Plan 9 from Bell Labs

Rob Pike
Dave Presotto
Sean Dorward
Bob Flandrena
Ken Thompson
Howard Trickey
Phil Winterbottom

Bell Laboratories
Murray Hill, New Jersey 07974
USA

Motivation

By the mid 1980's, the trend in computing was away from large centralized time-shared computers towards networks of smaller, personal machines, typically UNIX 'workstations'. People had grown weary of overloaded, bureaucratic timesharing machines and were eager to move to small, self-maintained systems, even if that meant a net loss in computing power. As microcomputers became faster, even that loss was recovered, and this style of computing remains popular today.

In the rush to personal workstations, though, some of their weaknesses were overlooked. First, the operating system they run, UNIX, is itself an old timesharing system and has had trouble adapting to ideas born after it. Graphics and networking were added to UNIX well into its lifetime and remain poorly integrated and difficult to administer. More important, the early focus on having private machines made it difficult for networks of machines to serve as seamlessly as the old monolithic timesharing systems. Timesharing centralized the management and amortization of costs and resources; personal computing fractured, democratized, and ultimately amplified administrative problems. The choice of an old timesharing operating system to run those personal machines made it difficult to bind things together smoothly.

Plan 9 began in the late 1980's as an attempt to have it both ways: to build a system that was centrally administered and cost-effective using cheap modern microcomputers as its computing elements. The idea was to build a time-sharing system out of workstations, but in a novel way. Different computers would handle different tasks: small, cheap machines in people's offices would serve as terminals providing access to large, central, shared resources such as computing servers and file servers. For the central machines, the coming wave of shared-memory multiprocessors seemed obvious candidates. The philosophy is much like that of the Cambridge Distributed System [NeHe82]. The early catch phrase was to build a UNIX out of a lot of little systems, not a system out of a lot of little UNIXes.

The problems with UNIX were too deep to fix, but some of its ideas could be brought along. The best was its use of the file system to coordinate naming of and

Appeared in a slightly different form in *Computing Systems*, Vol 8 #3, Summer 1995, pp. 221-254.

access to resources, even those, such as devices, not traditionally treated as files. For Plan 9, we adopted this idea by designing a network-level protocol, called 9P, to enable machines to access files on remote systems. Above this, we built a naming system that lets people and their computing agents build customized views of the resources in the network. This is where Plan 9 first began to look different: a Plan 9 user builds a private computing environment and recreates it wherever desired, rather than doing all computing on a private machine. It soon became clear that this model was richer than we had foreseen, and the ideas of per-process name spaces and file-system-like resources were extended throughout the system—to processes, graphics, even the network itself.

By 1989 the system had become solid enough that some of us began using it as our exclusive computing environment. This meant bringing along many of the services and applications we had used on UNIX. We used this opportunity to revisit many issues, not just kernel-resident ones, that we felt UNIX addressed badly. Plan 9 has new compilers, languages, libraries, window systems, and many new applications. Many of the old tools were dropped, while those brought along have been polished or rewritten.

Why be so all-encompassing? The distinction between operating system, library, and application is important to the operating system researcher but uninteresting to the user. What matters is clean functionality. By building a complete new system, we were able to solve problems where we thought they should be solved. For example, there is no real 'tty driver' in the kernel; that is the job of the window system. In the modern world, multi-vendor and multi-architecture computing are essential, yet the usual compilers and tools assume the program is being built to run locally; we needed to rethink these issues. Most important, though, the test of a system is the computing environment it provides. Producing a more efficient way to run the old UNIX warhorses is empty engineering; we were more interested in whether the new ideas suggested by the architecture of the underlying system encourage a more effective way of working. Thus, although Plan 9 provides an emulation environment for running POSIX commands, it is a backwater of the system. The vast majority of system software is developed in the 'native' Plan 9 environment.

There are benefits to having an all-new system. First, our laboratory has a history of building experimental peripheral boards. To make it easy to write device drivers, we want a system that is available in source form (no longer guaranteed with UNIX, even in the laboratory in which it was born). Also, we want to redistribute our work, which means the software must be locally produced. For example, we could have used some vendors' C compilers for our system, but even had we overcome the problems with cross-compilation, we would have difficulty redistributing the result.

This paper serves as an overview of the system. It discusses the architecture from the lowest building blocks to the computing environment seen by users. It also serves as an introduction to the rest of the Plan 9 Programmer's Manual, which it accompanies. More detail about topics in this paper can be found elsewhere in the manual.

Design

The view of the system is built upon three principles. First, resources are named and accessed like files in a hierarchical file system. Second, there is a standard protocol, called 9P, for accessing these resources. Third, the disjoint hierarchies provided by different services are joined together into a single private hierarchical file name space. The unusual properties of Plan 9 stem from the consistent, aggressive application of these principles.

A large Plan 9 installation has a number of computers networked together, each providing a particular class of service. Shared multiprocessor servers provide computing cycles; other large machines offer file storage. These machines are located in an air-conditioned machine room and are connected by high-performance networks. Lower bandwidth networks such as Ethernet or ISDN connect these servers to office-

and home-resident workstations or PCs, called terminals in Plan 9 terminology. Figure 1 shows the arrangement.

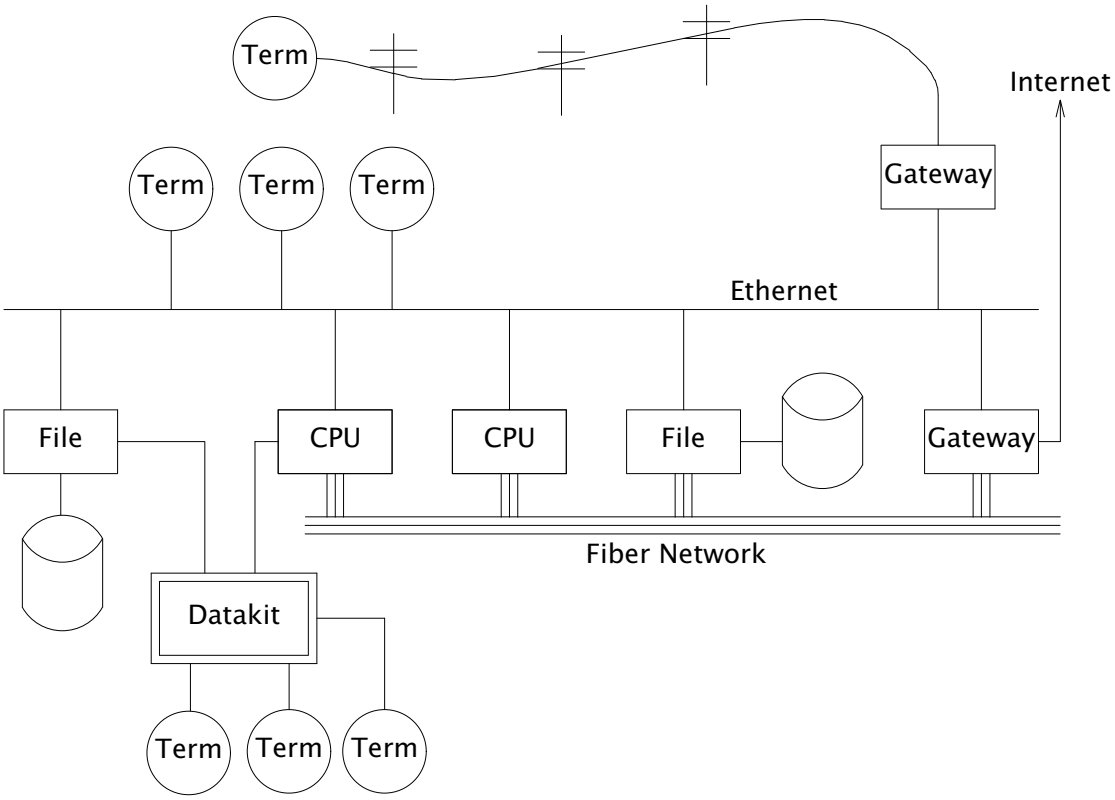


Figure 1. Structure of a large Plan 9 installation. CPU servers and file servers share fast local-area networks, while terminals use slower wider-area networks such as Ethernet, Datakit, or telephone lines to connect to them. Gateway machines, which are just CPU servers connected to multiple networks, allow machines on one network to see another.

The modern style of computing offers each user a dedicated workstation or PC. Plan 9's approach is different. The various machines with screens, keyboards, and mice all provide access to the resources of the network, so they are functionally equivalent, in the manner of the terminals attached to old timesharing systems. When someone uses the system, though, the terminal is temporarily personalized by that user. Instead of customizing the hardware, Plan 9 offers the ability to customize one's view of the system provided by the software. That customization is accomplished by giving local, personal names for the publicly visible resources in the network. Plan 9 provides the mechanism to assemble a personal view of the public space with local names for globally accessible resources. Since the most important resources of the network are files, the model of that view is file-oriented.

The client's local name space provides a way to customize the user's view of the network. The services available in the network all export file hierarchies. Those important to the user are gathered together into a custom name space; those of no immediate interest are ignored. This is a different style of use from the idea of a 'uniform global name space'. In Plan 9, there are known names for services and uniform names for files exported by those services, but the view is entirely local. As an analogy, consider the difference between the phrase 'my house' and the precise address of the speaker's home. The latter may be used by anyone but the former is easier to say and makes sense when spoken. It also changes meaning depending on who says it, yet that does

not cause confusion. Similarly, in Plan 9 the name /dev/cons always refers to the user's terminal and /bin/date the correct version of the date command to run, but which files those names represent depends on circumstances such as the architecture of the machine executing date. Plan 9, then, has local name spaces that obey globally understood conventions; it is the conventions that guarantee sane behavior in the presence of local names.

The 9P protocol is structured as a set of transactions that send a request from a client to a (local or remote) server and return the result. 9P controls file systems, not just files: it includes procedures to resolve file names and traverse the name hierarchy of the file system provided by the server. On the other hand, the client's name space is held by the client system alone, not on or with the server, a distinction from systems such as Sprite [OCDNW88]. Also, file access is at the level of bytes, not blocks, which distinguishes 9P from protocols like NFS and RFS. A paper by Welch compares Sprite, NFS, and Plan 9's network file system structures [Welc94].

This approach was designed with traditional files in mind, but can be extended to many other resources. Plan 9 services that export file hierarchies include I/O devices, backup services, the window system, network interfaces, and many others. One example is the process file system, /proc, which provides a clean way to examine and control running processes. Precursor systems had a similar idea [Kill84], but Plan 9 pushes the file metaphor much further [PPTTW93]. The file system model is well-understood, both by system builders and general users, so services that present file-like interfaces are easy to build, easy to understand, and easy to use. Files come with agreed-upon rules for protection, naming, and access both local and remote, so services built this way are ready-made for a distributed system. (This is a distinction from 'object-oriented' models, where these issues must be faced anew for every class of object.) Examples in the sections that follow illustrate these ideas in action.

The Command-level View

Plan 9 is meant to be used from a machine with a screen running the window system. It has no notion of 'teletype' in the UNIX sense. The keyboard handling of the bare system is rudimentary, but once the window system, 8½ [Pike91], is running, text can be edited with 'cut and paste' operations from a pop-up menu, copied between windows, and so on. 8½ permits editing text from the past, not just on the current input line. The text-editing capabilities of 8½ are strong enough to displace special features such as history in the shell, paging and scrolling, and mail editors. 8½ windows do not support cursor addressing and, except for one terminal emulator to simplify connecting to traditional systems, there is no cursor-addressing software in Plan 9.

Each window is created in a separate name space. Adjustments made to the name space in a window do not affect other windows or programs, making it safe to experiment with local modifications to the name space, for example to substitute files from the dump file system when debugging. Once the debugging is done, the window can be deleted and all trace of the experimental apparatus is gone. Similar arguments apply to the private space each window has for environment variables, notes (analogous to UNIX signals), etc.

Each window is created running an application, such as the shell, with standard input and output connected to the editable text of the window. Each window also has a private bitmap and multiplexed access to the keyboard, mouse, and other graphical resources through files like /dev/mouse, /dev/bitblt, and /dev/cons (analogous to UNIX's /dev/tty). These files are provided by 8½, which is implemented as a file server. Unlike X windows, where a new application typically creates a new window to run in, an 8½ graphics application usually runs in the window where it starts. It is possible and efficient for an application to create a new window, but that is not the style of the system. Again contrasting to X, in which a remote application makes a network call

to the X server to start running, a remote 8½ application sees the `mouse`, `bitblt`, and `cons` files for the window as usual in `/dev`; it does not know whether the files are local. It just reads and writes them to control the window; the network connection is already there and multiplexed.

The intended style of use is to run interactive applications such as the window system and text editor on the terminal and to run computation- or file-intensive applications on remote servers. Different windows may be running programs on different machines over different networks, but by making the name space equivalent in all windows, this is transparent: the same commands and resources are available, with the same names, wherever the computation is performed.

The command set of Plan 9 is similar to that of UNIX. The commands fall into several broad classes. Some are new programs for old jobs: programs like `ls`, `cat`, and `who` have familiar names and functions but are new, simpler implementations. `Who`, for example, is a shell script, while `ps` is just 95 lines of C code. Some commands are essentially the same as their UNIX ancestors: `awk`, `troff`, and others have been converted to ANSI C and extended to handle Unicode, but are still the familiar tools. Some are entirely new programs for old niches: the shell `rc`, text editor `sam`, debugger `acid`, and others displace the better-known UNIX tools with similar jobs. Finally, about half the commands are new.

Compatibility was not a requirement for the system. Where the old commands or notation seemed good enough, we kept them. When they didn't, we replaced them.

The File Server

A central file server stores permanent files and presents them to the network as a file hierarchy exported using 9P. The server is a stand-alone system, accessible only over the network, designed to do its one job well. It runs no user processes, only a fixed set of routines compiled into the boot image. Rather than a set of disks or separate file systems, the main hierarchy exported by the server is a single tree, representing files on many disks. That hierarchy is shared by many users over a wide area on a variety of networks. Other file trees exported by the server include special-purpose systems such as temporary storage and, as explained below, a backup service.

The file server has three levels of storage. The central server in our installation has about 100 megabytes of memory buffers, 27 gigabytes of magnetic disks, and 350 gigabytes of bulk storage in a write-once-read-many (WORM) jukebox. The disk is a cache for the WORM and the memory is a cache for the disk; each is much faster, and sees about an order of magnitude more traffic, than the level it caches. The addressable data in the file system can be larger than the size of the magnetic disks, because they are only a cache; our main file server has about 40 gigabytes of active storage.

The most unusual feature of the file server comes from its use of a WORM device for stable storage. Every morning at 5 o'clock, a *dump* of the file system occurs automatically. The file system is frozen and all blocks modified since the last dump are queued to be written to the WORM. Once the blocks are queued, service is restored and the read-only root of the dumped file system appears in a hierarchy of all dumps ever taken, named by its date. For example, the directory `/n/dump/1995/0315` is the root directory of an image of the file system as it appeared in the early morning of March 15, 1995. It takes a few minutes to queue the blocks, but the process to copy blocks to the WORM, which runs in the background, may take hours.

There are two ways the dump file system is used. The first is by the users themselves, who can browse the dump file system directly or attach pieces of it to their name space. For example, to track down a bug, it is straightforward to try the compiler from three months ago or to link a program with yesterday's library. With daily snapshots of all files, it is easy to find when a particular change was made or what changes were

made on a particular date. People feel free to make large speculative changes to files in the knowledge that they can be backed out with a single copy command. There is no backup system as such; instead, because the dump is in the file name space, backup problems can be solved with standard tools such as `cp`, `ls`, `grep`, and `diff`.

The other (very rare) use is complete system backup. In the event of disaster, the active file system can be initialized from any dump by clearing the disk cache and setting the root of the active file system to be a copy of the dumped root. Although easy to do, this is not to be taken lightly: besides losing any change made after the date of the dump, this recovery method results in a very slow system. The cache must be reloaded from WORM, which is much slower than magnetic disks. The file system takes a few days to reload the working set and regain its full performance.

Access permissions of files in the dump are the same as they were when the dump was made. Normal utilities have normal permissions in the dump without any special arrangement. The dump file system is read-only, though, which means that files in the dump cannot be written regardless of their permission bits; in fact, since directories are part of the read-only structure, even the permissions cannot be changed.

Once a file is written to WORM, it cannot be removed, so our users never see ''please clean up your files'' messages and there is no `df` command. We regard the WORM jukebox as an unlimited resource. The only issue is how long it will take to fill. Our WORM has served a community of about 50 users for five years and has absorbed daily dumps, consuming a total of 65% of the storage in the jukebox. In that time, the manufacturer has improved the technology, doubling the capacity of the individual disks. If we were to upgrade to the new media, we would have more free space than in the original empty jukebox. Technology has created storage faster than we can use it.

Unusual file servers

Plan 9 is characterized by a variety of servers that offer a file-like interface to unusual services. Many of these are implemented by user-level processes, although the distinction is unimportant to their clients; whether a service is provided by the kernel, a user process, or a remote server is irrelevant to the way it is used. There are dozens of such servers; in this section we present three representative ones.

Perhaps the most remarkable file server in Plan 9 is 8½, the window system. It is discussed at length elsewhere [Pike91], but deserves a brief explanation here. 8½ provides two interfaces: to the user seated at the terminal, it offers a traditional style of interaction with multiple windows, each running an application, all controlled by a mouse and keyboard. To the client programs, the view is also fairly traditional: programs running in a window see a set of files in `/dev` with names like `mouse`, `screen`, and `cons`. Programs that want to print text to their window write to `/dev/cons`; to read the mouse, they read `/dev/mouse`. In the Plan 9 style, bitmap graphics is implemented by providing a file `/dev/bitblt` on which clients write encoded messages to execute graphical operations such as `bitblt` (RasterOp). What is unusual is how this is done: 8½ is a file server, serving the files in `/dev` to the clients running in each window. Although every window looks the same to its client, each window has a distinct set of files in `/dev`. 8½ multiplexes its clients' access to the resources of the terminal by serving multiple sets of files. Each client is given a private name space with a *different* set of files that behave the same as in all other windows. There are many advantages to this structure. One is that 8½ serves the same files it needs for its own implementation—it multiplexes its own interface—so it may be run, recursively, as a client of itself. Also, consider the implementation of `/dev/tty` in UNIX, which requires special code in the kernel to redirect `open` calls to the appropriate device. Instead, in 8½ the equivalent service falls out automatically: 8½ serves `/dev/cons` as its basic function; there is nothing extra to do. When a program wants to read from the keyboard, it opens `/dev/cons`, but it is a private file, not a shared one with special

properties. Again, local name spaces make this possible; conventions about the consistency of the files within them make it natural.

8½ has a unique feature made possible by its design. Because it is implemented as a file server, it has the power to postpone answering read requests for a particular window. This behavior is toggled by a reserved key on the keyboard. Toggling once suspends client reads from the window; toggling again resumes normal reads, which absorb whatever text has been prepared, one line at a time. This allows the user to edit multi-line input text on the screen before the application sees it, obviating the need to invoke a separate editor to prepare text such as mail messages. A related property is that reads are answered directly from the data structure defining the text on the display: text may be edited until its final newline makes the prepared line of text readable by the client. Even then, until the line is read, the text the client will read can be changed. For example, after typing

```
% make
rm *
```

to the shell, the user can backspace over the final newline at any time until `make` finishes, holding off execution of the `rm` command, or even point with the mouse before the `rm` and type another command to be executed first.

There is no `ftp` command in Plan 9. Instead, a user-level file server called `ftpfs` dials the FTP site, logs in on behalf of the user, and uses the FTP protocol to examine files in the remote directory. To the local user, it offers a file hierarchy, attached to `/n/ftp` in the local name space, mirroring the contents of the FTP site. In other words, it translates the FTP protocol into 9P to offer Plan 9 access to FTP sites. The implementation is tricky; `ftpfs` must do some sophisticated caching for efficiency and use heuristics to decode remote directory information. But the result is worthwhile: all the local file management tools such as `cp`, `grep`, `diff`, and of course `ls` are available to FTP-served files exactly as if they were local files. Other systems such as Jade and Prospero have exploited the same opportunity [Rao81, Neu92], but because of local name spaces and the simplicity of implementing 9P, this approach fits more naturally into Plan 9 than into other environments.

One server, `exportfs`, is a user process that takes a portion of its own name space and makes it available to other processes by translating 9P requests into system calls to the Plan 9 kernel. The file hierarchy it exports may contain files from multiple servers. `Exportfs` is usually run as a remote server started by a local program, either `import` or `cpu`. `Import` makes a network call to the remote machine, starts `exportfs` there, and attaches its 9P connection to the local name space. For example,

```
import helix /net
```

makes Helix's network interfaces visible in the local `/net` directory. Helix is a central server and has many network interfaces, so this permits a machine with one network to access to any of Helix's networks. After such an import, the local machine may make calls on any of the networks connected to Helix. Another example is

```
import helix /proc
```

which makes Helix's processes visible in the local `/proc`, permitting local debuggers to examine remote processes.

