Plan 9, A Distributed System

Dave Presotto
Rob Pike
Ken Thompson
Howard Trickey

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Plan 9 is a computing environment physically distributed across many machines. The distribution itself is transparent to most programs giving both users and administrators wide latitude in configuring the topology of the environment. Two properties make this possible: a per process group name space and uniform access to all resources by representing them as files.

1. Introduction

Plan 9 is a general-purpose, multi-user, portable distributed system implemented on a variety of computers and networks. Because commands, libraries, and system calls are similar to those of the Unix operating system, it is possible to port many Unix programs to Plan 9 with little or no changes. A casual user would find little difference between the two systems.

What distinguishes Plan 9 is its organization. The goals of this organization were to reduce administration and to promote resource sharing. Our programming style was minimalism. We believe that a small number of well-chosen abstractions can, with much less code, provide most of the function of a larger system. This is the approach that made the Unix operating system so much smaller than its contemporaries such as Multics. In building Plan 9, we generalized proven ideas from the Unix operating system rather than add new untried concepts.

Plan 9 is divided along lines of service function. Diskless CPU servers concentrate computing power into large multiprocessors; file servers provide repositories for storage; and terminals give each user of the system a dedicated computer with bitmap screen and mouse on which to run a window system. The sharing of computing and file storage services provides a sense of community for a group of programmers, amortizes costs, and centralizes and hence simplifies management and administration.

Since both CPU servers and terminals use the same kernel, users may choose whether to run programs locally on their terminals or remotely on CPU servers. Plan 9 provides this flexibility without constraining the choice. Therefore, both users and administrators can configure their environment to be as distributed or centralized as they wish. At work, users tend to use their terminals more like workstations running all interactive programs locally and reserving the CPU servers for data or compute intensive jobs such as compiling and computing chess end games. At home, connected via a dedicated 9600 baud line to work, users choose what they run locally and remotely to reduce communication cost. Some applications, such as the editor [Pik87], are split into multiple programs to make this choice even more flexible.

Figure 1 in any Plan 9 paper shows how we have configured our environment. Multiprocessor CPU and file servers are clustered in a few computer rooms and connected via 7 megabyte/sec point-to-point links [Pre88]. This permits the CPU servers to be used as high performance compute engines without becoming starved for data. Terminals are connected to the servers via lower speed, lower cost distribution networks such as the 10 megabit Ethernet [Met80] and 2 megabit Incon [Kal, Res]. By emphasizing the shared service clusters we can quickly and cheaply incorporate new technologies as they arise. At the same time, users wishing more autonomy can incorporate as much computing power as they wish in their own offices without losing the advantage of transparently sharing other resources.

