, where the debugger could read or write the process space as file operations. Additional control functions used the *ioctl* system call interface to stop, start, or trace a running program. Later variants proposed incorporating these into *procfs* directly, eliminating process *ioctls* altogether. This has been implemented in FreeBSD's variant of *procfs* as a *ctl* file, one of a number of files under a directory for each process. Commands to start/stop/debug the running process are issued by writing text commands to the *ctl* file.

In more recent open-source operating systems (*e.g.,* FreeBSD and Linux), *procfs* and variants such as *kernfs* have entirely replaced the inspection of kernel data structures for common commands such as *ps(1)*, *vmstat(1)*, and *kill(1)*. In Linux, *procfs* is largely read-only, excepting only kernel variables, which are exclusively accessed using this interface. NetFS applies the *procfs* model to the network API. Diverse components of this API, including the routing table, configuration of interfaces, IP security keys, and firewall rules, are presented as a unified file system interface (*/net*). Commands and configuration are file system operations, further extending file system protection to the configuration of individual network components.

6.2 Plan 9

Plan 9 is a portable operating system that examined a number of new design paradigms [6]. Notably for NetFS, Plan 9 implements reliable protocol connections as files in a */net* directory. Individual connections are represented as directories with two files – one for control, one for data. A separate clone file, per protocol class, provides a method to create new connections. Opening this clone file results in the allocation of a free port; reading that file returns the port thus allocated.

Plan 9 focuses on a file system representation of user-level communication operations, *i.e.*, the socket API. It primarily provides programmers the simpler file system model; a secondary goal is to enable portability of tools across different protocols. Additionally, the use of text strings for control avoids byte-order issues, and provides remote access (*e.g.*, gateways) using existing remote file access. Operations that do not map directly to file semantics, such as TCP *listen*, are mapped to the side effect of file accesses. Even with these side effects, Plan 9's file system interface is typically more comprehensible than conventional socket incantations (*e.g.,* socket/bind/listen/connect).

NetFS includes a version of Plan 9 socket API, though it is not its primary focus. NetFS is directed at managing network configuration rather than individual connections. Individual connections are already afforded levels of mutual exclusion and fine-grained (per port) protection by existing implementations. Unification of the network configuration API does not necessarily require reimplementation of the socket API, though it is a component of NetFS, mostly for completeness.

6.3 Jail

The Jail system addresses the goals of limiting the behavior of processes that require root access. Jail supports a constrained environment for server processes sharing a single machine, *e.g.,* at an ISP. It provides a transitive environment that follows children spawned from a parent process, by creating a new process environment by modifying key system calls. Access to network interfaces, drivers, file systems, and kernel data structures are all accessed by modified code.

Although the footprint of Jail is very small (400 lines of code), Jail necessarily affects a large number of system calls (affecting 50 files), and thus may be difficult to port to different OSs, especially those whose process models differ substantially from FreeBSDs. NetFS, by contrast, implements a file system, which is a more commonly added feature. NetFS further focuses on network configuration, whereas Jail focuses on the entire process environment, including relative file system mount points.

Jail provides relative network parameters, similar to those of NetFS. Its networking model is more primitive than FreeBSD, however, as it provides only a single IP address per partition, where NetFS provides any subset of the /net directory. Jail intercepts network commands that would result in system-wide views, such as binds to INADDR_ANY and commands that list all the interfaces of a machine (*ifconfig –a*, *netstat –a*). Because such commands are implemented as file system operations in NetFS, it is easier to confine them to subsets of the network devices.

Finally, Jail assumes mutually exclusive partitions. Once a process enters a partition (a jail), it cannot leave; furthermore, jails do not overlap. NetFS supports recursive environments (subsets of existing subsets of resources), as well as overlapping environments (*e.g.*, gateway processes) because these models are already supported in the file system by its existing protection mechanisms.

6.4 Trusted BSD

The Trusted BSD project is a project that improves the kernel and user interface security structure to better support fine grained access control [10]. It augments existing kernel code to better accommodate fine-grained privileges, rather than just a single root privilege level.

Although NetFS does not focus on modifying existing kernel structures, it can capitalize on Trusted BSD features. ACLs added to various kernel data structures translate directly onto file system access, where NetFS operates.

7 Conclusion

NetFS is a valuable tool for network users of all levels. It does not require users to change from previous configuration methods and offers a new way to configure network parameters.

By representing the network configuration information through the file system, the task of configuring the network is simplified. The file system provides opportunities to use a single API for all aspects, including network interfaces, routes, *etc.* Access control is achieved through simple file system operations. Because the interface is unified, network configuration tasks are much easier and the learning curve is greatly reduced. Furthermore, NetFS can be used across different platforms, making it easier to port configuration programs across OSs.

Remote mount abilities provide the user with powerful remote administration facilities. Per process virtualization features provide unique opportunities to virtualize the network configuration tasks, thus making it even simpler to configure the network.

An initial prototype of the system based on Perl and named pipes exists for FreeBSD 4.7 and can be obtained by contacting the authors, and a KLM version is being developed for FreeBSD 5.0.

References

[1] Case, J., Fedor, M., Schoffstall, M., Davin, J., "A Simple Network Management Protocol (SNMP)", RFC-1157, 1990 .

[2] Faulkner, R, Gomes, R., "The Process File System and Process Model in UNIX System V," Proc. Winter Usenix, 1991, pp. 243-251.

[3] FreeBSD *man* pages, *e.g.*, http://www.freebsd.org/

[4] Killian, T. J., "Processes as Files," Proc. Summer Usenix Conf, 1984, pp. 203-207.

[5] Linux *man* pages, *e.g.*, http://www.linuxcentral.com/linux/man-pages

[6] Presotto, D., Winterbottom, P., "The Organization of Networks in Plan 9," Proc. Winter Usenix Conf., 1993, pp. 271-280.

[7] Sandberg R., Goldberg D., Kleiman D., Walsh D., Lyon B., "Design and Implementation of the Sun Network File System," Proceedings of the USENIX Conference, June 1985, pp. 119-130.

[8] Solaris 9 *man* pages. *e.g.*, http://docs.sun.com

[9] Touch, J., "Dynamic Internet Overlay Deployment and Management Using the X-Bone," Computer Networks (1Q2001). Previously in Proc. ICNP 2000, pp. 59-68.

[10] Watson, R., "TrustedBSD, Adding Trusted Operating System Features to FreeBSD," Proc. Usenix Technical Conference, June 2001.