The `cpu` command connects the local terminal to a remote CPU server. It works in the opposite direction to `import`: after calling the server, it starts a *local* `exportfs` and mounts it in the name space of a process, typically a newly created shell, on the server. It then rearranges the name space to make local device files (such as those served by the terminal's window system) visible in the server's `/dev` directory. The effect of running a `cpu` command is therefore to start a shell on a fast machine, one more tightly coupled to the file server, with a name space analogous to the local one.

All local device files are visible remotely, so remote applications have full access to local services such as bitmap graphics, /dev/cons, and so on. This is not the same as rlogin, which does nothing to reproduce the local name space on the remote system, nor is it the same as file sharing with, say, NFS, which can achieve some name space equivalence but not the combination of access to local hardware devices, remote files, and remote CPU resources. The cpu command is a uniquely transparent mechanism. For example, it is reasonable to start a window system in a window running a cpu command; all windows created there automatically start processes on the CPU server.

Configurability and administration

The uniform interconnection of components in Plan 9 makes it possible to configure a Plan 9 installation many different ways. A single laptop PC can function as a stand-alone Plan 9 system; at the other extreme, our setup has central multiprocessor CPU servers and file servers and scores of terminals ranging from small PCs to high-end graphics workstations. It is such large installations that best represent how Plan 9 operates.

The system software is portable and the same operating system runs on all hardware. Except for performance, the appearance of the system on, say, an SGI workstation is the same as on a laptop. Since computing and file services are centralized, and terminals have no permanent file storage, all terminals are functionally identical. In this way, Plan 9 has one of the good properties of old timesharing systems, where a user could sit in front of any machine and see the same system. In the modern workstation community, machines tend to be owned by people who customize them by storing private information on local disk. We reject this style of use, although the system itself can be used this way. In our group, we have a laboratory with many public-access machines—a terminal room—and a user may sit down at any one of them and work.

Central file servers centralize not just the files, but also their administration and maintenance. In fact, one server is the main server, holding all system files; other servers provide extra storage or are available for debugging and other special uses, but the system software resides on one machine. This means that each program has a single copy of the binary for each architecture, so it is trivial to install updates and bug fixes. There is also a single user database; there is no need to synchronize distinct /etc/passwd files. On the other hand, depending on a single central server does limit the size of an installation.

Another example of the power of centralized file service is the way Plan 9 administers network information. On the central server there is a directory, /lib/ndb, that contains all the information necessary to administer the local Ethernet and other networks. All the machines use the same database to talk to the network; there is no need to manage a distributed naming system or keep parallel files up to date. To install a new machine on the local Ethernet, choose a name and IP address and add these to a single file in /lib/ndb; all the machines in the installation will be able to talk to it immediately. To start running, plug the machine into the network, turn it on, and use BOOTP and TFTP to load the kernel. All else is automatic.

Finally, the automated dump file system frees all users from the need to maintain their systems, while providing easy access to backup files without tapes, special commands, or the involvement of support staff. It is difficult to overstate the improvement in lifestyle afforded by this service.

Plan 9 runs on a variety of hardware without constraining how to configure an installation. In our laboratory, we chose to use central servers because they amortize costs and administration. A sign that this is a good decision is that our cheap terminals remain comfortable places to work for about five years, much longer than workstations that must provide the complete computing environment. We do, however, upgrade the central machines, so the computation available from even old Plan 9 terminals improves

with time. The money saved by avoiding regular upgrades of terminals is instead spent on the newest, fastest multiprocessor servers. We estimate this costs about half the money of networked workstations yet provides general access to more powerful machines.

C Programming

Plan 9 utilities are written in several languages. Some are scripts for the shell, `rc` [Duff90]; a handful are written in a new C-like concurrent language called Alef [Wint95], described below. The great majority, though, are written in a dialect of ANSI C [ANSIC]. Of these, most are entirely new programs, but some originate in pre-ANSI C code from our research UNIX system [UNIX85]. These have been updated to ANSI C and reworked for portability and cleanliness.

The Plan 9 C dialect has some minor extensions, described elsewhere [Pike95], and a few major restrictions. The most important restriction is that the compiler demands that all function definitions have ANSI prototypes and all function calls appear in the scope of a prototyped declaration of the function. As a stylistic rule, the prototyped declaration is placed in a header file included by all files that call the function. Each system library has an associated header file, declaring all functions in that library. For example, the standard Plan 9 library is called `libc`, so all C source files include `<libc.h>`. These rules guarantee that all functions are called with arguments having the expected types — something that was not true with pre-ANSI C programs.

Another restriction is that the C compilers accept only a subset of the preprocessor directives required by ANSI. The main omission is `#if`, since we believe it is never necessary and often abused. Also, its effect is better achieved by other means. For instance, an `#if` used to toggle a feature at compile time can be written as a regular `if` statement, relying on compile-time constant folding and dead code elimination to discard object code.

Conditional compilation, even with `#ifdef`, is used sparingly in Plan 9. The only architecture-dependent `#ifdefs` in the system are in low-level routines in the graphics library. Instead, we avoid such dependencies or, when necessary, isolate them in separate source files or libraries. Besides making code hard to read, `#ifdefs` make it impossible to know what source is compiled into the binary or whether source protected by them will compile or work properly. They make it harder to maintain software.

The standard Plan 9 library overlaps much of ANSI C and POSIX [POSIX], but diverges when appropriate to Plan 9's goals or implementation. When the semantics of a function change, we also change the name. For instance, instead of UNIX's `creat`, Plan 9 has a `create` function that takes three arguments, the original two plus a third that, like the second argument of `open`, defines whether the returned file descriptor is to be opened for reading, writing, or both. This design was forced by the way 9P implements creation, but it also simplifies the common use of `create` to initialize a temporary file.

Another departure from ANSI C is that Plan 9 uses a 16-bit character set called Unicode [ISO10646, Unicode]. Although we stopped short of full internationalization, Plan 9 treats the representation of all major languages uniformly throughout all its software. To simplify the exchange of text between programs, the characters are packed into a byte stream by an encoding we designed, called UTF-8, which is now becoming accepted as a standard [FSSUTF]. It has several attractive properties, including byte-order independence, backwards compatibility with ASCII, and ease of implementation.

There are many problems in adapting existing software to a large character set with an encoding that represents characters with a variable number of bytes. ANSI C addresses some of the issues but falls short of solving them all. It does not pick a character set encoding and does not define all the necessary I/O library routines.

Furthermore, the functions it *does* define have engineering problems. Since the standard left too many problems unsolved, we decided to build our own interface. A separate paper has the details [Pike93].

A small class of Plan 9 programs do not follow the conventions discussed in this section. These are programs imported from and maintained by the UNIX community; `tex` is a representative example. To avoid reconverting such programs every time a new version is released, we built a porting environment, called the ANSI C/POSIX Environment, or APE [Tric95]. APE comprises separate include files, libraries, and commands, conforming as much as possible to the strict ANSI C and base-level POSIX specifications. To port network-based software such as X Windows, it was necessary to add some extensions to those specifications, such as the BSD networking functions.

Portability and Compilation

Plan 9 is portable across a variety of processor architectures. Within a single computing session, it is common to use several architectures: perhaps the window system running on an Intel processor connected to a MIPS-based CPU server with files resident on a SPARC system. For this heterogeneity to be transparent, there must be conventions about data interchange between programs; for software maintenance to be straightforward, there must be conventions about cross-architecture compilation.

To avoid byte order problems, data is communicated between programs as text whenever practical. Sometimes, though, the amount of data is high enough that a binary format is necessary; such data is communicated as a byte stream with a predefined encoding for multi-byte values. In the rare cases where a format is complex enough to be defined by a data structure, the structure is never communicated as a unit; instead, it is decomposed into individual fields, encoded as an ordered byte stream, and then reassembled by the recipient. These conventions affect data ranging from kernel or application program state information to object file intermediates generated by the compiler.

Programs, including the kernel, often present their data through a file system interface, an access mechanism that is inherently portable. For example, the system clock is represented by a decimal number in the file `/dev/time`; the `time` library function (there is no `time` system call) reads the file and converts it to binary. Similarly, instead of encoding the state of an application process in a series of flags and bits in private memory, the kernel presents a text string in the file named `status` in the `/proc` file system associated with each process. The Plan 9 `ps` command is trivial: it prints the contents of the desired status files after some minor reformatting; moreover, after

```
import helix /proc
```

a local `ps` command reports on the status of Helix's processes.

Each supported architecture has its own compilers and loader. The C and Alef compilers produce intermediate files that are portably encoded; the contents are unique to the target architecture but the format of the file is independent of compiling processor type. When a compiler for a given architecture is compiled on another type of processor and then used to compile a program there, the intermediate produced on the new architecture is identical to the intermediate produced on the native processor. From the compiler's point of view, every compilation is a cross-compilation.

Although each architecture's loader accepts only intermediate files produced by compilers for that architecture, such files could have been generated by a compiler executing on any type of processor. For instance, it is possible to run the MIPS compiler on a 486, then use the MIPS loader on a SPARC to produce a MIPS executable.

Since Plan 9 runs on a variety of architectures, even in a single installation, distinguishing the compilers and intermediate names simplifies multi-architecture

development from a single source tree. The compilers and the loader for each architecture are uniquely named; there is no `cc` command. The names are derived by concatenating a code letter associated with the target architecture with the name of the compiler or loader. For example, the letter '8' is the code letter for Intel *x*86 processors; the C compiler is named `8c`, the Alef compiler `8al`, and the loader is called `8l`. Similarly, the compiler intermediate files are suffixed `.8`, not `.o`.

The Plan 9 build program `mk`, a relative of `make`, reads the names of the current and target architectures from environment variables called `$cputype` and `$objtype`. By default the current processor is the target, but setting `$objtype` to the name of another architecture before invoking `mk` results in a cross-build:

```
% objtype=sparc mk
```

builds a program for the SPARC architecture regardless of the executing machine. The value of `$objtype` selects a file of architecture-dependent variable definitions that configures the build to use the appropriate compilers and loader. Although simple-minded, this technique works well in practice: all applications in Plan 9 are built from a single source tree and it is possible to build the various architectures in parallel without conflict.

Parallel programming

Plan 9's support for parallel programming has two aspects. First, the kernel provides a simple process model and a few carefully designed system calls for synchronization and sharing. Second, a new parallel programming language called Alef supports concurrent programming. Although it is possible to write parallel programs in C, Alef is the parallel language of choice.

There is a trend in new operating systems to implement two classes of processes: normal UNIX-style processes and light-weight kernel threads. Instead, Plan 9 provides a single class of process but allows fine control of the sharing of a process's resources such as memory and file descriptors. A single class of process is a feasible approach in Plan 9 because the kernel has an efficient system call interface and cheap process creation and scheduling.

Parallel programs have three basic requirements: management of resources shared between processes, an interface to the scheduler, and fine-grain process synchronization using spin locks. On Plan 9, new processes are created using the `rfork` system call. `Rfork` takes a single argument, a bit vector that specifies which of the parent process's resources should be shared, copied, or created anew in the child. The resources controlled by `rfork` include the name space, the environment, the file descriptor table, memory segments, and notes (Plan 9's analog of UNIX signals). One of the bits controls whether the `rfork` call will create a new process; if the bit is off, the resulting modification to the resources occurs in the process making the call. For example, a process calls `rfork(RFNAMEG)` to disconnect its name space from its parent's. Alef uses a fine-grained fork in which all the resources, including memory, are shared between parent and child, analogous to creating a kernel thread in many systems.

An indication that `rfork` is the right model is the variety of ways it is used. Other than the canonical use in the library routine `fork`, it is hard to find two calls to `rfork` with the same bits set; programs use it to create many different forms of sharing and resource allocation. A system with just two types of processes—regular processes and threads—could not handle this variety.

There are two ways to share memory. First, a flag to `rfork` causes all the memory segments of the parent to be shared with the child (except the stack, which is forked copy-on-write regardless). Alternatively, a new segment of memory may be attached using the `segattach` system call; such a segment will always be shared between parent and child.

The rendezvous system call provides a way for processes to synchronize. Alef uses it to implement communication channels, queuing locks, multiple reader/writer locks, and the sleep and wakeup mechanism. Rendezvous takes two arguments, a tag and a value. When a process calls rendezvous with a tag it sleeps until another process presents a matching tag. When a pair of tags match, the values are exchanged between the two processes and both rendezvous calls return. This primitive is sufficient to implement the full set of synchronization routines.

Finally, spin locks are provided by an architecture-dependent library at user level. Most processors provide atomic test and set instructions that can be used to implement locks. A notable exception is the MIPS R3000, so the SGI Power series multiprocessors have special lock hardware on the bus. User processes gain access to the lock hardware by mapping pages of hardware locks into their address space using the segattach system call.

A Plan 9 process in a system call will block regardless of its 'weight'. This means that when a program wishes to read from a slow device without blocking the entire calculation, it must fork a process to do the read for it. The solution is to start a satellite process that does the I/O and delivers the answer to the main program through shared memory or perhaps a pipe. This sounds onerous but works easily and efficiently in practice; in fact, most interactive Plan 9 applications, even relatively ordinary ones written in C, such as the text editor Sam [Pike87], run as multiprocess programs.

The kernel support for parallel programming in Plan 9 is a few hundred lines of portable code; a handful of simple primitives enable the problems to be handled cleanly at user level. Although the primitives work fine from C, they are particularly expressive from within Alef. The creation and management of slave I/O processes can be written in a few lines of Alef, providing the foundation for a consistent means of multiplexing data flows between arbitrary processes. Moreover, implementing it in a language rather than in the kernel ensures consistent semantics between all devices and provides a more general multiplexing primitive. Compare this to the UNIX select system call: select applies only to a restricted set of devices, legislates a style of multiprogramming in the kernel, does not extend across networks, is difficult to implement, and is hard to use.

Another reason parallel programming is important in Plan 9 is that multi-threaded user-level file servers are the preferred way to implement services. Examples of such servers include the programming environment Acme [Pike94], the name space exporting tool exportfs [PPTTW93], the HTTP daemon, and the network name servers cs and dns [PrWi93]. Complex applications such as Acme prove that careful operating system support can reduce the difficulty of writing multi-threaded applications without moving threading and synchronization primitives into the kernel.

Implementation of Name Spaces

User processes construct name spaces using three system calls: mount, bind, and unmount. The mount system call attaches a tree served by a file server to the current name space. Before calling mount, the client must (by outside means) acquire a connection to the server in the form of a file descriptor that may be written and read to transmit 9P messages. That file descriptor represents a pipe or network connection.

The mount call attaches a new hierarchy to the existing name space. The bind system call, on the other hand, duplicates some piece of existing name space at another point in the name space. The unmount system call allows components to be removed.

Using either bind or mount, multiple directories may be stacked at a single point in the name space. In Plan 9 terminology, this is a *union* directory and behaves like the concatenation of the constituent directories. A flag argument to bind and mount specifies the position of a new directory in the union, permitting new elements to be added either at the front or rear of the union or to replace it entirely. When a file lookup

is performed in a union directory, each component of the union is searched in turn and the first match taken; likewise, when a union directory is read, the contents of each of the component directories is read in turn. Union directories are one of the most widely used organizational features of the Plan 9 name space. For instance, the directory `/bin` is built as a union of `/$cputype/bin` (program binaries), `/rc/bin` (shell scripts), and perhaps more directories provided by the user. This construction makes the shell `$PATH` variable unnecessary.

One question raised by union directories is which element of the union receives a newly created file. After several designs, we decided on the following. By default, directories in unions do not accept new files, although the `create` system call applied to an existing file succeeds normally. When a directory is added to the union, a flag to `bind` or `mount` enables create permission (a property of the name space) in that directory. When a file is being created with a new name in a union, it is created in the first directory of the union with create permission; if that creation fails, the entire `create` fails. This scheme enables the common use of placing a private directory anywhere in a union of public ones, while allowing creation only in the private directory.

By convention, kernel device file systems are bound into the `/dev` directory, but to bootstrap the name space building process it is necessary to have a notation that permits direct access to the devices without an existing name space. The root directory of the tree served by a device driver can be accessed using the syntax *#c*, where *c* is a unique character (typically a letter) identifying the *type* of the device. Simple device drivers serve a single level directory containing a few files. As an example, each serial port is represented by a data and a control file:

```
% bind -a '#t' /dev
% cd /dev
% ls -l eia*
--rw-rw-rw- t 0 bootes bootes 0 Feb 24 21:14 eia1
--rw-rw-rw- t 0 bootes bootes 0 Feb 24 21:14 eia1ctl
--rw-rw-rw- t 0 bootes bootes 0 Feb 24 21:14 eia2
--rw-rw-rw- t 0 bootes bootes 0 Feb 24 21:14 eia2ctl
```

The `bind` program is an encapsulation of the `bind` system call; its `-a` flag positions the new directory at the end of the union. The data files `eia1` and `eia2` may be read and written to communicate over the serial line. Instead of using special operations on these files to control the devices, commands written to the files `eia1ctl` and `eia2ctl` control the corresponding device; for example, writing the text string `b1200` to `/dev/eia1ctl` sets the speed of that line to 1200 baud. Compare this to the UNIX `ioctl` system call: in Plan 9, devices are controlled by textual messages, free of byte order problems, with clear semantics for reading and writing. It is common to configure or debug devices using shell scripts.

It is the universal use of the 9P protocol that connects Plan 9's components together to form a distributed system. Rather than inventing a unique protocol for each service such as `rlogin`, FTP, TFTP, and X windows, Plan 9 implements services in terms of operations on file objects, and then uses a single, well-documented protocol to exchange information between computers. Unlike NFS, 9P treats files as a sequence of bytes rather than blocks. Also unlike NFS, 9P is stateful: clients perform remote procedure calls to establish pointers to objects in the remote file server. These pointers are called file identifiers or *fids*. All operations on files supply a fid to identify an object in the remote file system.

The 9P protocol defines 17 messages, providing means to authenticate users, navigate fids around a file system hierarchy, copy fids, perform I/O, change file attributes, and create and delete files. Its complete specification is in Section 5 of the Programmer's Manual [9man]. Here is the procedure to gain access to the name hierarchy supplied by a server. A file server connection is established via a pipe or network

connection. An initial `session` message performs a bilateral authentication between client and server. An `attach` message then connects a fid suggested by the client to the root of the server file tree. The `attach` message includes the identity of the user performing the attach; henceforth all fids derived from the root fid will have permissions associated with that user. Multiple users may share the connection, but each must perform an attach to establish his or her identity.

The `walk` message moves a fid through a single level of the file system hierarchy. The `clone` message takes an established fid and produces a copy that points to the same file as the original. Its purpose is to enable walking to a file in a directory without losing the fid on the directory. The `open` message locks a fid to a specific file in the hierarchy, checks access permissions, and prepares the fid for I/O. The `read` and `write` messages allow I/O at arbitrary offsets in the file; the maximum size transferred is defined by the protocol. The `clunk` message indicates the client has no further use for a fid. The `remove` message behaves like `clunk` but causes the file associated with the fid to be removed and any associated resources on the server to be deallocated.

9P has two forms: RPC messages sent on a pipe or network connection and a procedural interface within the kernel. Since kernel device drivers are directly addressable, there is no need to pass messages to communicate with them; instead each 9P transaction is implemented by a direct procedure call. For each fid, the kernel maintains a local representation in a data structure called a *channel*, so all operations on files performed by the kernel involve a channel connected to that fid. The simplest example is a user process's file descriptors, which are indexes into an array of channels. A table in the kernel provides a list of entry points corresponding one to one with the 9P messages for each device. A system call such as `read` from the user translates into one or more procedure calls through that table, indexed by the type character stored in the channel: `procread`, `eiaread`, etc. Each call takes at least one channel as an argument. A special kernel driver, called the *mount* driver, translates procedure calls to messages, that is, it converts local procedure calls to remote ones. In effect, this special driver becomes a local proxy for the files served by a remote file server. The channel pointer in the local call is translated to the associated fid in the transmitted message.

The mount driver is the sole RPC mechanism employed by the system. The semantics of the supplied files, rather than the operations performed upon them, create a particular service such as the `cpu` command. The mount driver demultiplexes protocol messages between clients sharing a communication channel with a file server. For each outgoing RPC message, the mount driver allocates a buffer labeled by a small unique integer, called a *tag*. The reply to the RPC is labeled with the same tag, which is used by the mount driver to match the reply with the request.