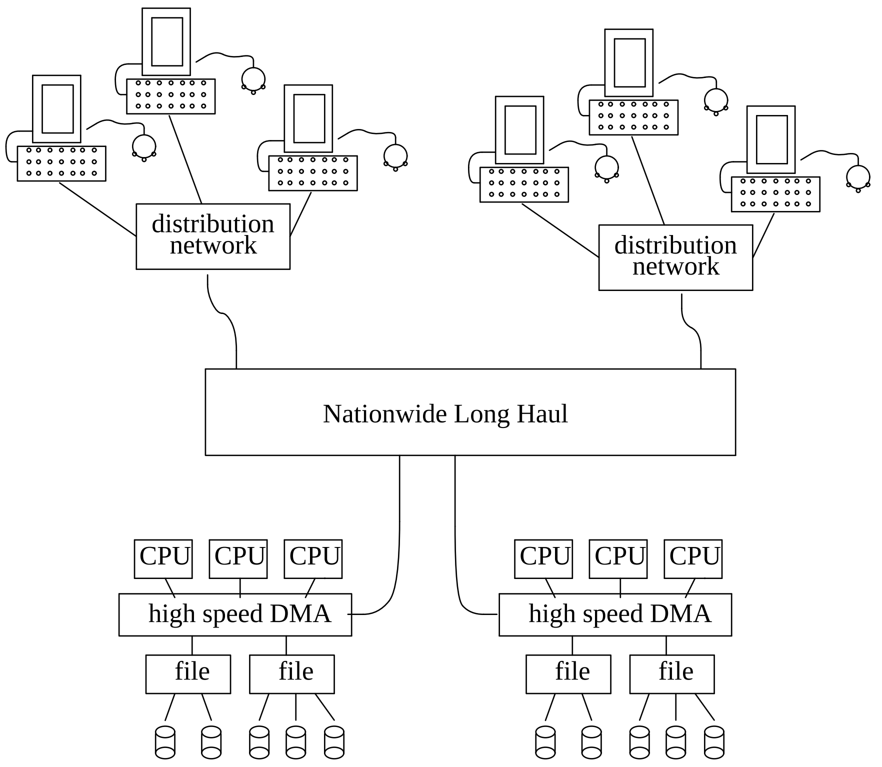

Figure 1 - Plan 9 Topology

The rest of this paper describes the features of Plan 9 that make possible such a flexible topology. For more information on hardware and use of the system, see our previous paper [Pik90] . For details of the file server, see [Qui] .

2. Minimalism

All resources that a process can access, aside from program memory, reside in one name space and are accessed uniformly. Simply stated, all resources are implemented to look like file systems and, henceforth, we shall call them file systems. A file system is a strict tree with no links. File systems can be the traditional type representing persistent storage on a disk as implemented by the shared file servers. They can also represent physical devices such as terminals or complex abstractions such as processes. The file systems can be implemented by kernel resident drivers, by user level processes, or by remote servers.

A file system representing a physical device normally contains one or two files. For example, an RS232 line is represented as a directory containing a `data` and a `ctl` file. The `data` file is the stream of bytes transmitted/received on the line. The `ctl` file is a control channel used to change device parameters such as baud rate.†

Some file systems represent software concepts. Environment variables (as in Unix) are implemented as files in a kernel resident file system. Even processes themselves are represented as directories with separate files representing different aspects of the process such as memory, text file, and control. Many things that require a system call in other operating systems are represented by I/O operations on files in Plan 9;

† We neither need nor have an `ioctl` system call.

reading the id of a process, the user id associated with a process, the time, etc.

A kernel data structure, called a *channel*, is used as a pointer to a file. A user level file descriptor is just a handle for a kernel channel. All I/O system calls eventually translate into nine primitive operations on channels. They are:

attach	– point a channel to the root of a file system. The file system is told which user is attaching.
clone	– make a copy of a channel. The new channel points to the same file as the old one.
walk	– do a one level directory lookup on the channel and point it to the new file (or directory).
stat	– get the attributes of the file pointed to.
wstat	– change the attributes of the file pointed to.
open	– check permissions prior to I/O on the channel.
read	– read from the opened file.
write	– write to the opened file.
close	– close the opened file.

Each kernel resident file system is implemented by a *device driver* containing a procedure for each primitive operation. The device drivers are accessed indirectly via a kernel array, `devtab,` which contains 9 pointers per driver, one to each primitive procedure. Each channel contains an offset into `devtab` indicating the driver to be used in accessing the file it points to.

Accessing file systems not resident in the kernel is via a special device driver, the *mount driver.* All channels pointing to this driver contain a pointer to a communication channel. The mount driver turns operations on such channels into request messages written to the communication channel. The mount driver is written as a multiplexor allowing multiple outstanding messages. Because the messages on the communication channel are transmitted using `read's` and `write's,` any type of channel can be used: a pipe to a process, a network connection, even an RS232 line. The `mount` system call, described below, is used to create a new mount device channel and supply a communication channel for it.

All Plan 9 components are connected using this file system protocol. The code used to encapsulate the primitives into request and reply messages is 580 lines long. The mount driver is 899 lines long. Compared to the equivalent NFS code implementing vnodes and XDR this is tiny.

