The Ubiquitous File Server in Plan 9

C H Forsyth

Vita Nuova Limited
3 Innovation Close
York Science Park
York England YO10 5ZF
forsyth@vitanuova.com
20 June 2005

1. Introduction

Plan 9 is a distributed system begun in the late 1980s by the Bell Labs research centre where Unix was born.[1] Rather than having each computer in a network act as if it still controlled everything, Plan 9 is designed to allow a single large system to be built from smaller cooperating systems performing specific tasks, notably file service, authentication, cpu service and interactive graphics. Although all system components can indeed run on a single machine, a typical development environment has a network of cheap single-user diskless terminals (usually PCs without disks these days) sharing permanent file storage and multi-processor cpu servers (often also diskless). Even the file storage implementation has two parts: a fairly conventional hierarchical file system stores data in a separate write-once block archiving service, and they can usefully run on separate machines. An authentication server runs only a special set of trusted processes to do authentication for the network, and implements only the few relevant protocols. Other cpu servers might provide resources for big computations or compilations, or provide SMTP, FTP and HTTP services to internal and external networks. A Plan 9 terminal runs programs that implement the interactive interface; it is not just a remote display as in some 'thin client' schemes.

Plan 9 supports a range of processor types, and it is possible to mix and match within a network: Sparc, MIPS, Alpha, PowerPC and Intel. In the past it was common to use big multi-processor MIPS machines as cpu servers with various other architectures as terminals. Currently of course most implementations use computers based on Intel or AMD processors because they are cheap and available, but ARM and PowerPC still have a place, not least in the embedded realm, and the system source is kept portable.

The system has some unusual elements, described in previous papers: its networking subsystem,[2,3] its window system,[4] the Acme programming environment,[5] the archival storage subsystem (Venti),[6] the archival file store above that (Fossil),[7] the earlier WORM-based file server,[8,9] the language-driven message exchange (plumber),[10] its C compiler suite,[11] its language-based debugger (Acid),[12] details of its kernel implementation,[13,14] its real-time support,[15] and its authentication subsystem.[16] Some aspects have been adopted by other systems: the Unicode representation UTF-8, now in common use, was originally developed for Plan 9.[17] Plan 9's defining novelty, however, remains its representation of resources in a distributed system.[18]

Libre Software Meeting, July 5-9, 2005, Dijon, France.

2. Resources as files

In Plan 9, system resources are accessed through a hierarchical *name space*: a tree of names, similar to a Unix file hierarchy, with directories at interior nodes and files at the leaves. Both files and directories have user/group/other access permissions and a few other attributes.

Resources represented this way include:

- devices (from UARTs to network interfaces)
- networks
- protocols
- services
- control and debugging of processes
- system management
- service discovery and naming
- graphics
- permanent storage structures (on disk, flash, memory, and network)

Operations on resources that would be specialised system calls or object methods in other systems are instead implemented using patterns of open- read- write- close on files and directories within the name space representing the resource. Examples are given below.

Unusually, but critically for the design, there is not a single global name space: instead a process or group of processes assembles the name space it needs for the application it implements, such as the user interface or a network gateway. The name space is not derived from a permanent copy but is *computable*: built and rearranged dynamically using three system calls: `bind`, `mount` and `unmount`. (There is a further primitive that controls sharing of name spaces between processes, discussed below.)

The `bind` call:

```
int bind(char *name, char *old, int flag)
```

takes the names of two existing files or directories. It makes the thing that *name* refers to (file or directory) visible at the name *old*. By default, *name* hides the previous contents of *old*; if both are directories, however, the *flag* can make *old* into a *union directory* that includes the contents of *name* either before or after the contents of *old*. Thus,

```
bind("/usr/forsyth/lib", "/lib", MBEFORE);
```

searches my private library before the system one. (The 'union' is actually a sequence of the names in the relevant directories, not a union of complete substructures.) The `unmount` call:

```
int unmount(char *name, char *old)
```

undoes the effects of a previous `bind`.

A *file server* in Plan 9 is any program that manufactures a name space by implementing the server-side of the system's file service protocol, 9P. Most of the significant functionality in the system, including devices, networks and graphics, is provided by file servers. The `mount` system call:

```
int mount(int fd, int afd, char *old, int flag, char *aname);
```

mounts the name space generated by the file server on file descriptor *fd* on an existing directory *old* in the current name space, either adding to or replacing the existing contents of the chosen mount point (as determined by *flag*, just as for `bind`). The *aname* is a string interpreted by the file server; some use it to select from a set of possible trees implemented by the server. Client programs subsequently access the server using the familiar system calls on that part of the resulting name space: a kernel component (the 'mount driver') converts file system operations in that space to 9P messages on the file descriptor, which are then acted upon by the file server.

One user-level file server, `exportfs`, allows a name space to be *exported* on a file descriptor. `Exportfs` reads 9P requests from the file descriptor and executes corresponding system calls in its own name space, sending the results of the system calls in 9P replies. That one mechanism provides the basis for distribution of all resources and services.