The kernel representation of the name space is called the *mount table*, which stores a list of bindings between channels. Each entry in the mount table contains a pair of channels: a *from* channel and a *to* channel. Every time a walk succeeds in moving a channel to a new location in the name space, the mount table is consulted to see if a 'from' channel matches the new name; if so the 'to' channel is cloned and substituted for the original. Union directories are implemented by converting the 'to' channel into a list of channels: a successful walk to a union directory returns a 'to' channel that forms the head of a list of channels, each representing a component directory of the union. If a walk fails to find a file in the first directory of the union, the list is followed, the next component cloned, and walk tried on that directory.

Each file in Plan 9 is uniquely identified by a set of integers: the type of the channel (used as the index of the function call table), the server or device number distinguishing the server from others of the same type (decided locally by the driver), and a *qid* formed from two 32-bit numbers called *path* and *version*. The path is a unique file number assigned by a device driver or file server when a file is created. The version number is updated whenever the file is modified; as described in the next section, it can be used

to maintain cache coherency between clients and servers.

The type and device number are analogous to UNIX major and minor device numbers; the qid is analogous to the i-number. The device and type connect the channel to a device driver and the qid identifies the file within that device. If the file recovered from a walk has the same type, device, and qid path as an entry in the mount table, they are the same file and the corresponding substitution from the mount table is made. This is how the name space is implemented.

File Caching

The 9P protocol has no explicit support for caching files on a client. The large memory of the central file server acts as a shared cache for all its clients, which reduces the total amount of memory needed across all machines in the network. Nonetheless, there are sound reasons to cache files on the client, such as a slow connection to the file server.

The version field of the qid is changed whenever the file is modified, which makes it possible to do some weakly coherent forms of caching. The most important is client caching of text and data segments of executable files. When a process `execs` a program, the file is re-opened and the qid's version is compared with that in the cache; if they match, the local copy is used. The same method can be used to build a local caching file server. This user-level server interposes on the 9P connection to the remote server and monitors the traffic, copying data to a local disk. When it sees a read of known data, it answers directly, while writes are passed on immediately—the cache is write-through—to keep the central copy up to date. This is transparent to processes on the terminal and requires no change to 9P; it works well on home machines connected over serial lines. A similar method can be applied to build a general client cache in unused local memory, but this has not been done in Plan 9.

Networks and Communication Devices

Network interfaces are kernel-resident file systems, analogous to the EIA device described earlier. Call setup and shutdown are achieved by writing text strings to the control file associated with the device; information is sent and received by reading and writing the data file. The structure and semantics of the devices is common to all networks so, other than a file name substitution, the same procedure makes a call using TCP over Ethernet as URP over Datakit [Fra80].

This example illustrates the structure of the TCP device:

```
% ls -lp /net/tcp
d-r-xr-xr-x I 0 bootes bootes 0 Feb 23 20:20 0
d-r-xr-xr-x I 0 bootes bootes 0 Feb 23 20:20 1
--rw-rw-rw- I 0 bootes bootes 0 Feb 23 20:20 clone
% ls -lp /net/tcp/0
--rw-rw---- I 0 rob    bootes 0 Feb 23 20:20 ctl
--rw-rw---- I 0 rob    bootes 0 Feb 23 20:20 data
--rw-rw---- I 0 rob    bootes 0 Feb 23 20:20 listen
--r--r--r-- I 0 bootes bootes 0 Feb 23 20:20 local
--r--r--r-- I 0 bootes bootes 0 Feb 23 20:20 remote
--r--r--r-- I 0 bootes bootes 0 Feb 23 20:20 status
%
```

The top directory, `/net/tcp`, contains a `clone` file and a directory for each connection, numbered 0 to *n*. Each connection directory corresponds to an TCP/IP connection. Opening `clone` reserves an unused connection and returns its control file. Reading the control file returns the textual connection number, so the user process can construct the full name of the newly allocated connection directory. The `local`, `remote`, and `status` files are diagnostic; for example, `remote` contains the address (for TCP, the

IP address and port number) of the remote side.

A call is initiated by writing a connect message with a network-specific address as its argument; for example, to open a Telnet session (port 23) to a remote machine with IP address 135.104.9.52, the string is:

```
connect 135.104.9.52!23
```

The write to the control file blocks until the connection is established; if the destination is unreachable, the write returns an error. Once the connection is established, the `telnet` application reads and writes the `data` file to talk to the remote Telnet daemon. On the other end, the Telnet daemon would start by writing

```
announce 23
```

to its control file to indicate its willingness to receive calls to this port. Such a daemon is called a *listener* in Plan 9.

A uniform structure for network devices cannot hide all the details of addressing and communication for dissimilar networks. For example, Datakit uses textual, hierarchical addresses unlike IP's 32-bit addresses, so an application given a control file must still know what network it represents. Rather than make every application know the addressing of every network, Plan 9 hides these details in a *connection server*, called `cs`. `Cs` is a file system mounted in a known place. It supplies a single control file that an application uses to discover how to connect to a host. The application writes the symbolic address and service name for the connection it wishes to make, and reads back the name of the `clone` file to open and the address to present to it. If there are multiple networks between the machines, `cs` presents a list of possible networks and addresses to be tried in sequence; it uses heuristics to decide the order. For instance, it presents the highest-bandwidth choice first.

A single library function called `dial` talks to `cs` to establish the connection. An application that uses `dial` needs no changes, not even recompilation, to adapt to new networks; the interface to `cs` hides the details.

The uniform structure for networks in Plan 9 makes the `import` command all that is needed to construct gateways.

Kernel structure for networks

The kernel plumbing used to build Plan 9 communications channels is called *streams* [Rit84][Presotto]. A stream is a bidirectional channel connecting a physical or pseudo-device to a user process. The user process inserts and removes data at one end of the stream; a kernel process acting on behalf of a device operates at the other end. A stream comprises a linear list of *processing modules*. Each module has both an upstream (toward the process) and downstream (toward the device) *put routine*. Calling the put routine of the module on either end of the stream inserts data into the stream. Each module calls the succeeding one to send data up or down the stream. Like UNIX streams [Rit84], Plan 9 streams can be dynamically configured.

The IL Protocol

The 9P protocol must run above a reliable transport protocol with delimited messages. 9P has no mechanism to recover from transmission errors and the system assumes that each read from a communication channel will return a single 9P message; it does not parse the data stream to discover message boundaries. Pipes and some network protocols already have these properties but the standard IP protocols do not. TCP does not delimit messages, while UDP [RFC768] does not provide reliable in-order delivery.

We designed a new protocol, called IL (Internet Link), to transmit 9P messages over IP. It is a connection-based protocol that provides reliable transmission of sequenced

messages between machines. Since a process can have only a single outstanding 9P request, there is no need for flow control in IL. Like TCP, IL has adaptive timeouts: it scales acknowledge and retransmission times to match the network speed. This allows the protocol to perform well on both the Internet and on local Ethernets. Also, IL does no blind retransmission, to avoid adding to the congestion of busy networks. Full details are in another paper [PrWi95].

In Plan 9, the implementation of IL is smaller and faster than TCP. IL is our main Internet transport protocol.

Overview of authentication

Authentication establishes the identity of a user accessing a resource. The user requesting the resource is called the *client* and the user granting access to the resource is called the *server*. This is usually done under the auspices of a 9P attach message. A user may be a client in one authentication exchange and a server in another. Servers always act on behalf of some user, either a normal client or some administrative entity, so authentication is defined to be between users, not machines.

Each Plan 9 user has an associated DES [NBS77] authentication key; the user's identity is verified by the ability to encrypt and decrypt special messages called challenges. Since knowledge of a user's key gives access to that user's resources, the Plan 9 authentication protocols never transmit a message containing a cleartext key.

Authentication is bilateral: at the end of the authentication exchange, each side is convinced of the other's identity. Every machine begins the exchange with a DES key in memory. In the case of CPU and file servers, the key, user name, and domain name for the server are read from permanent storage, usually non-volatile RAM. In the case of terminals, the key is derived from a password typed by the user at boot time. A special machine, known as the *authentication server*, maintains a database of keys for all users in its administrative domain and participates in the authentication protocols.

The authentication protocol is as follows: after exchanging challenges, one party contacts the authentication server to create permission-granting *tickets* encrypted with each party's secret key and containing a new conversation key. Each party decrypts its own ticket and uses the conversation key to encrypt the other party's challenge.

This structure is somewhat like Kerberos [MBSS87], but avoids its reliance on synchronized clocks. Also unlike Kerberos, Plan 9 authentication supports a 'speaks for' relation [LABW91] that enables one user to have the authority of another; this is how a CPU server runs processes on behalf of its clients.

Plan 9's authentication structure builds secure services rather than depending on firewalls. Whereas firewalls require special code for every service penetrating the wall, the Plan 9 approach permits authentication to be done in a single place—9P—for all services. For example, the `cpu` command works securely across the Internet.

Authenticating external connections

The regular Plan 9 authentication protocol is not suitable for text-based services such as Telnet or FTP. In such cases, Plan 9 users authenticate with hand-held DES calculators called *authenticators*. The authenticator holds a key for the user, distinct from the user's normal authentication key. The user 'logs on' to the authenticator using a 4-digit PIN. A correct PIN enables the authenticator for a challenge/response exchange with the server. Since a correct challenge/response exchange is valid only once and keys are never sent over the network, this procedure is not susceptible to replay attacks, yet is compatible with protocols like Telnet and FTP.

Special users

Plan 9 has no super-user. Each server is responsible for maintaining its own security, usually permitting access only from the console, which is protected by a password. For example, file servers have a unique administrative user called `adm`, with special privileges that apply only to commands typed at the server's physical console. These privileges concern the day-to-day maintenance of the server, such as adding new users and configuring disks and networks. The privileges do *not* include the ability to modify, examine, or change the permissions of any files. If a file is read-protected by a user, only that user may grant access to others.

CPU servers have an equivalent user name that allows administrative access to resources on that server such as the control files of user processes. Such permission is necessary, for example, to kill rogue processes, but does not extend beyond that server. On the other hand, by means of a key held in protected non-volatile RAM, the identity of the administrative user is proven to the authentication server. This allows the CPU server to authenticate remote users, both for access to the server itself and when the CPU server is acting as a proxy on their behalf.

Finally, a special user called `none` has no password and is always allowed to connect; anyone may claim to be `none`. `None` has restricted permissions; for example, it is not allowed to examine dump files and can read only world-readable files.

The idea behind `none` is analogous to the anonymous user in FTP services. On Plan 9, guest FTP servers are further confined within a special restricted name space. It disconnects guest users from system programs, such as the contents of `/bin`, but makes it possible to make local files available to guests by binding them explicitly into the space. A restricted name space is more secure than the usual technique of exporting an ad hoc directory tree; the result is a kind of cage around untrusted users.

The cpu command and proxied authentication

When a call is made to a CPU server for a user, say Peter, the intent is that Peter wishes to run processes with his own authority. To implement this property, the CPU server does the following when the call is received. First, the listener forks off a process to handle the call. This process changes to the user `none` to avoid giving away permissions if it is compromised. It then performs the authentication protocol to verify that the calling user really is Peter, and to prove to Peter that the machine is itself trustworthy. Finally, it reattaches to all relevant file servers using the authentication protocol to identify itself as Peter. In this case, the CPU server is a client of the file server and performs the client portion of the authentication exchange on behalf of Peter. The authentication server will give the process tickets to accomplish this only if the CPU server's administrative user name is allowed to *speak for* Peter.

The *speaks for* relation [LABW91] is kept in a table on the authentication server. To simplify the management of users computing in different authentication domains, it also contains mappings between user names in different domains, for example saying that user `rtm` in one domain is the same person as user `rtmorris` in another.

File Permissions

One of the advantages of constructing services as file systems is that the solutions to ownership and permission problems fall out naturally. As in UNIX, each file or directory has separate read, write, and execute/search permissions for the file's owner, the file's group, and anyone else. The idea of group is unusual: any user name is potentially a group name. A group is just a user with a list of other users in the group. Conventions make the distinction: most people have user names without group members, while groups have long lists of attached names. For example, the `sys` group traditionally has all the system programmers, and system files are accessible by group `sys`. Consider

the following two lines of a user database stored on a server:

```
pjw:pjw:
sys::pjw,ken,philw,presotto
```

The first establishes user `pjw` as a regular user. The second establishes user `sys` as a group and lists four users who are *members* of that group. The empty colon-separated field is space for a user to be named as the *group leader*. If a group has a leader, that user has special permissions for the group, such as freedom to change the group permissions of files in that group. If no leader is specified, each member of the group is considered equal, as if each were the leader. In our example, only `pjw` can add members to his group, but all of `sys`'s members are equal partners in that group.

Regular files are owned by the user that creates them. The group name is inherited from the directory holding the new file. Device files are treated specially: the kernel may arrange the ownership and permissions of a file appropriate to the user accessing the file.

A good example of the generality this offers is process files, which are owned and read-protected by the owner of the process. If the owner wants to let someone else access the memory of a process, for example to let the author of a program debug a broken image, the standard `chmod` command applied to the process files does the job.

Another unusual application of file permissions is the dump file system, which is not only served by the same file server as the original data, but represented by the same user database. Files in the dump are therefore given identical protection as files in the regular file system; if a file is owned by `pjw` and read-protected, once it is in the dump file system it is still owned by `pjw` and read-protected. Also, since the dump file system is immutable, the file cannot be changed; it is read-protected forever. Drawbacks are that if the file is readable but should have been read-protected, it is readable forever, and that user names are hard to re-use.

Performance

As a simple measure of the performance of the Plan 9 kernel, we compared the time to do some simple operations on Plan 9 and on SGI's IRIX Release 5.3 running on an SGI Challenge M with a 100MHz MIPS R4400 and a 1-megabyte secondary cache. The test program was written in Alef, compiled with the same compiler, and run on identical hardware, so the only variables are the operating system and libraries.

The program tests the time to do a context switch (`rendezvous` on Plan 9, `blockproc` on IRIX); a trivial system call (`rfork(0)` and `nap(0)`); and lightweight fork (`rfork(RFPROC)` and `sproc(PR_SFDS|PR_SADDR)`). It also measures the time to send a byte on a pipe from one process to another and the throughput on a pipe between two processes. The results appear in Table 1.

Test	Plan 9	IRIX
Context switch	39 µs	150 µs
System call	6 µs	36 µs
Light fork	1300 µs	2200 µs
Pipe latency	110 µs	200 µs
Pipe bandwidth	11678 KB/s	14545 KB/s

Table 1. Performance comparison.

Although the Plan 9 times are not spectacular, they show that the kernel is competitive with commercial systems.

Discussion

Plan 9 has a relatively conventional kernel; the system's novelty lies in the pieces outside the kernel and the way they interact. When building Plan 9, we considered all aspects of the system together, solving problems where the solution fit best. Sometimes the solution spanned many components. An example is the problem of heterogeneous instruction architectures, which is addressed by the compilers (different code characters, portable object code), the environment (`$cputype` and `$objtype`), the name space (binding in `/bin`), and other components. Sometimes many issues could be solved in a single place. The best example is 9P, which centralizes naming, access, and authentication. 9P is really the core of the system; it is fair to say that the Plan 9 kernel is primarily a 9P multiplexer.

Plan 9's focus on files and naming is central to its expressiveness. Particularly in distributed computing, the way things are named has profound influence on the system [Nee89]. The combination of local name spaces and global conventions to interconnect networked resources avoids the difficulty of maintaining a global uniform name space, while naming everything like a file makes the system easy to understand, even for novices. Consider the dump file system, which is trivial to use for anyone familiar with hierarchical file systems. At a deeper level, building all the resources above a single uniform interface makes interoperability easy. Once a resource exports a 9P interface, it can combine transparently with any other part of the system to build unusual applications; the details are hidden. This may sound object-oriented, but there are distinctions. First, 9P defines a fixed set of 'methods'; it is not an extensible protocol. More important, files are well-defined and well-understood and come prepackaged with familiar methods of access, protection, naming, and networking. Objects, despite their generality, do not come with these attributes defined. By reducing 'object' to 'file', Plan 9 gets some technology for free.

Nonetheless, it is possible to push the idea of file-based computing too far. Converting every resource in the system into a file system is a kind of metaphor, and metaphors can be abused. A good example of restraint is `/proc`, which is only a view of a process, not a representation. To run processes, the usual `fork` and `exec` calls are still necessary, rather than doing something like

```
cp /bin/date /proc/clone/mem
```

The problem with such examples is that they require the server to do things not under its control. The ability to assign meaning to a command like this does not imply the meaning will fall naturally out of the structure of answering the 9P requests it generates. As a related example, Plan 9 does not put machine's network names in the file name space. The network interfaces provide a very different model of naming, because using `open`, `create`, `read`, and `write` on such files would not offer a suitable place to encode all the details of call setup for an arbitrary network. This does not mean that the network interface cannot be file-like, just that it must have a more tightly defined structure.

What would we do differently next time? Some elements of the implementation are unsatisfactory. Using streams to implement network interfaces in the kernel allows protocols to be connected together dynamically, such as to attach the same TTY driver to TCP, URP, and IL connections, but Plan 9 makes no use of this configurability. (It was exploited, however, in the research UNIX system for which streams were invented.) Replacing streams by static I/O queues would simplify the code and make it faster.

Although the main Plan 9 kernel is portable across many machines, the file server is implemented separately. This has caused several problems: drivers that must be written twice, bugs that must be fixed twice, and weaker portability of the file system code. The solution is easy: the file server kernel should be maintained as a variant of the regular operating system, with no user processes and special compiled-in kernel processes

to implement file service. Another improvement to the file system would be a change of internal structure. The WORM jukebox is the least reliable piece of the hardware, but because it holds the metadata of the file system, it must be present in order to serve files. The system could be restructured so the WORM is a backup device only, with the file system proper residing on magnetic disks. This would require no change to the external interface.

Although Plan 9 has per-process name spaces, it has no mechanism to give the description of a process's name space to another process except by direct inheritance. The `cpu` command, for example, cannot in general reproduce the terminal's name space; it can only re-interpret the user's login profile and make substitutions for things like the name of the binary directory to load. This misses any local modifications made before running `cpu`. It should instead be possible to capture the terminal's name space and transmit its description to a remote process.

Despite these problems, Plan 9 works well. It has matured into the system that supports our research, rather than being the subject of the research itself. Experimental new work includes developing interfaces to faster networks, file caching in the client kernel, encapsulating and exporting name spaces, and the ability to re-establish the client state after a server crash. Attention is now focusing on using the system to build distributed applications.

One reason for Plan 9's success is that we use it for our daily work, not just as a research tool. Active use forces us to address shortcomings as they arise and to adapt the system to solve our problems. Through this process, Plan 9 has become a comfortable, productive programming environment, as well as a vehicle for further systems research.

References

[9man] *Plan 9 Programmer's Manual, Volume 1,* AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[ANSIC] *American National Standard for Information Systems – Programming Language C*, American National Standards Institute, Inc., New York, 1990.

[Duff90] Tom Duff, ''Rc – A Shell for Plan 9 and UNIX systems'', *Proc. of the Summer 1990 UKUUG Conf.,* London, July, 1990, pp. 21–33, reprinted, in a different form, in this volume.

[Fra80] A.G. Fraser, ''Datakit – A Modular Network for Synchronous and Asynchronous Traffic'', *Proc. Int. Conf. on Commun.,* June 1980, Boston, MA.

[FSSUTF] *File System Safe UCS Transformation Format (FSS–UTF),* X/Open Preliminary Specification, 1993. ISO designation is ISO/IEC JTC1/SC2/WG2 N 1036, dated 1994–08–01.

[ISO10646] ISO/IEC DIS 10646–1:1993 *Information technology – Universal Multiple–Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

[Kill84] T.J. Killian, ''Processes as Files'', *USENIX Summer 1984 Conf. Proc.,* June 1984, Salt Lake City, UT.

[LABW91] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber, ''Authentication in Distributed Systems: Theory and Practice'', *Proc. 13th ACM Symp. on Op. Sys. Princ.,* Asilomar, 1991, pp. 165–182.

[MBSS87] S. P. Miller, B. C. Neumann, J. I. Schiller, and J. H. Saltzer, ''Kerberos Authentication and Authorization System'', Massachusetts Institute of Technology, 1987.

[NBS77] National Bureau of Standards (U.S.), *Federal Information Processing Standard 46,* National Technical Information Service, Springfield, VA, 1977.