Of the 18000 lines of code that make up Plan 9, about 5000 lines perform memory management, process management, hardware interface, and system calls. The rest are for the 17 different file systems implementing devices, networks, process control, etc. Since most of the file systems are completely self contained, the complexity of the kernel code is even lower than its 18000 lines would imply. A working, albeit not very useful, kernel can be configured containing only the file systems implementing pipes, a local root, and a console. This totals 5899 lines of commented C code (counted using `wc *.[ch]`). As a comparison, Mach's micro-kernel without device drivers has 25530 lines of C code (calculated, we're told, by counting semi-colons). By the same metric our minimal kernel is only 4622 lines long, less than 1/5 the size. In fact, our kernel with every file system included is still less than half the size of their micro-kernel.

One might note the similarities between `devtab` and parts of the Unix operating system; the block device switch, character device switch, file system switch and vnodes. One advantage of Plan 9 is that we have recognized that these are all essentially the same mechanism and have implemented them as such.

3. Virtual Name Space

When a user boots a terminal or connects to a cpu server, a new process group is created for her processes. This process group starts with an initial name space that provides at minimum a root (`/`), some binaries for the processor the process is running on (`/bin/*`), and some local devices (`/dev/*`). The processes in the group can then either add to or rearrange their name space using two systems calls, `mount` and `bind.` The mount call is used to attach a new (not kernel resident) file system to a point in the name space. Its syntax is

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

where *fd* is a file descriptor for a communication stream such as a pipe or a network connection and *old* is the name of an existing file in the current name space where the file system will be attached. The attachment creates a new mount device channel whose communication channel is that referred to by *fd.* Subsequent accesses to *old* and any files below it in the hierarchy become request messages written to the communication stream.

The bind call is used to attach a kernel resident file system to the name space and also to rearrange pieces of the name space. Its syntax is

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

where *new* is a name in the current name space† and *old* is the same as in mount.

How the attachment works depends on the *flags* specified in the call. One possibility is that the old file is replaced by the new one. However, when both files are directories, Plan 9 allows another possibility. The result can be the union of the two directories. The effect is that of putting one directory behind the other. In the case of name conflicts for files contained in the directories, the one in front wins. *Flags* specifies whether the new directory replaces, goes in front of, or goes behind the old one. This concept is essentially the same as the search paths used in the Unix libraries and the various shells. In fact, Plan 9 has no search paths and uses these *union directories* in their place. When a command is executed, Plan 9 uses the directory `/bin` the same way Unix uses an execution path.

The ability to specify the complete name space for a process that contains all resources the process can access forms the basis for a true virtual machine. Any aspect of a process' world can be rearranged. Remote objects can be substituted for local ones. Processes can implement part or all of the name space of other processes. This capability is the basis for a number of important services, three of which we present here.

3.1. The Cpu Command

We consider the shared CPU servers as accelerators for our terminals, someplace where commands can run while maintaining the same environment. It is important that as little as possible change when running on the CPU server. The virtual name space provides us with a means to make the CPU servers actually feel this way to our users. A command, `cpu,` calls a CPU server across a network. A daemon process on the server answers the call, creates a new process group for the caller, sets up a name space, and starts a shell process in the new process group. The name space set up is an analogue of the name space of the calling process on the terminal. In particular, local resources on the terminal, such as the screen and the mouse, become visible to the server processes at the same place in the name space as on the terminal. The standard input, standard output, standard error, and current directory of the cpu command become those of the remote shell. The directories mounted on `/bin` are changed to be those that contain executables for the CPU server's processor type (the terminal may be a 68020 while a CPU server could be a MIPS). In general, a user typing the cpu command just notices that things such as compilations speed up while graphics operations slow down.

After the initial handshake to pass information describing the caller's environment, the cpu command becomes a file server answering file system requests from the network connection. The server daemon mounts the network connection to the terminal in a standard place, `/mnt/term,` and then binds the resources it decides to keep into the same places in the new name space. For example, it binds `/mnt/term/dev/mouse` onto `/dev/mouse,` `/mnt/term/dev/bitblt` onto `/dev/bitblt,` etc. Subsequent accesses to those files are converted by the mount driver in the CPU server into file system messages sent to the terminal.

