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