[Nee89] R. Needham, ''Names'', in *Distributed systems,* S. Mullender, ed., Addison Wesley, 1989

[NeHe82] R.M. Needham and A.J. Herbert, *The Cambridge Distributed Computing System,* Addison–Wesley, London, 1982

[Neu92] B. Clifford Neuman, ''The Prospero File System'', *USENIX File Systems Workshop Proc.,* Ann Arbor, 1992, pp. 13–28.

[OCDNW88] John Ousterhout, Andrew Cherenson, Fred Douglis, Mike Nelson, and Brent Welch, ''The Sprite Network Operating System'', *IEEE Computer,* 21(2), 23–38, Feb. 1988.

[Pike87] Rob Pike, ''The Text Editor sam'', *Software – Practice and Experience,* Nov 1987, **17**(11), pp. 813–845; reprinted in this volume.

[Pike91] Rob Pike, ''8½, the Plan 9 Window System'', *USENIX Summer Conf. Proc.,* Nashville, June, 1991, pp. 257–265, reprinted in this volume.

[Pike93] Rob Pike and Ken Thompson, ''Hello World or Καλημέρα κόσμε or こんにちは 世界'', *USENIX Winter Conf. Proc.,* San Diego, 1993, pp. 43–50, reprinted in this volume.

[Pike94] Rob Pike, ''Acme: A User Interface for Programmers'', *USENIX Proc. of the Winter 1994 Conf.,* San Francisco, CA,

[Pike95] Rob Pike, ''How to Use the Plan 9 C Compiler'', *Plan 9 Programmer's Manual, Volume 2,* AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[POSIX] *Information Technology—Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language],* IEEE, New York, 1990.

[PPTTW93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, ''The Use of Name Spaces in Plan 9'', *Op. Sys. Rev.,* Vol. 27, No. 2, April 1993, pp. 72–76, reprinted in this volume.

[Presotto]Dave Presotto, ''Multiprocessor Streams for Plan 9'', *UKUUG Summer 1990 Conf. Proc.,* July 1990, pp. 11–19.

[PrWi93] Dave Presotto and Phil Winterbottom, ''The Organization of Networks in Plan 9'', *USENIX Proc. of the Winter 1993 Conf.,* San Diego, CA, pp. 43–50, reprinted in this volume.

[PrWi95] Dave Presotto and Phil Winterbottom, ''The IL Protocol'', *Plan 9 Programmer's Manual, Volume 2,* AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[RFC768] J. Postel, RFC768, *User Datagram Protocol, DARPA Internet Program Protocol Specification,* August 1980.

[RFC793] RFC793, *Transmission Control Protocol, DARPA Internet Program Protocol Specification,* September 1981.

[Rao91] Herman Chung-Hwa Rao, *The Jade File System,* (Ph. D. Dissertation), Dept. of Comp. Sci, University of Arizona, TR 91–18.

[Rit84] D.M. Ritchie, ''A Stream Input-Output System'', *AT&T Bell Laboratories Technical Journal,* **63**(8), October, 1984.

[Tric95] Howard Trickey, ''APE — The ANSI/POSIX Environment'', *Plan 9 Programmer's Manual, Volume 2,* AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[Unicode]*The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volume 1,* The Unicode Consortium, Addison Wesley, New York, 1991.

[UNIX85] *UNIX Time-Sharing System Programmer's Manual, Research Version, Eighth Edition, Volume 1.* AT&T Bell Laboratories, Murray Hill, NJ, 1985.

[Welc94] Brent Welch, ''A Comparison of Three Distributed File System Architectures: Vnode, Sprite, and Plan 9'', *Computing Systems,* 7(2), pp. 175–199, Spring, 1994.

[Wint95] Phil Winterbottom, ''Alef Language Reference Manual'', *Plan 9 Programmer's Manual, Volume 2,* AT&T Bell Laboratories, Murray Hill, NJ, 1995.

The Use of Name Spaces in Plan 9

Rob Pike
Dave Presotto
Ken Thompson
Howard Trickey
Phil Winterbottom

Bell Laboratories
Murray Hill, New Jersey 07974
USA

ABSTRACT

Plan 9 is a distributed system built at the Computing Sciences Research Center of AT&T Bell Laboratories (now Lucent Technologies, Bell Labs) over the last few years. Its goal is to provide a production-quality system for software development and general computation using heterogeneous hardware and minimal software. A Plan 9 system comprises CPU and file servers in a central location connected together by fast networks. Slower networks fan out to workstation-class machines that serve as user terminals. Plan 9 argues that given a few carefully implemented abstractions it is possible to produce a small operating system that provides support for the largest systems on a variety of architectures and networks. The foundations of the system are built on two ideas: a per-process name space and a simple message-oriented file system protocol.

The operating system for the CPU servers and terminals is structured as a traditional kernel: a single compiled image containing code for resource management, process control, user processes, virtual memory, and I/O. Because the file server is a separate machine, the file system is not compiled in, although the management of the name space, a per-process attribute, is. The entire kernel for the multiprocessor SGI Power Series machine is 25000 lines of C, the largest part of which is code for four networks including the Ethernet with the Internet protocol suite. Fewer than 1500 lines are machine-specific, and a functional kernel with minimal I/O can be put together from source files totaling 6000 lines. [Pike90]

The system is relatively small for several reasons. First, it is all new: it has not had time to accrete as many fixes and features as other systems. Also, other than the network protocol, it adheres to no external interface; in particular, it is not Unix-compatible. Economy stems from careful selection of services and interfaces. Finally, wherever possible the system is built around two simple ideas: every resource in the system, either local or remote, is represented by a hierarchical file system; and a user or process assembles a private view of the system by constructing a file *name space* that connects these resources. [Needham]

Appeared in *Operating Systems Review,* Vol. 27, #2, April 1993, pp. 72-76 (reprinted from *Proceedings of the 5th ACM SIGOPS European Workshop,* Mont Saint-Michel, 1992, Paper n° 34).

File Protocol

All resources in Plan 9 look like file systems. That does not mean that they are repositories for permanent files on disk, but that the interface to them is file-oriented: finding files (resources) in a hierarchical name tree, attaching to them by name, and accessing their contents by read and write calls. There are dozens of file system types in Plan 9, but only a few represent traditional files. At this level of abstraction, files in Plan 9 are similar to objects, except that files are already provided with naming, access, and protection methods that must be created afresh for objects. Object-oriented readers may approach the rest of this paper as a study in how to make objects look like files.

The interface to file systems is defined by a protocol, called 9P, analogous but not very similar to the NFS protocol. The protocol talks about files, not blocks; given a connection to the root directory of a file server, the 9P messages navigate the file hierarchy, open files for I/O, and read or write arbitrary bytes in the files. 9P contains 17 message types: three for initializing and authenticating a connection and fourteen for manipulating objects. The messages are generated by the kernel in response to user- or kernel-level I/O requests. Here is a quick tour of the major message types. The `auth` and `attach` messages authenticate a connection, established by means outside 9P, and validate its user. The result is an authenticated *channel* that points to the root of the server. The `clone` message makes a new channel identical to an existing channel, which may be moved to a file on the server using a `walk` message to descend each level in the hierarchy. The `stat` and `wstat` messages read and write the attributes of the file pointed to by a channel. The `open` message prepares a channel for subsequent `read` and `write` messages to access the contents of the file, while `create` and `remove` perform, on the files, the actions implied by their names. The `clunk` message discards a channel without affecting the file. None of the 9P messages consider caching; file caches are provided, when needed, either within the server (centralized caching) or by implementing the cache as a transparent file system between the client and the 9P connection to the server (client caching).

For efficiency, the connection to local kernel-resident file systems, misleadingly called *devices,* is by regular rather than remote procedure calls. The procedures map one-to-one with 9P message types. Locally each channel has an associated data structure that holds a type field used to index a table of procedure calls, one set per file system type, analogous to selecting the method set for an object. One kernel-resident file system, the *mount device,* translates the local 9P procedure calls into RPC messages to remote services over a separately provided transport protocol such as TCP or IL, a new reliable datagram protocol, or over a pipe to a user process. Write and read calls transmit the messages over the transport layer. The mount device is the sole bridge between the procedural interface seen by user programs and remote and user-level services. It does all associated marshaling, buffer management, and multiplexing and is the only integral RPC mechanism in Plan 9. The mount device is in effect a proxy object. There is no RPC stub compiler; instead the mount driver and all servers just share a library that packs and unpacks 9P messages.

Examples

One file system type serves permanent files from the main file server, a stand-alone multiprocessor system with a 350-gigabyte optical WORM jukebox that holds the data, fronted by a two-level block cache comprising 7 gigabytes of magnetic disk and 128 megabytes of RAM. Clients connect to the file server using any of a variety of networks and protocols and access files using 9P. The file server runs a distinct operating system and has no support for user processes; other than a restricted set of commands available on the console, all it does is answer 9P messages from clients.

Once a day, at 5:00 AM, the file server sweeps through the cache blocks and marks dirty blocks copy-on-write. It creates a copy of the root directory and labels it with the

current date, for example `1995/0314`. It then starts a background process to copy the dirty blocks to the WORM. The result is that the server retains an image of the file system as it was early each morning. The set of old root directories is accessible using 9P, so a client may examine backup files using ordinary commands. Several advantages stem from having the backup service implemented as a plain file system. Most obviously, ordinary commands can access them. For example, to see when a bug was fixed

```
grep 'mouse bug fix' 1995/*/sys/src/cmd/8½/file.c
```

The owner, access times, permissions, and other properties of the files are also backed up. Because it is a file system, the backup still has protections; it is not possible to subvert security by looking at the backup.

The file server is only one type of file system. A number of unusual services are provided within the kernel as local file systems. These services are not limited to I/O devices such as disks. They include network devices and their associated protocols, the bitmap display and mouse, a representation of processes similar to `/proc` [Killian], the name/value pairs that form the 'environment' passed to a new process, profiling services, and other resources. Each of these is represented as a file system — directories containing sets of files — but the constituent files do not represent permanent storage on disk. Instead, they are closer in properties to UNIX device files.

For example, the *console* device contains the file `/dev/cons`, similar to the UNIX file `/dev/console`: when written, `/dev/cons` appends to the console typescript; when read, it returns characters typed on the keyboard. Other files in the console device include `/dev/time`, the number of seconds since the epoch, `/dev/cputime`, the computation time used by the process reading the device, `/dev/pid`, the process id of the process reading the device, and `/dev/user`, the login name of the user accessing the device. All these files contain text, not binary numbers, so their use is free of byte-order problems. Their contents are synthesized on demand when read; when written, they cause modifications to kernel data structures.

The *process* device contains one directory per live local process, named by its numeric process id: `/proc/1`, `/proc/2`, etc. Each directory contains a set of files that access the process. For example, in each directory the file `mem` is an image of the virtual memory of the process that may be read or written for debugging. The `text` file is a sort of link to the file from which the process was executed; it may be opened to read the symbol tables for the process. The `ctl` file may be written textual messages such as `stop` or `kill` to control the execution of the process. The `status` file contains a fixed-format line of text containing information about the process: its name, owner, state, and so on. Text strings written to the `note` file are delivered to the process as *notes,* analogous to UNIX signals. By providing these services as textual I/O on files rather than as system calls (such as `kill`) or special-purpose operations (such as `ptrace`), the Plan 9 process device simplifies the implementation of debuggers and related programs. For example, the command

```
cat /proc/*/status
```

is a crude form of the `ps` command; the actual `ps` merely reformats the data so obtained.

The *bitmap* device contains three files, `/dev/mouse`, `/dev/screen`, and `/dev/bitblt`, that provide an interface to the local bitmap display (if any) and pointing device. The `mouse` file returns a fixed-format record containing 1 byte of button state and 4 bytes each of *x* and *y* position of the mouse. If the mouse has not moved since the file was last read, a subsequent read will block. The `screen` file contains a memory image of the contents of the display; the `bitblt` file provides a procedural interface. Calls to the graphics library are translated into messages that are written to the `bitblt` file to perform bitmap graphics operations. (This is essentially a nested

RPC protocol.)

The various services being used by a process are gathered together into the process's *name space,* a single rooted hierarchy of file names. When a process forks, the child process shares the name space with the parent. Several system calls manipulate name spaces. Given a file descriptor `fd` that holds an open communications channel to a service, the call

```
mount(int fd, char *old, int flags)
```

authenticates the user and attaches the file tree of the service to the directory named by `old`. The `flags` specify how the tree is to be attached to `old`: replacing the current contents or appearing before or after the current contents of the directory. A directory with several services mounted is called a *union* directory and is searched in the specified order. The call

```
bind(char *new, char *old, int flags)
```

takes the portion of the existing name space visible at `new`, either a file or a directory, and makes it also visible at `old`. For example,

```
bind("1995/0301/sys/include", "/sys/include", REPLACE)
```

causes the directory of include files to be overlaid with its contents from the dump on March first.

A process is created by the `rfork` system call, which takes as argument a bit vector defining which attributes of the process are to be shared between parent and child instead of copied. One of the attributes is the name space: when shared, changes made by either process are visible in the other; when copied, changes are independent.

Although there is no global name space, for a process to function sensibly the local name spaces must adhere to global conventions. Nonetheless, the use of local name spaces is critical to the system. Both these ideas are illustrated by the use of the name space to handle heterogeneity. The binaries for a given architecture are contained in a directory named by the architecture, for example `/mips/bin`; in use, that directory is bound to the conventional location `/bin`. Programs such as shell scripts need not know the CPU type they are executing on to find binaries to run. A directory of private binaries is usually unioned with `/bin`. (Compare this to the *ad hoc* and special-purpose idea of the `PATH` variable, which is not used in the Plan 9 shell.) Local bindings are also helpful for debugging, for example by binding an old library to the standard place and linking a program to see if recent changes to the library are responsible for a bug in the program.

The window system, 8½ [Pike91], is a server for files such as `/dev/cons` and `/dev/bitblt`. Each client sees a distinct copy of these files in its local name space: there are many instances of `/dev/cons`, each served by 8½ to the local name space of a window. Again, 8½ implements services using local name spaces plus the use of I/O to conventionally named files. Each client just connects its standard input, output, and error files to `/dev/cons`, with analogous operations to access bitmap graphics. Compare this to the implementation of `/dev/tty` on UNIX, which is done by special code in the kernel that overloads the file, when opened, with the standard input or output of the process. Special arrangement must be made by a UNIX window system for `/dev/tty` to behave as expected; 8½ instead uses the provision of the corresponding file as its central idea, which to succeed depends critically on local name spaces.

The environment 8½ provides its clients is exactly the environment under which it is implemented: a conventional set of files in `/dev`. This permits the window system to be run recursively in one of its own windows, which is handy for debugging. It also means that if the files are exported to another machine, as described below, the window system or client applications may be run transparently on remote machines, even ones

without graphics hardware. This mechanism is used for Plan 9's implementation of the X window system: X is run as a client of 8½, often on a remote machine with lots of memory. In this configuration, using Ethernet to connect MIPS machines, we measure only a 10% degradation in graphics performance relative to running X on a bare Plan 9 machine.

An unusual application of these ideas is a statistics-gathering file system implemented by a command called `iostats`. The command encapsulates a process in a local name space, monitoring 9P requests from the process to the outside world — the name space in which `iostats` is itself running. When the command completes, `iostats` reports usage and performance figures for file activity. For example

```
iostats 8½
```

can be used to discover how much I/O the window system does to the bitmap device, font files, and so on.

The `import` command connects a piece of name space from a remote system to the local name space. Its implementation is to dial the remote machine and start a process there that serves the remote name space using 9P. It then calls `mount` to attach the connection to the name space and finally dies; the remote process continues to serve the files. One use is to access devices not available locally. For example, to write a floppy one may say

```
import lab.pc /a: /n/dos
cp foo /n/dos/bar
```

The call to `import` connects the file tree from `/a:` on the machine `lab.pc` (which must support 9P) to the local directory `/n/dos`. Then the file `foo` can be written to the floppy just by copying it across.

Another application is remote debugging:

```
import helix /proc
```

makes the process file system on machine `helix` available locally; commands such as `ps` then see `helix`'s processes instead of the local ones. The debugger may then look at a remote process:

```
db /proc/27/text /proc/27/mem
```

allows breakpoint debugging of the remote process. Since `db` infers the CPU type of the process from the executable header on the text file, it supports cross-architecture debugging, too. Care is taken within `db` to handle issues of byte order and floating point; it is possible to breakpoint debug a big-endian MIPS process from a little-endian i386.

Network interfaces are also implemented as file systems [Presotto]. For example, `/net/tcp` is a directory somewhat like `/proc`: it contains a set of numbered directories, one per connection, each of which contains files to control and communicate on the connection. A process allocates a new connection by accessing `/net/tcp/clone`, which evaluates to the directory of an unused connection. To make a call, the process writes a textual message such as `'connect 135.104.53.2!512'` to the `ctl` file and then reads and writes the `data` file. An `rlogin` service can be implemented in a few of lines of shell code.

This structure makes network gatewaying easy to provide. We have machines with Datakit interfaces but no Internet interface. On such a machine one may type

```
import helix /net
telnet tcp!ai.mit.edu
```

The `import` uses Datakit to pull in the TCP interface from `helix`, which can then be used directly; the `tcp!` notation is necessary because we routinely use multiple

networks and protocols on Plan 9—it identifies the network in which `ai.mit.edu` is a valid name.

In practice we do not use `rlogin` or `telnet` between Plan 9 machines. Instead a command called `cpu` in effect replaces the CPU in a window with that on another machine, typically a fast multiprocessor CPU server. The implementation is to recreate the name space on the remote machine, using the equivalent of `import` to connect pieces of the terminal's name space to that of the process (shell) on the CPU server, making the terminal a file server for the CPU. CPU-local devices such as fast file system connections are still local; only terminal-resident devices are imported. The result is unlike UNIX `rlogin`, which moves into a distinct name space on the remote machine, or file sharing with `NFS`, which keeps the name space the same but forces processes to execute locally. Bindings in `/bin` may change because of a change in CPU architecture, and the networks involved may be different because of differing hardware, but the effect feels like simply speeding up the processor in the current name space.

Position

These examples illustrate how the ideas of representing resources as file systems and per-process name spaces can be used to solve problems often left to more exotic mechanisms. Nonetheless there are some operations in Plan 9 that are not mapped into file I/O. An example is process creation. We could imagine a message to a control file in `/proc` that creates a process, but the details of constructing the environment of the new process — its open files, name space, memory image, etc. — are too intricate to be described easily in a simple I/O operation. Therefore new processes on Plan 9 are created by fairly conventional `rfork` and `exec` system calls; `/proc` is used only to represent and control existing processes.

Plan 9 does not attempt to map network name spaces into the file system name space, for several reasons. The different addressing rules for various networks and protocols cannot be mapped uniformly into a hierarchical file name space. Even if they could be, the various mechanisms to authenticate, select a service, and control the connection would not map consistently into operations on a file.

Shared memory is another resource not adequately represented by a file name space. Plan 9 takes care to provide mechanisms to allow groups of local processes to share and map memory. Memory is controlled by system calls rather than special files, however, since a representation in the file system would imply that memory could be imported from remote machines.

Despite these limitations, file systems and name spaces offer an effective model around which to build a distributed system. Used well, they can provide a uniform, familiar, transparent interface to a diverse set of distributed resources. They carry well-understood properties of access, protection, and naming. The integration of devices into the hierarchical file system was the best idea in UNIX. Plan 9 pushes the concepts much further and shows that file systems, when used inventively, have plenty of scope for productive research.

References

[Killian] T. Killian, ''Processes as Files'', USENIX Summer Conf. Proc., Salt Lake City, 1984
[Needham] R. Needham, ''Names'', in *Distributed systems,* S. Mullender, ed., Addison Wesley, 1989
[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, ''Plan 9 from Bell Labs'', UKUUG Proc. of the Summer 1990 Conf., London, England, 1990
[Presotto] D. Presotto, ''Multiprocessor Streams for Plan 9'', UKUUG Proc. of the Summer 1990 Conf., London, England, 1990
[Pike91] Pike, R., ''8.5, The Plan 9 Window System'', USENIX Summer Conf. Proc., Nashville, 1991

The Organization of Networks in Plan 9

Dave Presotto
Phil Winterbottom

presotto,philw@plan9.bell-labs.com

ABSTRACT

In a distributed system networks are of paramount importance. This paper describes the implementation, design philosophy, and organization of network support in Plan 9. Topics include network requirements for distributed systems, our kernel implementation, network naming, user interfaces, and performance. We also observe that much of this organization is relevant to current systems.

1. Introduction

Plan 9 [Pike90] is a general-purpose, multi-user, portable distributed system implemented on a variety of computers and networks. What distinguishes Plan 9 is its organization. The goals of this organization were to reduce administration and to promote resource sharing. One of the keys to its success as a distributed system is the organization and management of its networks.