† Local kernel resources are referred to by a syntactic escape (hack) in the name space. Any name starting with a ''#'' refers to a local resource. The first character following the ''#'' specifies the type of resource and the remaining characters are a parameter specifying the instance of the resource. Thus, to bind the local console to a standard place in the name space, one would use `bind("#c", "/dev", FRONT).`

3.2. The Window System

The user interface is made up of three files:

`/dev/bitblt` – writes represent bitblt operations to the screen

`/dev/mouse` – reads return mouse events, i.e., button clicks and movement

`/dev/cons` – reads return keyboard input, writes put characters to the screen.

Between them, these devices represent all I/O to the user. The window system, 8.5 [Pik91], offers processes a multiplexed view to these devices. When a window is opened, the window system starts a new process group for a command (usually a shell) that will run in that window. In that process group's name space, the window system mounts a pipe to itself in front of `/dev.` Subsequent references by the new process group to any of these devices are sent as file system messages to the window server. 8.5 interprets those requests as accesses of the window instead of the whole screen. Similarly, 8.5 multiplexes the mouse and the keyboard so that mouse and keyboard input is available to processes only when their window is selected.

The result is that any program written to use the kernel resident user interface will also work inside a window. Because this is also true of the window system itself, new versions of the window system can be run and debugged in windows of the current window system.

3.3. Network Gateways

One, sometimes insurmountable, problem is accessing a network to which a system is not physically attached. For example, a system may be connected to our Datakit [Fra80] network but not to the DoD Internet. Many gateways exist that try to solve this problem by performing protocol to protocol translation. Unfortunately, few transport protocols have completely equivalent concepts. In order to perform the best translation, it is be necessary to know the semantics requested by the program. For example, TP4 has message delimiters but TCP does not. A protocol translator going from TCP to TP4 would not know which bytes correspond to a single write by the sender.

In Plan 9, every network interface is a file system. A gateway is a file server that serves its own network interfaces to other machines. A process that wants to get at a remote network connects to the gateway and mounts the gateway's interface to the remote network into its name space. Whenever the process accesses the interface, the mount driver will send the request to the gateway. Thus, the gateway sees exactly what the process does.

4. File Caching

In building our environment, we've been reluctant to add local disk file systems to any of our terminals or CPU servers. There are essentially two reasons for this choice. The first is administration. Anyone with a local disk must administer it. Any disk that has unique long term state requires both knowledge and time to administer. In fact, the Bell Labs computer center at Murray Hill is doing a lucrative business maintaining other peoples' disked Sun workstations because the owners have neither the time nor the experience necessary to do it themselves.

The second reason is sharing. Although most workstations can export access to their local file systems, when left up to individual users, this rarely happens. Terminals become personified and users become tied to a particular room to do their work.

Plan 9 survives without local disk file systems thanks partially to hardware and partially to caching. The CPU servers do so because their links to the file servers transfer at a substantial percentage of memory speed. The file servers maintain large main memory caches for their disk file systems. These servers are configured with 128 megabytes or more of main memory to ensure that there is plenty of room for cache. Getting a file from a file server is generally faster than it would be to get it from a local disk.

Office terminals are connected to the file servers by shared 1 or 10 megabit/sec links. Home terminals use 9600 or 19200 baud links. In both cases, the link is much slower than access to a local disk would be. To avoid the obvious performance hit, we use caching. To keep the caches coherent, we use file identifiers supplied by the file server. The identifiers are unique 64 bit quantities. 32 bits identify the file, the other 32 bits identify the version of the file. The version number is incremented each time the file is modified. Each time a file is opened the file server returns the identifier with the reply. Therefore, it is possible to guarantee coherency at each opening of a file.