9P was originally introduced to allow clients to share a conventional file system, much as with Sun's Network File System.[19] As Pike and Ritchie observe,[20] the design breakthrough for Plan 9 was to realise that given 9P, once resources are implemented as file systems, they can readily be exported, uniformly, building a distributed system with less fuss.

3. Designer's names

Although Plan 9 has the usual collection of Unix-like commands and filters, it offers the new possibility of implementing system and application functions as file servers. Indeed, even the kernel representation for devices, services, network interfaces, and protocols is the same: they are all kernel-resident file servers. Thus the design of a given component, whether device driver, system service or application, often begins by designing a suitable name space, at a level of abstraction above that of (say) the API for any particular programming language. In other words, in Plan 9, the name space provides the focus for design. We look at a reasonable collection of examples, then describe the underlying protocol that links everything together.

Example: network interfaces

Plan 9 does not provide special 'socket' system calls to access networks. Instead, devices, interfaces, protocols and services are all represented by file trees, conventionally collected together under `/net`. Figure 1 shows a subset of the `/net` directory on my Thinkpad terminal. Most of the entries are from an instance of the file server that represents an Internet protocol stack (not all its protocols are shown above), but `ether0` represents the wired Ethernet connection, and `ether1` represents my wireless connection, both provided by separate instances of the file server type that represents Ethernet devices.

The `ether` directories show a common Plan 9 naming convention. A file server that multiplexes many connections represents each connection as a numbered directory, each with a `ctl` file to control that connection, and `data` to access that connection's data or messages. The `ctl` file when read contains the number of the directory in decimal (ie, as text). Finally, opening the `clone` file is equivalent to opening the `ctl` file of an unused or newly-allocated connection directory. Thus, to get a new connection, open the `clone` file, and read it to find the directory number and access the other files. Textual messages on the `ctl` file do the work of special system calls in other systems. The file ownership and permissions prevent undesired access to the connections of other users on multi-user systems.

The `ether` connections correspond to different ethernet packet types; the type for a given connection is set by writing `connect` *n* to its `ctl` file. Packets of that type can then be read and written as binary messages in standard Ether format on the `data` file. The file `/net/ether0/addr` contains the MAC address of the interface.

The `ipifc` directory configures network interfaces into the Internet protocol subsystem. A new interface is allocated by opening the `clone` file, and writing a control message to *bind* a given medium's interface into the IP stack, in a way appropriate to the medium. For instance:

```
echo bind ether /net/ether0 >/net/ipifc/clone
```

will allocate a new connection in `ipifc`, and use `/net/ether0` as a device that adheres to the rules for an `ether` medium. Other media include `pkt`, which allows a user program to read and write the connection's `data` file to send and receive IP packets, and `netdev`, for a device that consumes and produces IP packets. The `pkt` interface can be used for PPP and NATP implementations. Subsequently, other control messages can be written to set Internet addresses and masks for the interface:

```
/net/
        arp
        ether0/
                addr
                clone
                ifstats
                stats
                0/
                        ctl
                        data
                        ifstats
                        stats
                        type
                ...
        ether1/
                ...
        ...
        ipifc/
                clone
                stats
                0/
                        ctl
                        data
                        err
                        listen
                        local
                        remote
                        snoop
                        status
                ...
        iproute
        tcp/
                clone
                stats
                0/
                        ctl
                        data
                        err
                        listen
                        local
                        remote
                        status
                1/
                        ...
                ...
        ...
```

Figure 1. Subset of `/net` on a terminal with several Internet network interfaces.

```
echo add 144.32.112.70 255.255.254.0 0 >/net/ipifc/0/ctl
```

Other properties can also be set by control messages. The current routing table for the interface is presented as text on `iproute`, which also accepts textual messages to change it.

Individual protocols, as configured, have their own subdirectories in the IP interface directory, thus `tcp`, `udp` etc. Outgoing connections are made by writing `connect` *address*`!`*port* to the `ctl` file. The write gives a diagnostic if the connection cannot be made; otherwise, read and write the `data` file to exchange data on the connection. A process listens for incoming calls by first writing `announce` *port* to `ctl`, and then opening the `listen` file. The open blocks until an incoming call is made. It then returns a file descriptor open on the `ctl` file for a new directory for that connection. For an active connection, the `local` and `remote` files contain text representing its endpoint addresses.

Only one instance of the IP stack server appears above, and that is the usual case, but there can be

more than one, each serving a complete and self- contained IP stack. Firewalls and gateways often use more than one stack, each serving different sets of physical interfaces, such as an 'inside' and 'outside' Ether, with different sets of services offered on each. Packets cannot ordinarily move from one interface to another, but a filtering firewall can mount both interfaces in its name space, give itself a `pkt` interface on each stack, and copy packets between them, filtering and adjusting addresses as required.