A Plan 9 system comprises file servers, CPU servers and terminals. The file servers and CPU servers are typically centrally located multiprocessor machines with large memories and high speed interconnects. A variety of workstation-class machines serve as terminals connected to the central servers using several networks and protocols. The architecture of the system demands a hierarchy of network speeds matching the needs of the components. Connections between file servers and CPU servers are high-bandwidth point-to-point fiber links. Connections from the servers fan out to local terminals using medium speed networks such as Ethernet [Met80] and Datakit [Fra80]. Low speed connections via the Internet and the AT&T backbone serve users in Oregon and Illinois. Basic Rate ISDN data service and 9600 baud serial lines provide slow links to users at home.

Since CPU servers and terminals use the same kernel, users may choose to run programs locally on their terminals or remotely on CPU servers. The organization of Plan 9 hides the details of system connectivity allowing both users and administrators to configure their environment to be as distributed or centralized as they wish. Simple commands support the construction of a locally represented name space spanning many machines and networks. At work, users tend to use their terminals like workstations, running interactive programs locally and reserving the CPU servers for data or compute intensive jobs such as compiling and computing chess endgames. At home or when connected over a slow network, users tend to do most work on the CPU server to minimize traffic on the slow links. The goal of the network organization is to provide the same environment to the user wherever resources are used.

Originally appeared in *Proc. of the Winter 1993 USENIX Conf.,* pp. 271-280, San Diego, CA

2. Kernel Network Support

Networks play a central role in any distributed system. This is particularly true in Plan 9 where most resources are provided by servers external to the kernel. The importance of the networking code within the kernel is reflected by its size; of 25,000 lines of kernel code, 12,500 are network and protocol related. Networks are continually being added and the fraction of code devoted to communications is growing. Moreover, the network code is complex. Protocol implementations consist almost entirely of synchronization and dynamic memory management, areas demanding subtle error recovery strategies. The kernel currently supports Datakit, point-to-point fiber links, an Internet (IP) protocol suite and ISDN data service. The variety of networks and machines has raised issues not addressed by other systems running on commercial hardware supporting only Ethernet or FDDI.

2.1. The File System protocol

A central idea in Plan 9 is the representation of a resource as a hierarchical file system. Each process assembles a view of the system by building a *name space* [Needham] connecting its resources. File systems need not represent disc files; in fact, most Plan 9 file systems have no permanent storage. A typical file system dynamically represents some resource like a set of network connections or the process table. Communication between the kernel, device drivers, and local or remote file servers uses a protocol called 9P. The protocol consists of 17 messages describing operations on files and directories. Kernel resident device and protocol drivers use a procedural version of the protocol while external file servers use an RPC form. Nearly all traffic between Plan 9 systems consists of 9P messages. 9P relies on several properties of the underlying transport protocol. It assumes messages arrive reliably and in sequence and that delimiters between messages are preserved. When a protocol does not meet these requirements (for example, TCP does not preserve delimiters) we provide mechanisms to marshal messages before handing them to the system.

A kernel data structure, the *channel*, is a handle to a file server. Operations on a channel generate the following 9P messages. The `session` and `attach` messages authenticate a connection, established by means external to 9P, and validate its user. The result is an authenticated channel referencing the root of the server. The `clone` message makes a new channel identical to an existing channel, much like the `dup` system call. A channel may be moved to a file on the server using a `walk` message to descend each level in the hierarchy. The `stat` and `wstat` messages read and write the attributes of the file referenced by a channel. The `open` message prepares a channel for subsequent `read` and `write` messages to access the contents of the file. `Create` and `remove` perform the actions implied by their names on the file referenced by the channel. The `clunk` message discards a channel without affecting the file.

A kernel resident file server called the *mount driver* converts the procedural version of 9P into RPCs. The *mount* system call provides a file descriptor, which can be a pipe to a user process or a network connection to a remote machine, to be associated with the mount point. After a mount, operations on the file tree below the mount point are sent as messages to the file server. The mount driver manages buffers, packs and unpacks parameters from messages, and demultiplexes among processes using the file server.

2.2. Kernel Organization

The network code in the kernel is divided into three layers: hardware interface, protocol processing, and program interface. A device driver typically uses streams to connect the two interface layers. Additional stream modules may be pushed on a device to process protocols. Each device driver is a kernel-resident file system. Simple device drivers serve a single level directory containing just a few files; for example, we

represent each UART by a data and a control file.

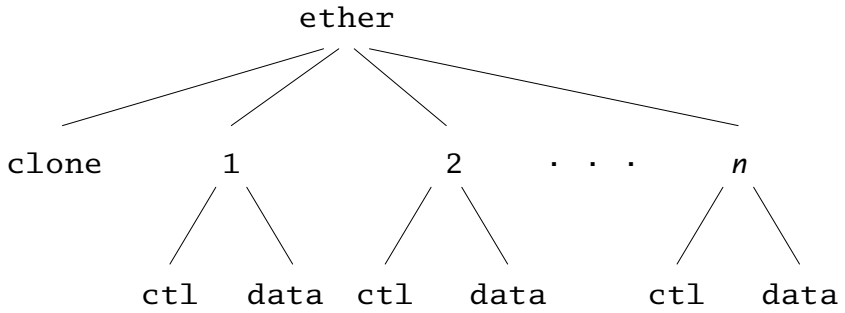
```
cpu% cd /dev
cpu% ls -l eia*
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia1
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia1ctl
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia2
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia2ctl
cpu%
```

The control file is used to control the device; writing the string `b1200` to `/dev/eia1ctl` sets the line to 1200 baud.

Multiplexed devices present a more complex interface structure. For example, the LANCE Ethernet driver serves a two level file tree (Figure 1) providing

- device control and configuration
- user-level protocols like ARP
- diagnostic interfaces for snooping software.

The top directory contains a `clone` file and a directory for each connection, numbered `1` to `n`. Each connection directory corresponds to an Ethernet packet type. Opening the `clone` file finds an unused connection directory and opens its `ctl` file. Reading the control file returns the ASCII connection number; the user process can use this value to construct the name of the proper connection directory. In each connection directory files named `ctl`, `data`, `stats`, and `type` provide access to the connection. Writing the string `connect 2048` to the `ctl` file sets the packet type to 2048 and configures the connection to receive all IP packets sent to the machine. Subsequent reads of the file `type` yield the string `2048`. The `data` file accesses the media; reading it returns the next packet of the selected type. Writing the file queues a packet for transmission after appending a packet header containing the source address and packet type. The `stats` file returns ASCII text containing the interface address, packet input/output counts, error statistics, and general information about the state of the interface.



If several connections on an interface are configured for a particular packet type, each receives a copy of the incoming packets. The special packet type `-1` selects all packets. Writing the strings `promiscuous` and `connect -1` to the `ctl` file configures a conversation to receive all packets on the Ethernet.

Although the driver interface may seem elaborate, the representation of a device as a set of files using ASCII strings for communication has several advantages. Any mechanism supporting remote access to files immediately allows a remote machine to use our interfaces as gateways. Using ASCII strings to control the interface avoids byte order problems and ensures a uniform representation for devices on the same machine and even allows devices to be accessed remotely. Representing dissimilar devices by the same set of files allows common tools to serve several networks or interfaces. Programs like `stty` are replaced by `echo` and shell redirection.

2.3. Protocol devices

Network connections are represented as pseudo-devices called protocol devices. Protocol device drivers exist for the Datakit URP protocol and for each of the Internet IP protocols TCP, UDP, and IL. IL, described below, is a new communication protocol used by Plan 9 for transmitting file system RPC's. All protocol devices look identical so user programs contain no network-specific code.

Each protocol device driver serves a directory structure similar to that of the Ethernet driver. The top directory contains a `clone` file and a directory for each connection numbered 0 to `n`. Each connection directory contains files to control one connection and to send and receive information. A TCP connection directory looks like this:

```
cpu% cd /net/tcp/2
cpu% ls -l
--rw-rw---- I 0 ehg    bootes 0 Jul 13 21:14 ctl
--rw-rw---- I 0 ehg    bootes 0 Jul 13 21:14 data
--rw-rw---- I 0 ehg    bootes 0 Jul 13 21:14 listen
--r--r--r-- I 0 bootes bootes 0 Jul 13 21:14 local
--r--r--r-- I 0 bootes bootes 0 Jul 13 21:14 remote
--r--r--r-- I 0 bootes bootes 0 Jul 13 21:14 status
cpu% cat local remote status
135.104.9.31 5012
135.104.53.11 564
tcp/2 1 Established connect
cpu%
```

The files `local`, `remote`, and `status` supply information about the state of the connection. The `data` and `ctl` files provide access to the process end of the stream implementing the protocol. The `listen` file is used to accept incoming calls from the network.

The following steps establish a connection.

1) The clone device of the appropriate protocol directory is opened to reserve an unused connection.
2) The file descriptor returned by the open points to the `ctl` file of the new connection. Reading that file descriptor returns an ASCII string containing the connection number.
3) A protocol/network specific ASCII address string is written to the `ctl` file.
4) The path of the `data` file is constructed using the connection number. When the `data` file is opened the connection is established.

A process can read and write this file descriptor to send and receive messages from the network. If the process opens the `listen` file it blocks until an incoming call is received. An address string written to the `ctl` file before the listen selects the ports or services the process is prepared to accept. When an incoming call is received, the open completes and returns a file descriptor pointing to the `ctl` file of the new connection. Reading the `ctl` file yields a connection number used to construct the path of the `data` file. A connection remains established while any of the files in the connection directory are referenced or until a close is received from the network.

2.4. Streams

A *stream* [Rit84a][Presotto] is a bidirectional channel connecting a physical or pseudo-device to user processes. The user processes insert and remove data at one end of the stream. Kernel processes acting on behalf of a device insert data at the other end. Asynchronous communications channels such as pipes, TCP conversations, Datakit conversations, and RS232 lines are implemented using streams.

A stream comprises a linear list of *processing modules*. Each module has both an upstream (toward the process) and downstream (toward the device) *put routine*. Calling the put routine of the module on either end of the stream inserts data into the stream. Each module calls the succeeding one to send data up or down the stream.

An instance of a processing module is represented by a pair of *queues*, one for each direction. The queues point to the put procedures and can be used to queue information traveling along the stream. Some put routines queue data locally and send it along the stream at some later time, either due to a subsequent call or an asynchronous event such as a retransmission timer or a device interrupt. Processing modules create helper kernel processes to provide a context for handling asynchronous events. For example, a helper kernel process awakens periodically to perform any necessary TCP retransmissions. The use of kernel processes instead of serialized run-to-completion service routines differs from the implementation of Unix streams. Unix service routines cannot use any blocking kernel resource and they lack a local long-lived state. Helper kernel processes solve these problems and simplify the stream code.

There is no implicit synchronization in our streams. Each processing module must ensure that concurrent processes using the stream are synchronized. This maximizes concurrency but introduces the possibility of deadlock. However, deadlocks are easily avoided by careful programming; to date they have not caused us problems.

Information is represented by linked lists of kernel structures called *blocks*. Each block contains a type, some state flags, and pointers to an optional buffer. Block buffers can hold either data or control information, i.e., directives to the processing modules. Blocks and block buffers are dynamically allocated from kernel memory.

2.4.1. User Interface

A stream is represented at user level as two files, `ctl` and `data`. The actual names can be changed by the device driver using the stream, as we saw earlier in the example of the UART driver. The first process to open either file creates the stream automatically. The last close destroys it. Writing to the `data` file copies the data into kernel blocks and passes them to the downstream put routine of the first processing module. A write of less than 32K is guaranteed to be contained by a single block. Concurrent writes to the same stream are not synchronized, although the 32K block size assures atomic writes for most protocols. The last block written is flagged with a delimiter to alert downstream modules that care about write boundaries. In most cases the first put routine calls the second, the second calls the third, and so on until the data is output. As a consequence, most data is output without context switching.

Reading from the `data` file returns data queued at the top of the stream. The read terminates when the read count is reached or when the end of a delimited block is encountered. A per stream read lock ensures only one process can read from a stream at a time and guarantees that the bytes read were contiguous bytes from the stream.

Like UNIX streams [Rit84a], Plan 9 streams can be dynamically configured. The stream system intercepts and interprets the following control blocks:

`push` *name* adds an instance of the processing module *name* to the top of the stream.

`pop` removes the top module of the stream.

`hangup` sends a hangup message up the stream from the device end.

Other control blocks are module-specific and are interpreted by each processing module as they pass.

The convoluted syntax and semantics of the UNIX `ioctl` system call convinced us to leave it out of Plan 9. Instead, `ioctl` is replaced by the `ctl` file. Writing to the `ctl` file is identical to writing to a `data` file except the blocks are of type *control*. A

processing module parses each control block it sees. Commands in control blocks are ASCII strings, so byte ordering is not an issue when one system controls streams in a name space implemented on another processor. The time to parse control blocks is not important, since control operations are rare.

2.4.2. Device Interface

The module at the downstream end of the stream is part of a device interface. The particulars of the interface vary with the device. Most device interfaces consist of an interrupt routine, an output put routine, and a kernel process. The output put routine stages data for the device and starts the device if it is stopped. The interrupt routine wakes up the kernel process whenever the device has input to be processed or needs more output staged. The kernel process puts information up the stream or stages more data for output. The division of labor among the different pieces varies depending on how much must be done at interrupt level. However, the interrupt routine may not allocate blocks or call a put routine since both actions require a process context.

2.4.3. Multiplexing

The conversations using a protocol device must be multiplexed onto a single physical wire. We push a multiplexer processing module onto the physical device stream to group the conversations. The device end modules on the conversations add the necessary header onto downstream messages and then put them to the module downstream of the multiplexer. The multiplexing module looks at each message moving up its stream and puts it to the correct conversation stream after stripping the header controlling the demultiplexing.

This is similar to the Unix implementation of multiplexer streams. The major difference is that we have no general structure that corresponds to a multiplexer. Each attempt to produce a generalized multiplexer created a more complicated structure and underlined the basic difficulty of generalizing this mechanism. We now code each multiplexer from scratch and favor simplicity over generality.

2.4.4. Reflections

Despite five year's experience and the efforts of many programmers, we remain dissatisfied with the stream mechanism. Performance is not an issue; the time to process protocols and drive device interfaces continues to dwarf the time spent allocating, freeing, and moving blocks of data. However the mechanism remains inordinately complex. Much of the complexity results from our efforts to make streams dynamically configurable, to reuse processing modules on different devices and to provide kernel synchronization to ensure data structures don't disappear under foot. This is particularly irritating since we seldom use these properties.

Streams remain in our kernel because we are unable to devise a better alternative. Larry Peterson's X-kernel [Pet89a] is the closest contender but doesn't offer enough advantage to switch. If we were to rewrite the streams code, we would probably statically allocate resources for a large fixed number of conversations and burn memory in favor of less complexity.

3. The IL Protocol

None of the standard IP protocols is suitable for transmission of 9P messages over an Ethernet or the Internet. TCP has a high overhead and does not preserve delimiters. UDP, while cheap, does not provide reliable sequenced delivery. Early versions of the system used a custom protocol that was efficient but unsatisfactory for internetwork transmission. When we implemented IP, TCP, and UDP we looked around for a suitable replacement with the following properties:

- Reliable datagram service with sequenced delivery
- Runs over IP
- Low complexity, high performance
- Adaptive timeouts

None met our needs so a new protocol was designed. IL is a lightweight protocol designed to be encapsulated by IP. It is a connection-based protocol providing reliable transmission of sequenced messages between machines. No provision is made for flow control since the protocol is designed to transport RPC messages between client and server. A small outstanding message window prevents too many incoming messages from being buffered; messages outside the window are discarded and must be retransmitted. Connection setup uses a two way handshake to generate initial sequence numbers at each end of the connection; subsequent data messages increment the sequence numbers allowing the receiver to resequence out of order messages. In contrast to other protocols, IL does not do blind retransmission. If a message is lost and a timeout occurs, a query message is sent. The query message is a small control message containing the current sequence numbers as seen by the sender. The receiver responds to a query by retransmitting missing messages. This allows the protocol to behave well in congested networks, where blind retransmission would cause further congestion. Like TCP, IL has adaptive timeouts. A round-trip timer is used to calculate acknowledge and retransmission times in terms of the network speed. This allows the protocol to perform well on both the Internet and on local Ethernets.

In keeping with the minimalist design of the rest of the kernel, IL is small. The entire protocol is 847 lines of code, compared to 2200 lines for TCP. IL is our protocol of choice.

4. Network Addressing

A uniform interface to protocols and devices is not sufficient to support the transparency we require. Since each network uses a different addressing scheme, the ASCII strings written to a control file have no common format. As a result, every tool must know the specifics of the networks it is capable of addressing. Moreover, since each machine supplies a subset of the available networks, each user must be aware of the networks supported by every terminal and server machine. This is obviously unacceptable.

Several possible solutions were considered and rejected; one deserves more discussion. We could have used a user-level file server to represent the network name space as a Plan 9 file tree. This global naming scheme has been implemented in other distributed systems. The file hierarchy provides paths to directories representing network domains. Each directory contains files representing the names of the machines in that domain; an example might be the path `/net/name/usa/edu/mit/ai`. Each machine file contains information like the IP address of the machine. We rejected this representation for several reasons. First, it is hard to devise a hierarchy encompassing all representations of the various network addressing schemes in a uniform manner. Datakit and Ethernet address strings have nothing in common. Second, the address of a machine is often only a small part of the information required to connect to a service on the machine. For example, the IP protocols require symbolic service names to be mapped into numeric port numbers, some of which are privileged and hence special. Information of this sort is hard to represent in terms of file operations. Finally, the size and number of the networks being represented burdens users with an unacceptably large amount of information about the organization of the network and its connectivity. In this case the Plan 9 representation of a resource as a file is not appropriate.

If tools are to be network independent, a third-party server must resolve network names. A server on each machine, with local knowledge, can select the best network for any particular destination machine or service. Since the network devices present a common interface, the only operation which differs between networks is name resolution. A symbolic name must be translated to the path of the clone file of a protocol device and an ASCII address string to write to the `ctl` file. A connection server (CS) provides this service.

4.1. Network Database

On most systems several files such as `/etc/hosts`, `/etc/networks`, `/etc/services`, `/etc/hosts.equiv`, `/etc/bootptab`, and `/etc/named.d` hold network information. Much time and effort is spent administering these files and keeping them mutually consistent. Tools attempt to automatically derive one or more of the files from information in other files but maintenance continues to be difficult and error prone.

Since we were writing an entirely new system, we were free to try a simpler approach. One database on a shared server contains all the information needed for network administration. Two ASCII files comprise the main database: `/lib/ndb/local` contains locally administered information and `/lib/ndb/global` contains information imported from elsewhere. The files contain sets of attribute/value pairs of the form *attr*=*value*, where *attr* and *value* are alphanumeric strings. Systems are described by multi-line entries; a header line at the left margin begins each entry followed by zero or more indented attribute/value pairs specifying names, addresses, properties, etc. For example, the entry for our CPU server specifies a domain name, an IP address, an Ethernet address, a Datakit address, a boot file, and supported protocols.

```
sys=helix
        dom=helix.research.bell-labs.com
        bootf=/mips/9power
        ip=135.104.9.31 ether=0800690222f0
        dk=nj/astro/helix
        proto=il flavor=9cpu
```

If several systems share entries such as network mask and gateway, we specify that information with the network or subnetwork instead of the system. The following entries define a Class B IP network and a few subnets derived from it. The entry for the network specifies the IP mask, file system, and authentication server for all systems on the network. Each subnetwork specifies its default IP gateway.

```
ipnet=mh-astro-net ip=135.104.0.0 ipmask=255.255.255.0
        fs=bootes.research.bell-labs.com
        auth=1127auth
ipnet=unix-room ip=135.104.117.0
        ipgw=135.104.117.1
ipnet=third-floor ip=135.104.51.0
        ipgw=135.104.51.1
ipnet=fourth-floor ip=135.104.52.0
        ipgw=135.104.52.1
```

Database entries also define the mapping of service names to port numbers for TCP, UDP, and IL.

```
tcp=echo        port=7
tcp=discard     port=9
tcp=systat      port=11
tcp=daytime     port=13
```

All programs read the database directly so consistency problems are rare. However the database files can become large. Our global file, containing all information about both Datakit and Internet systems in AT&T, has 43,000 lines. To speed searches, we build hash table files for each attribute we expect to search often. The hash file entries point to entries in the master files. Every hash file contains the modification time of its master file so we can avoid using an out-of-date hash table. Searches for attributes that aren't hashed or whose hash table is out-of-date still work, they just take longer.