Office terminals only cache pages of executable files. Whenever a program terminates, its unmodified text and data pages are not immediately freed. Instead they are retained until the space is required by other programs. When a program is rerun its executable file is reopened and the current version number returned. If the version number has not changed and pages remain from the last run, they are reused. If the version number has changed, any remaining pages of the stale version are discarded. Since most data intensive work is done on the CPU servers, this simple cache saves most of the traffic between office terminals and the file servers. Other caching could be helpful but would require much more complexity.

This cache might also have sufficed for home terminals if it were persistent, but it is not. Therefore, we have added disks to our home terminals to be used as write through caches of the file server files. As a write through cache, it contains no state that isn't duplicated on the file servers. Therefore, it needs little maintenance compared to a local file system. If the code discovers a disk problem, it reformats the disk discarding the current contents. If the user should suspect that the cache is contaminated, she can request that it be reformatted at the next boot. The system slows down until subsequent use refills the cache but no information is lost. The user need not consciously update the disk because the cache uses file identifiers to maintain coherency with the file servers. Each time a file is opened, the cache discards any stale data it might have for that file. The user doesn't have to copy what she needs to the disk because it is done as a consequence of her using the data.

The disk based cache is implemented by a process that resides between the kernel and the file server connection. For every read request, the process satisfies as much as it can with data cached on the disk. It gets the rest from the file server. Any new data that passes through it is saved on the disk. When the cache fills up the least recently used file is discarded. The amount of data cached for any one file is limited to 1.75 megabytes to prevent one file from displacing all others.

Because the disk based cache only checks for coherency when a file is opened, it provides slightly different semantics than that seen on office terminals which do not cache data files. This looser coherency constraint forces programs that communicate via files to ensure an open between each transaction. Thus far we have not had to change any programs because of it.

5. Conclusion

We have presented a distributed system that is simple in structure and flexible in its use. Both the flexibility and simplicity are the result of two properties, a per process group name space and a single resource interface. Coupled with some minimal caching we provide a simple system that is as usable at home as at work.

6. Acknowledgements

Many people helped build the system. We would like especially to thank Bart Locanthi, who built our terminal, the Gnot, and encouraged us to program it; Tom Duff, who wrote the command interpreter `rc`; Tom Killian, who built and programmed the Gnot's SCSI interface; Ted Kowalski, who cheerfully endured early versions of the software; and Dennis Ritchie, who frequently provided us with much-needed wisdom.

References

Fra80. A. G. Fraser, ‘‘Datakit–A Modular Network for Synchronous and Asynchronous Traffic,’’ in *Proc. Int. Conf. on Commun.*, Boston, MA (June 1980).

Kal. C. R. Kalmanek, ‘‘INCON: Network Maintenance and Privacy,’’ Internal Memorandum 220106-0450, AT&T Bell Laboratories.

Met80. R. Metcalfe, D. Boggs, C. Crane, E. Taft, J. Shoch, and J. Hupp, ‘‘The Ethernet Local Network: Three Reports,’’ CSL-80-2, XEROX Palo Alto Research Centers (February, 1980).

Pik91. Pike, R., ‘‘8.5, The Plan 9 Window System,’’ *1991 USENIX Summer Conference Proceedings* (1991).

Pik87. Rob Pike, ‘‘The Text Editor sam,’’ *Software - Practice and Experience* **17**(11), pp. 813-845 (November 1987).

Pik90. R. Pike, D. Presotto, K. Thompson, and H. Trickey, ‘‘Plan 9 from Bell Labs,’’ in *UKUUG Proceedings of the Summer 1990 Conference*, London, England (July, 1990).

Pre88. D. Presotto, ‘‘Plan 9 from Bell Labs - The Network,’’ in *EUUG Proceedings of the Spring 1988 Conference*, London, England (April, 1988).

Qui. S. Quinlan, ‘‘A Cached WORM File System,’’ *Software – Practice and Experience*, p. To appear.

Res. R. C. Restrick, ‘‘INCON Wire Interface Integrated Circuit Design,’’ Internal Memorandum 52413-860314-01TM, AT&T Bell Laboratories.