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

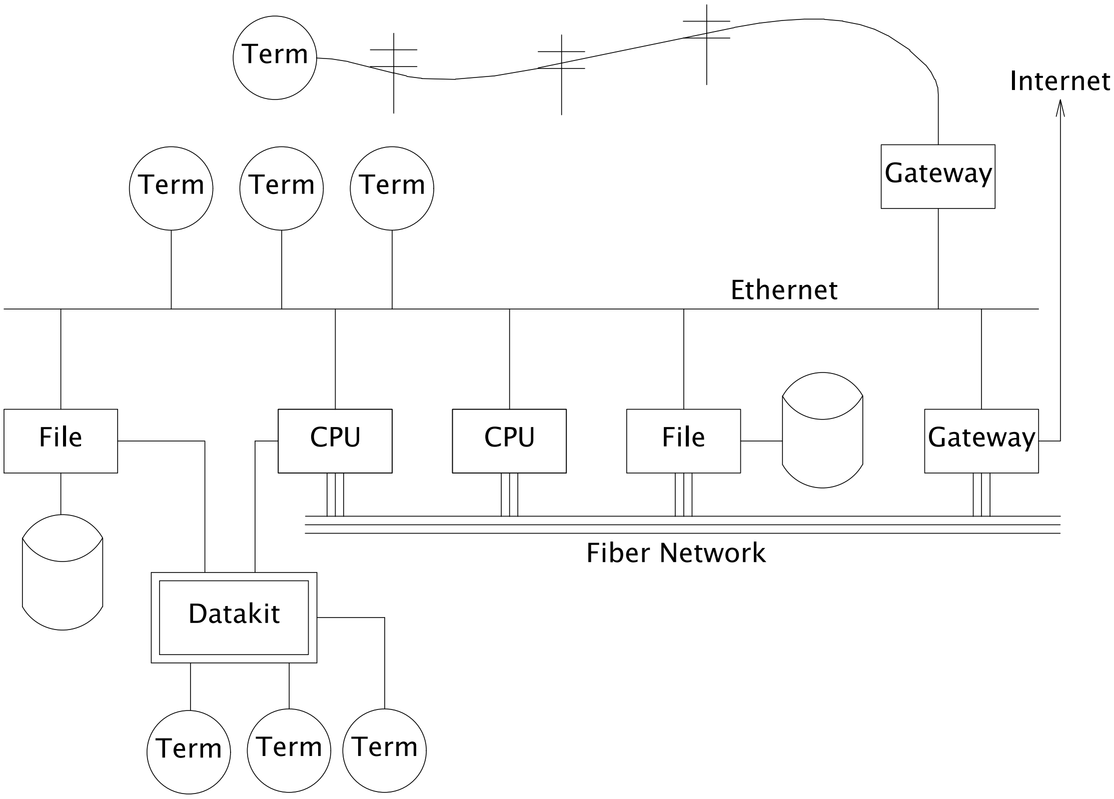

Figure 1. Structure of a large Plan 9 installation. CPU servers and file servers share fast local-area networks, while terminals use slower wider-area networks such as Ethernet, Datakit, or telephone lines to connect to them. Gateway machines, which are just CPU servers connected to multiple networks,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