4.2. Connection Server

On each system a user level connection server process, CS, translates symbolic names to addresses. CS uses information about available networks, the network database, and other servers (such as DNS) to translate names. CS is a file server serving a single file, `/net/cs`. A client writes a symbolic name to `/net/cs` then reads one line for each matching destination reachable from this system. The lines are of the form *filename message*, where *filename* is the path of the clone file to open for a new connection and *message* is the string to write to it to make the connection. The following example illustrates this. `Ndb/csquery` is a program that prompts for strings to write to `/net/cs` and prints the replies.

```
% ndb/csquery
> net!helix!9fs
/net/il/clone 135.104.9.31!17008
/net/dk/clone nj/astro/helix!9fs
```

CS provides meta-name translation to perform complicated searches. The special network name `net` selects any network in common between source and destination supporting the specified service. A host name of the form $*attr* is the name of an attribute in the network database. The database search returns the value of the matching attribute/value pair most closely associated with the source host. Most closely associated is defined on a per network basis. For example, the symbolic name `tcp!$auth!rexauth` causes CS to search for the `auth` attribute in the database entry for the source system, then its subnetwork (if there is one) and then its network.

```
% ndb/csquery
> net!$auth!rexauth
/net/il/clone 135.104.9.34!17021
/net/dk/clone nj/astro/p9auth!rexauth
/net/il/clone 135.104.9.6!17021
/net/dk/clone nj/astro/musca!rexauth
```

Normally CS derives naming information from its database files. For domain names however, CS first consults another user level process, the domain name server (DNS). If no DNS is reachable, CS relies on its own tables.

Like CS, the domain name server is a user level process providing one file, `/net/dns`. A client writes a request of the form *domain-name type*, where *type* is a domain name service resource record type. DNS performs a recursive query through the Internet domain name system producing one line per resource record found. The client reads `/net/dns` to retrieve the records. Like other domain name servers, DNS caches information learned from the network. DNS is implemented as a multi-process shared memory application with separate processes listening for network and local requests.

5. Library routines

The section on protocol devices described the details of making and receiving connections across a network. The dance is straightforward but tedious. Library routines are provided to relieve the programmer of the details.

5.1. Connecting

The dial library call establishes a connection to a remote destination. It returns an open file descriptor for the data file in the connection directory.

```
int  dial(char *dest, char *local, char *dir, int *cfdp)
```

dest is the symbolic name/address of the destination.

local is the local address. Since most networks do not support this, it is usually zero.

dir is a pointer to a buffer to hold the path name of the protocol directory representing this connection. Dial fills this buffer if the pointer is non-zero.

cfdp is a pointer to a file descriptor for the ctl file of the connection. If the pointer is non-zero, dial opens the control file and tucks the file descriptor here.

Most programs call dial with a destination name and all other arguments zero. Dial uses CS to translate the symbolic name to all possible destination addresses and attempts to connect to each in turn until one works. Specifying the special name net in the network portion of the destination allows CS to pick a network/protocol in common with the destination for which the requested service is valid. For example, assume the system research.bell-labs.com has the Datakit address nj/astro/research and IP addresses 135.104.117.5 and 129.11.4.1. The call

```
fd = dial("net!research.bell-labs.com!login", 0, 0, 0, 0);
```

tries in succession to connect to nj/astro/research!login on the Datakit and both 135.104.117.5!513 and 129.11.4.1!513 across the Internet.

Dial accepts addresses instead of symbolic names. For example, the destinations tcp!135.104.117.5!513 and tcp!research.bell-labs.com!login are equivalent references to the same machine.

5.2. Listening

A program uses four routines to listen for incoming connections. It first announce()s its intention to receive connections, then listen()s for calls and finally accept()s or reject()s them. Announce returns an open file descriptor for the ctl file of a connection and fills dir with the path of the protocol directory for the announcement.

```
int  announce(char *addr, char *dir)
```

Addr is the symbolic name/address announced; if it does not contain a service, the announcement is for all services not explicitly announced. Thus, one can easily write the equivalent of the inetd program without having to announce each separate service. An announcement remains in force until the control file is closed.

Listen returns an open file descriptor for the ctl file and fills ldir with the path of the protocol directory for the received connection. It is passed dir from the announcement.

```
int  listen(char *dir, char *ldir)
```

Accept and reject are called with the control file descriptor and ldir returned by listen. Some networks such as Datakit accept a reason for a rejection; networks such as IP ignore the third argument.

```
int  accept(int ctl, char *ldir)
int  reject(int ctl, char *ldir, char *reason)
```

The following code implements a typical TCP listener. It announces itself, listens for connections, and forks a new process for each. The new process echoes data on the connection until the remote end closes it. The "*" in the symbolic name means the announcement is valid for any addresses bound to the machine the program is run on.

```
int
echo_server(void)
{
	int dfd, lcfd;
	char adir[40], ldir[40];
	int n;
	char buf[256];

	afd = announce("tcp!*!echo", adir);
	if(afd < 0)
		return -1;

	for(;;){
		/* listen for a call */
		lcfd = listen(adir, ldir);
		if(lcfd < 0)
			return -1;

		/* fork a process to echo */
		switch(fork()){
		case 0:
			/* accept the call and open the data file */
			dfd = accept(lcfd, ldir);
			if(dfd < 0)
				return -1;

			/* echo until EOF */
			while((n = read(dfd, buf, sizeof(buf))) > 0)
				write(dfd, buf, n);
			exits(0);
		case -1:
			perror("forking");
		default:
			close(lcfd);
			break;
		}

	}
}
```

6. User Level

Communication between Plan 9 machines is done almost exclusively in terms of 9P messages. Only the two services `cpu` and `exportfs` are used. The `cpu` service is analogous to `rlogin`. However, rather than emulating a terminal session across the network, `cpu` creates a process on the remote machine whose name space is an analogue of the window in which it was invoked. `Exportfs` is a user level file server which allows a piece of name space to be exported from machine to machine across a network. It is used by the `cpu` command to serve the files in the terminal's name space when they are accessed from the cpu server.

By convention, the protocol and device driver file systems are mounted in a directory called `/net`. Although the per-process name space allows users to configure an arbitrary view of the system, in practice their profiles build a conventional name space.

6.1. Exportfs

Exportfs is invoked by an incoming network call. The *listener* (the Plan 9 equivalent of inetd) runs the profile of the user requesting the service to construct a name space before starting exportfs. After an initial protocol establishes the root of the file tree being exported, the remote process mounts the connection, allowing exportfs to act as a relay file server. Operations in the imported file tree are executed on the remote server and the results returned. As a result the name space of the remote machine appears to be exported into a local file tree.

The import command calls exportfs on a remote machine, mounts the result in the local name space, and exits. No local process is required to serve mounts; 9P messages are generated by the kernel's mount driver and sent directly over the network.

Exportfs must be multithreaded since the system calls open, read and write may block. Plan 9 does not implement the select system call but does allow processes to share file descriptors, memory and other resources. Exportfs and the configurable name space provide a means of sharing resources between machines. It is a building block for constructing complex name spaces served from many machines.

The simplicity of the interfaces encourages naive users to exploit the potential of a richly connected environment. Using these tools it is easy to gateway between networks. For example a terminal with only a Datakit connection can import from the server helix:

```
import -a helix /net
telnet ai.mit.edu
```

The import command makes a Datakit connection to the machine helix where it starts an instance exportfs to serve /net. The import command mounts the remote /net directory after (the -a option to import) the existing contents of the local /net directory. The directory contains the union of the local and remote contents of /net. Local entries supersede remote ones of the same name so networks on the local machine are chosen in preference to those supplied remotely. However, unique entries in the remote directory are now visible in the local /net directory. All the networks connected to helix, not just Datakit, are now available in the terminal. The effect on the name space is shown by the following example:

```
philw-gnot% ls /net
/net/cs
/net/dk
philw-gnot% import -a musca /net
philw-gnot% ls /net
/net/cs
/net/cs
/net/dk
/net/dk
/net/dns
/net/ether
/net/il
/net/tcp
/net/udp
```

6.2. Ftpfs

We decided to make our interface to FTP a file system rather than the traditional command. Our command, *ftpfs,* dials the FTP port of a remote system, prompts for login and password, sets image mode, and mounts the remote file system onto /n/ftp. Files and directories are cached to reduce traffic. The cache is updated whenever a file is created. Ftpfs works with TOPS-20, VMS, and various Unix flavors as the

remote system.

7. Cyclone Fiber Links

The file servers and CPU servers are connected by high-bandwidth point-to-point links. A link consists of two VME cards connected by a pair of optical fibers. The VME cards use 33MHz Intel 960 processors and AMD's TAXI fiber transmitter/receivers to drive the lines at 125 Mbit/sec. Software in the VME card reduces latency by copying messages from system memory to fiber without intermediate buffering.

8. Performance

We measured both latency and throughput of reading and writing bytes between two processes for a number of different paths. Measurements were made on two- and four-CPU SGI Power Series processors. The CPUs are 25 MHz MIPS 3000s. The latency is measured as the round trip time for a byte sent from one process to another and back again. Throughput is measured using 16k writes from one process to another.

Table 1 - Performance

test	throughput MBytes/sec	latency millisec
pipes	8.15	.255
IL/ether	1.02	1.42
URP/Datakit	0.22	1.75
Cyclone	3.2	0.375

9. Conclusion

The representation of all resources as file systems coupled with an ASCII interface has proved more powerful than we had originally imagined. Resources can be used by any computer in our networks independent of byte ordering or CPU type. The connection server provides an elegant means of decoupling tools from the networks they use. Users successfully use Plan 9 without knowing the topology of the system or the networks they use. More information about 9P can be found in the Section 5 of the Plan 9 Programmer's Manual, Volume I.

10. References

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, ''Plan 9 from Bell Labs'', *UKUUG Proc. of the Summer 1990 Conf. , London, England, 1990.*

[Needham] R. Needham, ''Names'', in *Distributed systems,* S. Mullender, ed., Addison Wesley, 1989.

[Presotto] D. Presotto, ''Multiprocessor Streams for Plan 9'', *UKUUG Proc. of the Summer 1990 Conf. ,* London, England, 1990.

[Met80] R. Metcalfe, D. Boggs, C. Crane, E. Taf and J. Hupp, ''The Ethernet Local Network: Three reports'', *CSL-80-2,* XEROX Palo Alto Research Center, February 1980.

[Fra80] A. G. Fraser, ''Datakit - A Modular Network for Synchronous and Asynchronous Traffic'', *Proc. Int'l Conf. on Communication,* Boston, June 1980.

[Pet89a] L. Peterson, ''RPC in the X-Kernel: Evaluating new Design Techniques'', *Proc. Twelfth Symp. on Op. Sys. Princ.,* Litchfield Park, AZ, December 1990.

[Rit84a] D. M. Ritchie, ''A Stream Input-Output System'', *AT&T Bell Laboratories Technical Journal, 68(8),* October 1984.

Security in Plan 9

Russ Cox, MIT LCS
Eric Grosse, Bell Labs
Rob Pike, Bell Labs
Dave Presotto, Avaya Labs and Bell Labs
Sean Quinlan, Bell Labs
{rsc,ehg,rob,presotto,seanq}@plan9.bell-labs.com

ABSTRACT

The security architecture of the Plan 9™ operating system has recently been redesigned to address some technical shortcomings. This redesign provided an opportunity also to make the system more convenient to use securely. Plan 9 has thus improved in two ways not usually seen together: it has become more secure *and* easier to use.

The central component of the new architecture is a per-user self-contained agent called `factotum`. `Factotum` securely holds a copy of the user's keys and negotiates authentication protocols, on behalf of the user, with secure services around the network. Concentrating security code in a single program offers several advantages including: ease of update or repair to broken security software and protocols; the ability to run secure services at a lower privilege level; uniform management of keys for all services; and an opportunity to provide single sign on, even to unchanged legacy applications. `Factotum` has an unusual architecture: it is implemented as a Plan 9 file server.

1. Introduction

Secure computing systems face two challenges: first, they must employ sophisticated technology that is difficult to design and prove correct; and second, they must be easy for regular people to use. The question of ease of use is sometimes neglected, but it is essential: weak but easy-to-use security can be more effective than strong but difficult-to-use security if it is more likely to be used. People lock their front doors when they leave the house, knowing full well that a burglar is capable of picking the lock (or avoiding the door altogether); yet few would accept the cost and awkwardness of a bank vault door on the house even though that might reduce the probability of a robbery. A related point is that users need a clear model of how the security operates (if not how it actually provides security) in order to use it well; for example, the clarity of a lock icon on a web browser is offset by the confusing and typically insecure steps for installing X.509 certificates.

The security architecture of the Plan 9 operating system [Pike95] has recently been redesigned to make it both more secure and easier to use. By *security* we mean three things: first, the business of authenticating users and services; second, the safe handling, deployment, and use of keys and other secret information; and third, the use of

To appear, in a slightly different form, in *Proc. of the 2002 Usenix Security Symposium,* San Francisco.

encryption and integrity checks to safeguard communications from prying eyes.

The old security architecture of Plan 9 had several engineering problems in common with other operating systems. First, it had an inadequate notion of security domain. Once a user provided a password to connect to a local file store, the system required that the same password be used to access all the other file stores. That is, the system treated all network services as belonging to the same security domain.

Second, the algorithms and protocols used in authentication, by nature tricky and difficult to get right, were compiled into the various applications, kernel modules, and file servers. Changes and fixes to a security protocol required that all components using that protocol needed to be recompiled, or at least relinked, and restarted.

Third, the file transport protocol, 9P [Pike93], that forms the core of the Plan 9 system, had its authentication protocol embedded in its design. This meant that fixing or changing the authentication used by 9P required deep changes to the system. If someone were to find a way to break the protocol, the system would be wide open and very hard to fix.

These and a number of lesser problems, combined with a desire for more widespread use of encryption in the system, spurred us to rethink the entire security architecture of Plan 9.

The centerpiece of the new architecture is an agent, called `factotum`, that handles the user's keys and negotiates all security interactions with system services and applications. Like a trusted assistant with a copy of the owner's keys, `factotum` does all the negotiation for security and authentication. Programs no longer need to be compiled with cryptographic code; instead they communicate with `factotum` agents that represent distinct entities in the cryptographic exchange, such as a user and server of a secure service. If a security protocol needs to be added, deleted, or modified, only `factotum` needs to be updated for all system services to be kept secure.

Building on `factotum`, we modified secure services in the system to move user authentication code into `factotum`; made authentication a separable component of the file server protocol; deployed new security protocols; designed a secure file store, called `secstore`, to protect our keys but make them easy to get when they are needed; designed a new kernel module to support transparent use of Transport Layer Security (TLS) [RFC2246]; and began using encryption for all communications within the system. The overall architecture is illustrated in Figure 1a.

Secure protocols and algorithms are well understood and are usually not the weakest link in a system's security. In practice, most security problems arise from buggy servers, confusing software, or administrative oversights. It is these practical problems that we are addressing. Although this paper describes the algorithms and protocols we are using, they are included mainly for concreteness. Our main intent is to present a simple security architecture built upon a small trusted code base that is easy to verify (whether by manual or automatic means), easy to understand, and easy to use.

Although it is a subjective assessment, we believe we have achieved our goal of ease of use. That we have achieved our goal of improved security is supported by our plan to move our currently private computing environment onto the Internet outside the corporate firewall. The rest of this paper explains the architecture and how it is used, to explain why a system that is easy to use securely is also safe enough to run in the open network.

2. An Agent for Security

One of the primary reasons for the redesign of the Plan 9 security infrastructure was to remove the authentication method both from the applications and from the kernel. Cryptographic code is large and intricate, so it should be packaged as a separate component that can be repaired or modified without altering or even relinking applications

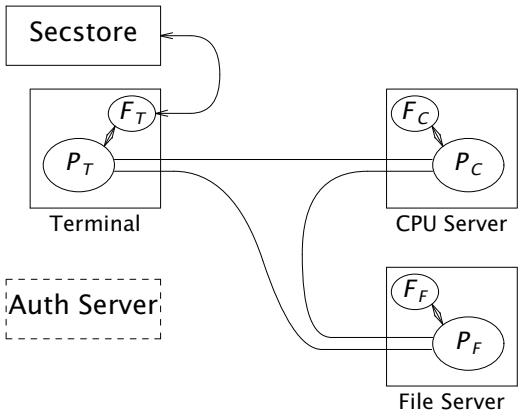


Figure 1a. Components of the security architecture. Each box is a (typically) separate machine; each ellipse a process. The ellipses labeled F_X are `factotum` processes; those labeled P_X are the pieces and proxies of a distributed program. The authentication server is one of several repositories for users' security information that `factotum` processes consult as required. `Secstore` is a shared resource for storing private information such as keys; `factotum` consults it for the user during bootstrap.

and services that depend on it. If a security protocol is broken, it should be trivial to repair, disable, or replace it on the fly. Similarly, it should be possible for multiple programs to use a common security protocol without embedding it in each program.

Some systems use dynamically linked libraries (DLLs) to address these configuration issues. The problem with this approach is that it leaves security code in the same address space as the program using it. The interactions between the program and the DLL can therefore accidentally or deliberately violate the interface, weakening security. Also, a program using a library to implement secure services must run at a privilege level necessary to provide the service; separating the security to a different program makes it possible to run the services at a weaker privilege level, isolating the privileged code to a single, more trustworthy component.

Following the lead of the SSH agent [Ylon96], we give each user an agent process responsible for holding and using the user's keys. The agent program is called `factotum` because of its similarity to the proverbial servant with the power to act on behalf of his master because he holds the keys to all the master's possessions. It is essential that `factotum` keep the keys secret and use them only in the owner's interest. Later we'll discuss some changes to the kernel to reduce the possibility of `factotum` leaking information inadvertently.

`Factotum` is implemented, like most Plan 9 services, as a file server. It is conventionally mounted upon the directory `/mnt/factotum`, and the files it serves there are analogous to virtual devices that provide access to, and control of, the services of the `factotum`. The next few sections describe the design of `factotum` and how it operates with the other pieces of Plan 9 to provide security services.

2.1. Logging in

To make the discussions that follow more concrete, we begin with a couple of examples showing how the Plan 9 security architecture appears to the user. These examples both involve a user `gre` logging in after booting a local machine. The user may or may not have a secure store in which all his keys are kept. If he does, `factotum` will prompt him for the password to the secure store and obtain keys from it, prompting only when a key isn't found in the store. Otherwise, `factotum` must prompt for each key.

In the typescripts, `\n` represents a literal newline character typed to force a default

response. User input is in italics, and long lines are folded and indented to fit.

This first example shows a user logging in without help from the secure store. First, `factotum` prompts for a user name that the local kernel will use:

```
user[none]: gre
```

(Default responses appear in square brackets.) The kernel then starts accessing local resources and requests, through `factotum`, a user/password pair to do so:

```
!Adding key: dom=cs.bell-labs.com
    proto=p9sk1
user[gre]: \n
password: ****
```

Now the user is logged in to the local system, and the mail client starts up:

```
!Adding key: proto=apop
    server=plan9.bell-labs.com
user[gre]: \n
password: ****
```

`Factotum` is doing all the prompting and the applications being started are not even touching the keys. Note that it's always clear which key is being requested.

Now consider the same login sequence, but in the case where `gre` has a secure store account:

```
user[none]: gre
secstore password: ********
STA PIN+SecurID: ********
```

That's the last `gre` will hear from `factotum` unless an attempt is made to contact a system for which no key is kept in the secure store.

2.2. The factotum

Each computer running Plan 9 has one user id that owns all the resources on that system — the scheduler, local disks, network interfaces, etc. That user, the *host owner*, is the closest analogue in Plan 9 to a Unix `root` account (although it is far weaker; rather than having special powers, as its name implies the host owner is just a regular user that happens to own the resources of the local machine). On a single-user system, which we call a terminal, the host owner is the id of the terminal's user. Shared servers such as CPU servers normally have a pseudo-user that initially owns all resources. At boot time, the Plan 9 kernel starts a `factotum` executing as, and therefore with the privileges of, the host owner.

New processes run as the same user as the process which created them. When a process must take on the identity of a new user, such as to provide a login shell on a shared CPU server, it does so by proving to the host owner's `factotum` that it is authorized to do so. This is done by running an authentication protocol with `factotum` to prove that the process has access to secret information which only the new user should possess. For example, consider the setup in Figure 1a. If a user on the terminal wants to log in to the CPU server using the Plan 9 `cpu` service [Pike93], then P_T might be the `cpu` client program and P_C the `cpu` server. Neither P_C nor P_T knows the details of the authentication. They do need to be able to shuttle messages back and forth between the two `factotums`, but this is a generic function easily performed without knowing, or being able to extract, secrets in the messages. P_T will make a network connection to P_C. P_T and P_C will then relay messages between the `factotum` owned by the user, F_T, and the one owned by the CPU server, F_C, until mutual authentication has been established. Later sections describe the RPC between `factotum` and applications and the library functions to support proxy operations.

The kernel always uses a single local instance of `factotum`, running as the host owner, for its authentication purposes, but a regular user may start other `factotum` agents. In fact, the `factotum` representing the user need not be running on the same machine as its client. For instance, it is easy for a user on a CPU server, through standard Plan 9 operations, to replace the `/mnt/factotum` in the user's private file name space on the server with a connection to the `factotum` running on the terminal. (The usual file system permissions prevent interlopers from doing so maliciously.) This permits secure operations on the CPU server to be transparently validated by the user's own `factotum`, so secrets need never leave the user's terminal. The SSH agent [Ylon96] does much the same with special SSH protocol messages, but an advantage to making our agent a file system is that we need no new mechanism to access our remote agent; remote file access is sufficient.

Within `factotum`, each protocol is implemented as a state machine with a generic interface, so protocols are in essence pluggable modules, easy to add, modify, or drop. Writing a message to and reading a message from `factotum` each require a separate RPC and result in a single state transition. Therefore `factotum` always runs to completion on every RPC and never blocks waiting for input during any authentication. Moreover, the number of simultaneous authentications is limited only by the amount of memory we're willing to dedicate to representing the state machines.

Authentication protocols are implemented only within `factotum`, but adding and removing protocols does require relinking the binary, so `factotum` processes (but no others) need to be restarted in order to take advantage of new or repaired protocols.

At the time of writing, `factotum` contains authentication modules for the Plan 9 shared key protocol (p9sk1), SSH's RSA authentication, passwords in the clear, APOP, CRAM, PPP's CHAP, Microsoft PPP's MSCHAP, and VNC's challenge/response.

2.3. Local capabilities

A capability system, managed by the kernel, is used to empower `factotum` to grant permission to another process to change its user id. A kernel device driver implements two files, `/dev/caphash` and `/dev/capuse`. The write-only file `/dev/caphash` can be opened only by the host owner, and only once. `Factotum` opens this file immediately after booting.

To use the files, `factotum` creates a string of the form *userid1*@*userid2*@*random-string*, uses SHA1 HMAC to hash *userid1*@*userid2* with key *random-string*, and writes that hash to `/dev/caphash`. `Factotum` then passes the original string to another process on the same machine, running as user *userid1*, which writes the string to `/dev/capuse`. The kernel hashes the string and looks for a matching hash in its list. If it finds one, the writing process's user id changes from *userid1* to *userid2*. Once used, or if a timeout expires, the capability is discarded by the kernel.

The capabilities are local to the machine on which they are created. Hence a `factotum` running on one machine cannot pass capabilities to processes on another and expect them to work.

2.4. Keys

We define the word *key* to mean not only a secret, but also a description of the context in which that secret is to be used: the protocol, server, user, etc. to which it applies. That is, a key is a combination of secret and descriptive information used to authenticate the identities of parties transmitting or receiving information. The set of keys used in any authentication depends both on the protocol and on parameters passed by the program requesting the authentication.

Taking a tip from SDSI [RiLa], which represents security information as textual S-expressions, keys in Plan 9 are represented as plain UTF-8 text. Text is easily

understood and manipulated by users. By contrast, a binary or other cryptic format can actually reduce overall security. Binary formats are difficult for users to examine and can only be cracked by special tools, themselves poorly understood by most users. For example, very few people know or understand what's inside their X.509 certificates. Most don't even know where in the system to find them. Therefore, they have no idea what they are trusting, and why, and are powerless to change their trust relationships. Textual, centrally stored and managed keys are easier to use and safer.

Plan 9 has historically represented databases as attribute/value pairs, since they are a good foundation for selection and projection operations. `Factotum` therefore represents the keys in the format *attribute*=*value*, where *attribute* is an identifier, possibly with a single-character prefix, and *value* is an arbitrary quoted string. The pairs themselves are separated by white space. For example, a Plan 9 key and an APOP key might be represented like this:

```
dom=bell-labs.com proto=p9sk1 user=gre
        !password='don''t tell'
proto=apop server=x.y.com user=gre
        !password='open sesame'
```

If a value is empty or contains white space or single quotes, it must be quoted; quotes are represented by doubled single quotes. Attributes that begin with an exclamation mark (`!`) are considered *secret*. `Factotum` will never let a secret value escape its address space and will suppress keyboard echo when asking the user to type one.

A program requesting authentication selects a key by providing a *query*, a list of elements to be matched by the key. Each element in the list is either an *attribute*=*value* pair, which is satisfied by keys with exactly that pair; or an attribute followed by a question mark, *attribute*?, which is satisfied by keys with some pair specifying the attribute. A key matches a query if every element in the list is satisfied. For instance, to select the APOP key in the previous example, an APOP client process might specify the query

```
server=x.y.com proto=apop
```

Internally, `factotum`'s APOP module would add the requirements of having `user` and `!password` attributes, forming the query

```
server=x.y.com proto=apop user? !password?
```

when searching for an appropriate key.

`Factotum` modules expect keys to have some well-known attributes. For instance, the `proto` attribute specifies the protocol module responsible for using a particular key, and protocol modules may expect other well-known attributes (many expect keys to have `!password` attributes, for example). Additional attributes can be used as comments or for further discrimination without intervention by `factotum`; for example, the APOP and IMAP mail clients conventionally include a `server` attribute to select an appropriate key for authentication.

Unlike in SDSI, keys in Plan 9 have no nested structure. This design keeps the representation simple and straightforward. If necessary, we could add a nested attribute or, in the manner of relational databases, an attribute that selects another tuple, but so far the simple design has been sufficient.

A simple common structure for all keys makes them easy for users to administer, but the set of attributes and their interpretation is still protocol-specific and can be subtle. Users may still need to consult a manual to understand all details. Many attributes (`proto`, `user`, `password`, `server`) are self-explanatory and our short experience has not uncovered any particular difficulty in handling keys. Things will likely get messier, however, when we grapple with public keys and their myriad components.

2.5. Protecting keys

Secrets must be prevented from escaping factotum. There are a number of ways they could leak: another process might be able to debug the agent process, the agent might swap out to disk, or the process might willingly disclose the key. The last is the easiest to avoid: secret information in a key is marked as such, and whenever factotum prints keys or queries for new ones, it is careful to avoid displaying secret information. (The only exception to this is the ''plaintext password'' protocol, which consists of sending the values of the user and !password attributes. Only keys tagged with proto=pass can have their passwords disclosed by this mechanism.)

Preventing the first two forms of leakage requires help from the kernel. In Plan 9, every process is represented by a directory in the /proc file system. Using the files in this directory, other processes could (with appropriate access permission) examine factotum's memory and registers. Factotum is protected from processes of other users by the default access bits of its /proc directory. However, we'd also like to protect the agent from other processes owned by the same user, both to avoid honest mistakes and to prevent an unattended terminal being exploited to discover secret passwords. To do this, we added a control message to /proc called private. Once the factotum process has written private to its /proc/*pid*/ctl file, no process can access factotum's memory through /proc. (Plan 9 has no other mechanism, such as /dev/kmem, for accessing a process's memory.)

Similarly, the agent's address space should not be swapped out, to prevent discovering unencrypted keys on the swapping media. The noswap control message in /proc prevents this scenario. Neither private nor noswap is specific to factotum. User-level file servers such as dossrv, which interprets FAT file systems, could use noswap to keep their buffer caches from being swapped to disk.

Despite our precautions, attackers might still find a way to gain access to a process running as the host owner on a machine. Although they could not directly access the keys, attackers could use the local factotum to perform authentications for them. In the case of some keys, for example those locking bank accounts, we want a way to disable or at least detect such access. That is the role of the confirm attribute in a key. Whenever a key with a confirm attribute is accessed, the local user must confirm use of the key via a local GUI. The next section describes the actual mechanism.

We have not addressed leaks possible as a result of someone rebooting or resetting a machine running factotum. For example, someone could reset a machine and reboot it with a debugger instead of a kernel, allowing them to examine the contents of memory and find keys. We have not found a satisfactory solution to this problem.

2.6. Factotum transactions

External programs manage factotum's internal key state through its file interface, writing textual key and delkey commands to the /mnt/factotum/ctl file. Both commands take a list of attributes as an argument. Key creates a key with the given attributes, replacing any extant key with an identical set of public attributes. Delkey deletes all keys that match the given set of attributes. Reading the ctl file returns a list of keys, one per line, displaying only public attributes. The following example illustrates these interactions.

```
% cd /mnt/factotum
% ls -l
-lrw------- gre gre 0 Jan 30 22:17 confirm
--rw------- gre gre 0 Jan 30 22:17 ctl
-lr-------- gre gre 0 Jan 30 22:17 log
-lrw------- gre gre 0 Jan 30 22:17 needkey
--r--r--r-- gre gre 0 Jan 30 22:17 proto
--rw-rw-rw- gre gre 0 Jan 30 22:17 rpc
% cat >ctl
key dom=bell-labs.com proto=p9sk1 user=gre
    !password='don''t tell'
key proto=apop server=x.y.com user=gre
    !password='bite me'
^D
% cat ctl
key dom=bell-labs.com proto=p9sk1 user=gre
key proto=apop server=x.y.com user=gre
% echo 'delkey proto=apop' >ctl
% cat ctl
key dom=bell-labs.com proto=p9sk1 user=gre
%
```

(A file with the `l` bit set can be opened by only one process at a time.)

The heart of the interface is the `rpc` file. Programs authenticate with `factotum` by writing a request to the `rpc` file and reading back the reply; this sequence is called an RPC *transaction*. Requests and replies have the same format: a textual verb possibly followed by arguments, which may be textual or binary. The most common reply verb is `ok`, indicating success. An RPC session begins with a `start` transaction; the argument is a key query as described earlier. Once started, an RPC conversation usually consists of a sequence of `read` and `write` transactions. If the conversation is successful, an `authinfo` transaction will return information about the identities learned during the transaction. The `attr` transaction returns a list of attributes for the current conversation; the list includes any attributes given in the `start` query as well as any public attributes from keys being used.

As an example of the `rpc` file in action, consider a mail client connecting to a mail server and authenticating using the POP3 protocol's APOP challenge-response command. There are four programs involved: the mail client P_C, the client `factotum` F_C, the mail server P_S, and the server `factotum` F_S. All authentication computations are handled by the `factotum` processes. The mail programs' role is just to relay messages.

At startup, the mail server at `x.y.com` begins an APOP conversation with its `factotum` to obtain the banner greeting, which includes a challenge:

$P_S \rightarrow F_S$: `start proto=apop role=server`
$F_S \rightarrow P_S$: `ok`
$P_S \rightarrow F_S$: `read`
$F_S \rightarrow P_S$: `ok +OK POP3` *challenge*

Having obtained the challenge, the server greets the client:

$P_S \rightarrow P_C$: `+OK POP3` *challenge*

The client then uses an APOP conversation with its `factotum` to obtain a response:

$P_C \rightarrow F_C$: `start proto=apop role=client server=x.y.com`
$F_C \rightarrow P_C$: `ok`
$P_C \rightarrow F_C$: `write +OK POP3` *challenge*
$F_C \rightarrow P_C$: `ok`
$P_C \rightarrow F_C$: `read`
$F_C \rightarrow P_C$: `ok APOP gre` *response*

`Factotum` requires that `start` requests include a `proto` attribute, and the APOP module requires an additional `role` attribute, but the other attributes are optional and only restrict the key space. Before responding to the `start` transaction, the client `factotum` looks for a key to use for the rest of the conversation. Because of the arguments in the `start` request, the key must have public attributes `proto=apop` and `server=x.y.com`; as mentioned earlier, the APOP module additionally requires that the key have `user` and `!password` attributes. Now that the client has obtained a response from its `factotum`, it echoes that response to the server:

$P_C \rightarrow P_S$: `APOP gre` *response*

Similarly, the server passes this message to its `factotum` and obtains another to send back.

$P_S \rightarrow F_S$: `write APOP gre` *response*
$F_S \rightarrow P_S$: `ok`
$P_S \rightarrow F_S$: `read`
$F_S \rightarrow P_S$: `ok +OK welcome`

$P_S \rightarrow P_C$: `+OK welcome`

Now the authentication protocol is done, and the server can retrieve information about what the protocol established.

$P_S \rightarrow F_S$: `authinfo`
$F_S \rightarrow P_S$: `ok client=gre capability=`*capability*

The `authinfo` data is a list of *attr=value* pairs, here a client user name and a capability. (Protocols that establish shared secrets or provide mutual authentication indicate this by adding appropriate *attr=value* pairs.) The capability can be used by the server to change its identity to that of the client, as described earlier. Once it has changed its identity, the server can access and serve the client's mailbox.

Two more files provide hooks for a graphical `factotum` control interface. The first, `confirm`, allows the user detailed control over the use of certain keys. If a key has a `confirm=` attribute, then the user must approve each use of the key. A separate program with a graphical interface reads from the `confirm` file to see when a confirmation is necessary. The read blocks until a key usage needs to be approved, whereupon it will return a line of the form

`confirm tag=1` *attributes*

requesting permission to use the key with those public attributes. The graphical interface then prompts the user for approval and writes back

`tag=1 answer=yes`

(or `answer=no`).

The second file, `needkey`, diverts key requests. In the APOP example, if a suitable key had not been found during the `start` transaction, `factotum` would have indicated failure by returning a response indicating what key was needed:

$F_C \rightarrow P_C$: `needkey proto=apop server=x.y.com user? !password?`

A typical client would then prompt the user for the desired key information, create a new key via the `ctl` file, and then reissue the `start` request. If the `needkey` file is open, then instead of failing, the transaction will block, and the next read from the `/mnt/factotum/needkey` file will return a line of the form

`needkey tag=1` *attributes*

The graphical interface then prompts the user for the needed key information, creates the key via the `ctl` file, and writes back `tag=1` to resume the transaction.

The remaining files are informational and used for debugging. The `proto` file contains a list of supported protocols (to see what protocols the system supports, `cat /mnt/factotum/proto`), and the `log` file contains a log of operations and debugging output enabled by a `debug` control message.

The next few sections explain how `factotum` is used by system services.

3. Authentication in 9P

Plan 9 uses a remote file access protocol, 9P [Pike93], to connect to resources such as the file server and remote processes. The original design for 9P included special messages at the start of a conversation to authenticate the user. Multiple users can share a single connection, such as when a CPU server runs processes for many users connected to a single file server, but each must authenticate separately. The authentication protocol, similar to that of Kerberos [Stei88], used a sequence of messages passed between client, file server, and authentication server to verify the identities of the user, calling machine, and serving machine. One major drawback to the design was that the authentication method was defined by 9P itself and could not be changed. Moreover, there was no mechanism to relegate authentication to an external (trusted) agent, so a process implementing 9P needed, besides support for file service, a substantial body of cryptographic code to implement a handful of startup messages in the protocol.

A recent redesign of 9P addressed a number of file service issues outside the scope of this paper. On issues of authentication, there were two goals: first, to remove details about authentication from the protocol itself; second, to allow an external program to execute the authentication part of the protocol. In particular, we wanted a way to quickly incorporate ideas found in other systems such as SFS [Mazi99].

Since 9P is a file service protocol, the solution involved creating a new type of file to be served: an *authentication file*. Connections to a 9P service begin in a state that allows no general file access but permits the client to open an authentication file by sending a special message, generated by the new `fauth` system call:

```
afd = fauth(int fd, char *servicename);
```

Here `fd` is the user's file descriptor for the established network connection to the 9P server and `servicename` is the name of the desired service offered on that server, typically the file subsystem to be accessed. The returned file descriptor, `afd`, is a unique handle representing the authentication file created for this connection to authenticate to this service; it is analogous to a capability. The authentication file represented by `afd` is not otherwise addressable on the server, such as through the file name hierarchy. In all other respects, it behaves like a regular file; most important, it accepts standard read and write operations.

To prove its identity, the user process (via `factotum`) executes the authentication protocol, described in the next section of this paper, over the `afd` file descriptor with ordinary reads and writes. When client and server have successfully negotiated, the authentication file changes state so it can be used as evidence of authority in `mount`.

Once identity is established, the process presents the (now verified) `afd` as proof of identity to the `mount` system call:

```
mount(int fd, int afd, char *mountpoint,
      int flag, char *servicename)
```

If the `mount` succeeds, the user now has appropriate permissions for the file hierarchy made visible at the mount point.

This sequence of events has several advantages. First, the actual authentication protocol is implemented using regular reads and writes, not special 9P messages, so they can be processed, forwarded, proxied, and so on by any 9P agent without special arrangement. Second, the business of negotiating the authentication by reading and writing the authentication file can be delegated to an outside agent, in particular `factotum`; the programs that implement the client and server ends of a 9P conversation need no authentication or cryptographic code. Third, since the authentication protocol is not defined by 9P itself, it is easy to change and can even be negotiated dynamically. Finally, since `afd` acts like a capability, it can be treated like one: handed to another process to give it special permissions; kept around for later use when authentication is again required; or closed to make sure no other process can use it.

All these advantages stem from moving the authentication negotiation into reads and writes on a separate file. As is often the case in Plan 9, making a resource (here authentication) accessible with a file-like interface reduces *a priori* the need for special interfaces.

3.1. Plan 9 shared key protocol

In addition to the various standard protocols supported by `factotum`, we use a shared key protocol for native Plan 9 authentication. This protocol provides backward compatibility with older versions of the system. One reason for the new architecture is to let us replace such protocols in the near future with more cryptographically secure ones.

P9sk1 is a shared key protocol that uses tickets much like those in the original Kerberos. The difference is that we've replaced the expiration time in Kerberos tickets with a random nonce parameter and a counter. We summarize it here:

$C \rightarrow S$: $nonce_C$
$S \rightarrow C$: $nonce_S, uid_S, domain_S$

$C \rightarrow A$: $nonce_S, uid_S, domain_S, uid_C,$ $factotum_C$
$A \rightarrow C$: $K_C\{nonce_S, uid_C, uid_S, K_n\},$ $K_S\{nonce_S, uid_C, uid_S, K_n\}$

$C \rightarrow S$: $K_S\{nonce_S, uid_C, uid_S, K_n\},$ $K_n\{nonce_S, counter\}$
$S \rightarrow C$: $K_n\{nonce_C, counter\}$

(Here $K\{x\}$ indicates x encrypted with DES key K.) The first two messages exchange nonces and server identification. After this initial exchange, the client contacts the authentication server to obtain a pair of encrypted tickets, one encrypted with the client key and one with the server key. The client relays the server ticket to the server. The server believes that the ticket is new because it contains $nonce_S$ and that the ticket is from the authentication server because it is encrypted in the server key K_S. The ticket is basically a statement from the authentication server that now uid_C and uid_S share a secret K_n. The authenticator $K_n\{nonce_S, counter\}$ convinces the server that the client knows K_n and thus must be uid_C. Similarly, authenticator $K_n\{nonce_C, counter\}$ convinces the client that the server knows K_n and thus must be uid_S. Tickets can be reused, without contacting the authentication server again, by incrementing the counter

before each authenticator is generated.

In the future we hope to introduce a public key version of p9sk1, which would allow authentication even when the authentication server is not available.

3.2. The authentication server

Each Plan 9 security domain has an authentication server (AS) that all users trust to keep the complete set of shared keys. It also offers services for users and administrators to manage the keys, create and disable accounts, and so on. It typically runs on a standalone machine with few other services. The AS comprises two services, `keyfs` and `authsrv`.

`Keyfs` is a user-level file system that manages an encrypted database of user accounts. Each account is represented by a directory containing the files `key`, containing the Plan 9 key for p9sk1; `secret` for the challenge/response protocols (APOP, VNC, CHAP, MSCHAP, CRAM); `log` for authentication outcomes; `expire` for an expiration time; and `status`. If the expiration time passes, if the number of successive failed authentications exceeds 50, or if `disabled` is written to the status file, any attempt to access the `key` or `secret` files will fail.

`Authsrv` is a network service that brokers shared key authentications for the protocols p9sk1, APOP, VNC, CHAP, MSCHAP, and CRAM. Remote users can also call `authsrv` to change their passwords.

The p9sk1 protocol was described in the previous section. The challenge/response protocols differ in detail but all follow the general structure:

$C \rightarrow S$: $nonce_C$
$S \rightarrow C$: $nonce_S, uid_S, domain_S$
$C \rightarrow A$: $nonce_S, uid_S, domain_S,$ $hostid_C, uid_C$
$A \rightarrow C$: $K_C\{nonce_S, uid_C, uid_S, K_n\},$ $K_S\{nonce_S, uid_C, uid_S, K_n\}$
$C \rightarrow S$: $K_S\{nonce_S, uid_C, uid_S, K_n\},$ $K_n\{nonce_S\}$
$S \rightarrow C$: $K_n\{nonce_C\}$

The password protocol is:

$C \rightarrow A$: uid_C
$A \rightarrow C$: $K_c\{K_n\}$
$C \rightarrow A$: $K_n\{password_{old}, password_{new}\}$
$A \rightarrow C$: OK

To avoid replay attacks, the pre-encryption clear text for each of the protocols (as well as for p9sk1) includes a tag indicating the encryption's role in the protocol. We elided them in these outlines.

3.3. Protocol negotiation

Rather than require particular protocols for particular services, we implemented a negotiation metaprotocol, *p9any*, which chooses the actual authentication protocol to use. P9any is used now by all native services on Plan 9.

The metaprotocol is simple. The callee sends a null-terminated string of the form:

$$\texttt{v.}n\ \ proto_1@domain_1\ \ proto_2@domain_2\ \ \ldots$$

where n is a decimal version number, $proto_k$ is the name of a protocol for which the `factotum` has a key, and $domain_k$ is the name of the domain in which the key is valid. The caller then responds

proto@domain

indicating its choice. Finally the callee responds

OK

Any other string indicates failure. At this point the chosen protocol commences. The final fixed-length reply is used to make it easy to delimit the I/O stream should the chosen protocol require the caller rather than the callee to send the first message.

With this negotiation metaprotocol, the underlying authentication protocols used for Plan 9 services can be changed under any application just by changing the keys known by the factotum agents at each end.

P9any is vulnerable to man in the middle attacks to the extent that the attacker may constrain the possible choices by changing the stream. However, we believe this is acceptable since the attacker cannot force either side to choose algorithms that it is unwilling to use.

4. Library Interface to Factotum

Although programs can access factotum's services through its file system interface, it is more common to use a C library that packages the interaction. There are a number of routines in the library, not all of which are relevant here, but a few examples should give their flavor.

First, consider the problem of mounting a remote file server using 9P. An earlier discussion showed how the fauth and mount system calls use an authentication file, afd, as a capability, but not how factotum manages afd. The library contains a routine, amount (authenticated mount), that is used by most programs in preference to the raw fauth and mount calls. Amount engages factotum to validate afd; here is the complete code:

```
int
amount(int fd, char *mntpt,
  int flags, char *aname)
{
  int afd, ret;
  AuthInfo *ai;

  afd = fauth(fd, aname);
  if(afd >= 0){
    ai = auth_proxy(afd, amount_getkey,
      "proto=p9any role=client");
    if(ai != NULL)
      auth_freeAI(ai);
  }
  ret = mount(fd, afd, mntpt,
    flags, aname);
  if(afd >= 0)
    close(afd);
  return ret;
}
```

where parameter fd is a file descriptor returned by open or dial for a new connection to a file server. The conversation with factotum occurs in the call to auth_proxy, which specifies, as a key query, which authentication protocol to use (here the metaprotocol p9any) and the role being played (client). Auth_proxy will read and write the factotum files, and the authentication file descriptor afd, to validate the user's right to access the service. If the call is successful, any auxiliary data, held in an AuthInfo structure, is freed. In any case, the mount is then called with the (perhaps validated) afd. A 9P server can cause the fauth system call to fail,

as an indication that authentication is not required to access the service.

The second argument to auth_proxy is a function, here amount_getkey, to be called if secret information such as a password or response to a challenge is required as part of the authentication. This function, of course, will provide this data to factotum as a key message on the /mnt/factotum/ctl file.

Although the final argument to auth_proxy in this example is a simple string, in general it can be a formatted-print specifier in the manner of printf, to enable the construction of more elaborate key queries.

As another example, consider the Plan 9 cpu service, which exports local devices to a shell process on a remote machine, typically to connect the local screen and keyboard to a more powerful computer. At heart, cpu is a superset of a service called exportfs [Pike93], which allows one machine to see an arbitrary portion of the file name space of another machine, such as to export the network device to another machine for gatewaying. However, cpu is not just exportfs because it also delivers signals such as interrupt and negotiates the initial environment for the remote shell.

To authenticate an instance of cpu requires factotum processes on both ends: the local, client end running as the user on a terminal and the remote, server end running as the host owner of the server machine. Here is schematic code for the two ends:

```
/* client */
int
p9auth(int fd)
{
  AuthInfo *ai;

  ai = auth_proxy(fd, auth_getkey,
    "proto=p9any role=client");
  if(ai == NULL)
    return -1;

  /* start cpu protocol here */
}

/* server */
int
srvp9auth(int fd, char *user)
{
  AuthInfo *ai;

  ai = auth_proxy(fd, NULL,
    "proto=p9any role=server");
  if(ai == NULL)
    return -1;
  /* set user id for server process */
  if(auth_chuid(ai, NULL) < 0)
    return -1;

  /* start cpu protocol here */
}
```

Auth_chuid encapsulates the negotiation to change a user id using the caphash and capuse files of the (server) kernel. Note that although the client process may ask the user for new keys, using auth_getkey, the server machine, presumably a shared machine with a pseudo-user for the host owner, sets the key-getting function to NULL.

5. Secure Store

`Factotum` keeps its keys in volatile memory, which must somehow be initialized at boot time. Therefore, `factotum` must be supplemented by a persistent store, perhaps a floppy disk containing a key file of commands to be copied into `/mnt/factotum/ctl` during bootstrap. But removable media are a nuisance to carry and are vulnerable to theft. Keys could be stored encrypted on a shared file system, but only if those keys are not necessary for authenticating to the file system in the first place. Even if the keys are encrypted under a user password, a thief might well succeed with a dictionary attack. Other risks of local storage are loss of the contents through mechanical mishap or dead batteries. Thus for convenience and safety we provide a `secstore` (secure store) server in the network to hold each user's permanent list of keys, a *key file*.

`Secstore` is a file server for encrypted data, used only during bootstrapping. It must provide strong authentication and resistance to passive and active protocol attacks while assuming nothing more from the client than a password. Once `factotum` has loaded the key file, further encrypted or authenticated file storage can be accomplished by standard mechanisms.

The cryptographic technology that enables `secstore` is a form of encrypted key exchange called PAK [Boyk00], analogous to EKE [Bell93], SRP [Wu98], or SPEKE [Jabl]. PAK was chosen because it comes with a proof of equivalence in strength to Diffie-Hellman; subtle flaws in some earlier encrypted key exchange protocols and implementations have encouraged us to take special care. In outline, the PAK protocol is:

$C \rightarrow S$: $C, g^x H$
$S \rightarrow C$: $S, g^y, hash(g^{xy}, C, S)$
$C \rightarrow S$: $hash(g^{xy}, S, C)$

where H is a preshared secret between client C and server S. There are several variants of PAK, all presented in papers mainly concerned with proofs of cryptographic properties. To aid implementers, we have distilled a description of the specific version we use into an Appendix to this paper. The Plan 9 open source license provides for use of Lucent's encrypted key exchange patents in this context.

As a further layer of defense against password theft, we provide (within the encrypted channel $C \rightarrow S$) information that is validated at a RADIUS server, such as the digits from a hardware token [RFC2138]. This provides two-factor authentication, which potentially requires tricking two independent administrators in any attack by social engineering.

The key file stored on the server is encrypted with AES (Rijndael) using CBC with a 10-byte initialization vector and trailing authentication padding. All this is invisible to the user of `secstore`. For that matter, it is invisible to the `secstore` server as well; if the AES Modes of Operation are standardized and a new encryption format designed, it can be implemented by a client without change to the server. The `secstore` is deliberately not backed up; the user is expected to use more than one `secstore` or save the key file on removable media and lock it away. The user's password is hashed to create the H used in the PAK protocol; a different hash of the password is used as the file encryption key. Finally, there is a command (inside the authenticated, encrypted channel between client and `secstore`) to change passwords by sending a new H; for consistency, the client process must at the same time fetch and re-encrypt all files.

When `factotum` starts, it dials the local `secstore` and checks whether the user has an account. If so, it prompts for the user's `secstore` password and fetches the key file. The PAK protocol ensures mutual authentication and prevents dictionary attacks on the password by passive wiretappers or active intermediaries. Passwords saved in the key file can be long random strings suitable for simpler challenge/response authentication protocols. Thus the user need only remember a single, weaker password to enable strong, ''single sign on'' authentication to unchanged legacy applications scattered

across multiple authentication domains.

6. Transport Layer Security

Since the Plan 9 operating system is designed for use in network elements that must withstand direct attack, unguarded by firewall or VPN, we seek to ensure that all applications use channels with appropriate mutual authentication and encryption. A principal tool for this is TLS 1.0 [RFC2246]. (TLS 1.0 is nearly the same as SSL 3.0, and our software is designed to interoperate with implementations of either standard.)

TLS defines a record layer protocol for message integrity and privacy through the use of message digesting and encryption with shared secrets. We implement this service as a kernel device, though it could be performed at slightly higher cost by invoking a separate program. The library interface to the TLS kernel device is:

```
int pushtls(int fd, char *hashalg,
    char *cryptalg, int isclient,
    char *secret, char *dir);
```

Given a file descriptor, the names of message digest and encryption algorithms, and the shared secret, `pushtls` returns a new file descriptor for the encrypted connection. (The final argument `dir` receives the name of the directory in the TLS device that is associated with the new connection.) The function is named by analogy with the ''push'' operation supported by the stream I/O system of Research Unix and the first two editions of Plan 9. Because adding encryption is as simple as replacing one file descriptor with another, adding encryption to a particular network service is usually trivial.

The Plan 9 shared key authentication protocols establish a shared 56-bit secret as a side effect. Native Plan 9 network services such as `cpu` and `exportfs` use these protocols for authentication and then invoke `pushtls` with the shared secret.

Above the record layer, TLS specifies a handshake protocol using public keys to establish the session secret. This protocol is widely used with HTTP and IMAP4 to provide server authentication, though with client certificates it could provide mutual authentication. The library function

```
int tlsClient(int fd, TLSconn *conn)
```

handles the initial handshake and returns the result of `pushtls`. On return, it fills the `conn` structure with the session ID used and the X.509 certificate presented by the server, but makes no effort to verify the certificate. Although the original design intent of X.509 certificates expected that they would be used with a Public Key Infrastructure, reliable deployment has been so long delayed and problematic that we have adopted the simpler policy of just using the X.509 certificate as a representation of the public key, depending on a locally-administered directory of SHA1 thumbprints to allow applications to decide which public keys to trust for which purposes.

7. Related Work and Discussion

Kerberos, one of the earliest distributed authentication systems, keeps a set of authentication tickets in a temporary file called a ticket cache. The ticket cache is protected by Unix file permissions. An environment variable containing the file name of the ticket cache allows for different ticket caches in different simultaneous login sessions. A user logs in by typing his or her Kerberos password. The login program uses the Kerberos password to obtain a temporary ticket-granting ticket from the authentication server, initializes the ticket cache with the ticket-granting ticket, and then forgets the password. Other applications can use the ticket-granting ticket to sign tickets for themselves on behalf of the user during the login session. The ticket cache is removed when the user logs out [Stei88]. The ticket cache relieves the user from typing a password every time authentication is needed.

The secure shell SSH develops this idea further, replacing the temporary file with a named Unix domain socket connected to a user-level program, called an agent. Once the SSH agent is started and initialized with one or more RSA private keys, SSH clients can employ it to perform RSA authentications on their behalf. In the absence of an agent, SSH typically uses RSA keys read from encrypted disk files or uses passphrase-based authentication, both of which would require prompting the user for a passphrase whenever authentication is needed [Ylon96]. The self-certifying file system SFS uses a similar agent [Kami00], not only for moderating the use of client authentication keys but also for verifying server public keys [Mazi99].

`Factotum` is a logical continuation of this evolution, replacing the program-specific SSH or SFS agents with a general agent capable of serving a wide variety of programs. Having one agent for all programs removes the need to have one agent for each program. It also allows the programs themselves to be protocol-agnostic, so that, for example, one could build an SSH workalike capable of using any protocol supported by `factotum`, without that program knowing anything about the protocols. Traditionally each program needs to implement each authentication protocol for itself, an $O(n^2)$ coding problem that `factotum` reduces to $O(n)$.

Previous work on agents has concentrated on their use by clients authenticating to servers. Looking in the other direction, Sun Microsystem's pluggable authentication module (PAM) is one of the earliest attempts to provide a general authentication mechanism for Unix-like operating systems [Sama96]. Without a central authority like PAM, system policy is tied up in the various implementations of network services. For example, on a typical Unix, if a system administrator decides not to allow plaintext passwords for authentication, the configuration files for a half dozen different servers — `rlogind`, `telnetd`, `ftpd`, `sshd`, and so on — need to be edited. PAM solves this problem by hiding the details of a given authentication mechanism behind a common library interface. Directed by a system-wide configuration file, an application selects a particular authentication mechanism by dynamically loading the appropriate shared library. PAM is widely used on Sun's Solaris and some Linux distributions.

`Factotum` achieves the same goals using the agent approach. `Factotum` is the only process that needs to create capabilities, so all the network servers can run as untrusted users (e.g., Plan 9's `none` or Unix's `nobody`), which greatly reduces the harm done if a server is buggy and is compromised. In fact, if `factotum` were implemented on Unix along with an analogue to the Plan 9 capability device, venerable programs like `su` and `login` would no longer need to be installed "setuid root."

Several other systems, such as Password Safe [Schn], store multiple passwords in an encrypted file, so that the user only needs to remember one password. Our `secstore` solution differs from these by placing the storage in a hardened location in the network, so that the encrypted file is less liable to be stolen for offline dictionary attack and so that it is available even when a user has several computers. In contrast, Microsoft's Passport system [Micr] keeps credentials in the network, but centralized at one extremely-high-value target. The important feature of Passport, setting up trust relationships with e-merchants, is outside our scope. The `secstore` architecture is almost identical to Perlman and Kaufman's [Perl99] but with newer EKE technology. Like them, we chose to defend mainly against outside attacks on `secstore`; if additional defense of the files on the server itself is desired, one can use distributed techniques [Ford00].

We made a conscious choice of placing encryption, message integrity, and key management at the application layer (TLS, just above layer 4) rather than at layer 3, as in IPsec. This leads to a simpler structure for the network stack, easier integration with applications and, most important, easier network administration since we can recognize which applications are misbehaving based on TCP port numbers. TLS does suffer (relative to IPsec) from the possibility of forged TCP Reset, but we feel that this is adequately dealt

with by randomized TCP sequence numbers. In contrast with other TLS libraries, Plan 9 does not require the application to change `write` calls to `sslwrite` but simply to add a few lines of code at startup [Resc01].

8. Conclusion

Writing safe code is difficult. Stack attacks, mistakes in logic, and bugs in compilers and operating systems can each make it possible for an attacker to subvert the intended execution sequence of a service. If the server process has the privileges of a powerful user, such as `root` on Unix, then so does the attacker. `Factotum` allows us to constrain the privileged execution to a single process whose core is a few thousand lines of code. Verifying such a process, both through manual and automatic means, is much easier and less error prone than requiring it of all servers.

An implementation of these ideas is in Plan 9 from Bell Labs, Fourth Edition, freely available from `http://plan9.bell-labs.com/plan9`.

Acknowledgments

William Josephson contributed to the implementation of password changing in `secstore`. We thank Phil MacKenzie and Martín Abadi for helpful comments on early parts of the design. Chuck Blake, Peter Bosch, Frans Kaashoek, Sape Mullender, and Lakshman Y. N., predominantly Dutchmen, gave helpful comments on the paper. Russ Cox is supported by a fellowship from the Fannie and John Hertz Foundation.

References

[Bell93] S.M. Bellovin and M. Merritt, ''Augmented Encrypted Key Exchange,'' Proceedings of the 1st ACM Conference on Computer and Communications Security, 1993, pp. 244 - 250.

[Boyk00] Victor Boyko, Philip MacKenzie, and Sarvar Patel, ''Provably Secure Password-Authenticated Key Exchange using Diffie-Hellman,'' Eurocrypt 2000, 156-171.

[RFC2246] T . Dierks and C. Allen, ''The TLS Protocol, Version 1.0,'' RFC 2246.

[Ford00] Warwick Ford and Burton S. Kaliski, Jr., ''Server-Assisted Generation of a Strong Secret from a Password,'' IEEE Fifth International Workshop on Enterprise Security, National Institute of Standards and Technology (NIST), Gaithersburg MD, June 14 - 16, 2000.

[Jabl] David P. Jablon, ''Strong Password-Only Authenticated Key Exchange,'' `http://integritysciences.com/speke97.html`.

[Kami00] Michael Kaminsky. ''Flexible Key Management with SFS Agents,'' Master's Thesis, MIT, May 2000.

[Mack] Philip MacKenzie, private communication.

[Mazi99] David Mazières, Michael Kaminsky, M. Frans Kaashoek and Emmett Witchel, ''Separating key management from file system security,'' Symposium on Operating Systems Principles, 1999, pp. 124-139.

[Micr] Microsoft Passport, `http://www.passport.com/`.

[Perl99] Radia Perlman and Charlie Kaufman, ''Secure Password-Based Protocol for Downloading a Private Key,'' Proc. 1999 Network and Distributed System Security Symposium, Internet Society, January 1999.

[Pike95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom, ''Plan 9 from Bell Labs,'' Computing Systems, **8**, 3, Summer 1995, pp. 221-254.

[Pike93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Phil Winterbottom, ''The Use of Name Spaces in Plan 9,'' Operating Systems Review, **27**, 2, April 1993, pp.

72–76 (reprinted from Proceedings of the 5th ACM SIGOPS European Workshop, Mont Saint-Michel, 1992, Paper n° 34).

[Resc01] Eric Rescorla, ''SSL and TLS: Designing and Building Secure Systems,'' Addison-Wesley, 2001. ISBN 0-201-61598-3, p. 387.

[RFC2138] C. Rigney, A. Rubens, W. Simpson, S. Willens, ''Remote Authentication Dial In User Service (RADIUS),'' RFC2138, April 1997.

[RiLa] Ronald L. Rivest and Butler Lampson, ''SDSI—A Simple Distributed Security Infrastructure,'' `http://theory.lcs.mit.edu/~rivest/sdsi10.ps`.

[Schn] Bruce Schneier, Password Safe, `http://www.counterpane.com/-passsafe.html`.

[Sama96] Vipin Samar, ''Unified Login with Pluggable Authentication Modules (PAM),'' Proceedings of the Third ACM Conference on Computer Communications and Security, March 1996, New Delhi, India.

[Stei88] Jennifer G. Steiner, Clifford Neumann, and Jeffrey I. Schiller, ''*Kerberos*: An Authentication Service for Open Network Systems,'' Proceedings of USENIX Winter Conference, Dallas, Texas, February 1988, pp. 191–202.

[Wu98] T. Wu, ''The Secure Remote Password Protocol,'' Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium, San Diego, CA, March 1998, pp. 97–111.

[Ylon96] Ylonen, T., ''SSH—Secure Login Connections Over the Internet,'' 6th USENIX Security Symposium, pp. 37–42. San Jose, CA, July 1996.

Appendix: Summary of the PAK protocol

Let $q > 2^{160}$ and $p > 2^{1024}$ be primes such that $p = rq + 1$ with r not a multiple of q. Take $h \in Z_p^*$ such that $g \equiv h^r$ is not 1. These parameters may be chosen by the NIST algorithm for DSA, and are public, fixed values. The client C knows a secret π and computes $H \equiv (H_1(C, \pi))^r$ and H^{-1}, where H_1 is a hash function yielding a random element of Z_p^*, and H^{-1} may be computed by gcd. (All arithmetic is modulo p.) The client gives H^{-1} to the server S ahead of time by a private channel. To start a new connection, the client generates a random value x, computes $m \equiv g^x H$, then calls the server and sends C and m. The server checks $m \neq 0 \mod p$, generates random y, computes $\mu \equiv g^y$, $\sigma \equiv (mH^{-1})^y$, and sends S, μ, $k \equiv sha1(\text{"server"}, C, S, m, \mu, \sigma, H^{-1})$. Next the client computes $\sigma = \mu^x$, verifies k, and sends $k' \equiv sha1(\text{"client"}, C, S, m, \mu, \sigma, H^{-1})$. The server then verifies k' and both sides begin using session key $K \equiv sha1(\text{"session"}, C, S, m, \mu, \sigma, H^{-1})$. In the published version of PAK, the server name S is included in the initial hash H, but doing so is inconvenient in our application, as the server may be known by various equivalent names.

MacKenzie has shown [Mack] that the equivalence proof [Boyk00] can be adapted to cover our version.