Example: domain name service

The Domain Name System implementation on Plan 9 is a file server, `ndb/dns`. It serves one file, which conventionally appears at `/net/dns`. An application that wishes to translate between a name and resource opens that file, writes the name and resource type (as a single line of text), and reads successive possible translations. If the name cannot be translated, the write returns an error. A small utility program `ndb/dnsquery` allows us to work some examples:

```
term% ndb/dnsquery
> www.vitanuova.com
www.vitanuova.com ip    193.195.70.8
> vitanuova.com mx
vitanuova.com mx        10 smtp.vitanuova.com
> www.cnn.com
cnn.com ip      64.236.16.52
cnn.com ip      64.236.16.116
cnn.com ip      64.236.24.20
cnn.com ip      64.236.24.12
cnn.com ip      64.236.24.4
cnn.com ip      64.236.16.20
cnn.com ip      64.236.24.28
cnn.com ip      64.236.16.84
> 64.236.16.52
52.16.236.64.in-addr.arpa ptr   www4.cnn.com
> www.wotsmyname.com
!dns: name does not exist
```

One advantage of this implementation is that the cache of DNS translations is quite naturally shared by all applications, without special handling. As it happens, the DNS service is most often not used directly by applications, but by means of the more general *connection service*, described next.

Using a file server to hold shared state, as `dns` does, is a common one. For example, the mail ratification service `ratfs` provides a persistent representation of the system's spam blocking list, serving a name space that allows the list to be queried by many concurrent instances of *smtpd*, and updated by simple shell scripts using *echo*, or more elaborate spam filters, simultaneously. As another example, a Plan 9 Usenet news server maintains the history of article IDs as an in-memory database, and serves files such as `msgid` and `post` that allow efficient query and update by many clients that are filing different incoming news streams.

Example: connection service

The connection service `ndb/cs` translates a symbolic network address into a set of recipes for connecting to that service. Network addresses in Plan 9 are text with a standard form:

[*network* `!`] *netaddr* [`!` *service*]

Network identifies a particular network type or protocol, such as `il` (a reliable datagram protocol), `tcp`, `udp`, `telco`, `ether0`, and others. In general, it is the name of a directory under `/net`, so the set is extensible. The *network* name `net` is the default, and means 'any supported network'. *Service* is often a symbolic name, even when the protocol itself uses port numbers. This syntax works for IP, datakit, x.25, atm, IrDA, ether, telephones, and others. The connection service uses its knowledge of the available networks and underlying naming systems (such as DNS

and Plan 9's own network database) to work out possible ways of converting the components of the symbolic address to physical network addresses. It serves a single file, conventionally `/net/cs`. A client writes to the file the symbolic name of the service desired; each subsequent read returns a set of instructions for one possible way of making the connection, expressed as operations on other files in the name space.

For example, to translate the symbolic address `net!doppio.bloggs.com!9fs` a client process opens `/net/cs`, writes that string to it, and if the write succeeds, reads back a list of recipes, as text, one for each read:

```
/net/il/clone 200.1.1.67!17008
/net/tcp/clone 200.1.1.67!564
```

Each is interpreted as 'open the file named by the first field and write the value of the second field into it'. The effect is to make a network connection to the desired service. There can be several such recipes (one per read) if the destination address has more than one translation (eg, a host with several network interfaces). In the case above, the IL/IP protocol can be used as well as TCP/IP; that will be attempted first. Note that the server can distinguish opens by different clients, and thus a client sees only the recipes for its own requests. If a name cannot be translated, the server returns an error to the client on the initial write.

When an Internet domain name appears in an address, *cs* uses `/net/dns` described above to translate it to one or more numeric Internet addresses.

Example: mail boxes

Access to mail boxes is through a name space presented by the program `upas/fs`. Each mail box is represented by a directory in the server's name space, conventionally mounted on `/mail/fs`. It serves up mail boxes in the structure shown below:

```
/mail/fs/
        ctl
        mbox/
                1/
                        bcc
                        body
                        cc
                        date
                        digest
                        disposition
                        filename
                        from
                        header
                        ...
                        raw
                        rawbody
                        rawheader
                        ...
                2/
                        bcc
                        body
                        cc
                        ...
                        1/
                                bcc
                                body
                                ...
                        2/
                                bcc
                                body
                                ...
```

Each message appears as a directory. The files in each message directory contain the parsed elements of the body and header of the corresponding message. For instance:

```
% cat /mail/fs/mbox/782/subject
Re: [iPAQ] A working 802.11 / AP combination?
```

Access to the unparsed bytes of body, header or the whole message is provided through the several `raw` files. The `ctl` file in the root directory accepts requests to delete one or more mail messages, or the message directory can simply be deleted by the `remove` system call (or `rm` command). Attachments are represented by subdirectories with the same structure, and so on recursively.

Mail reading applications are easy to write, because the correct parsing of the RFC822 mail header, interpretation of the MIME structure, and character set conversion, all of which is non-trivial, is done centrally by `upas/fs`. For instance, there is a little library of shell functions that supports virus and spam detection, using `grep -s` and other commands, including specialised ones such as `upas/bayes`, applied to files in the mailbox name space.

Furthermore, several mail readers can access the same mail box simultaneously with consistent results as new messages arrive and others are deleted. Note that although `upas/fs` presents a file- systemlike view of its internal mail box representation, that representation is not actually a collection of files and directories on disc, and the contents of each part of the hierarchy is provided only in response to a client's request. For instance, when a client reads the `subject` file, a message is sent on its behalf to `upas/fs`, which sends a reply message containing a value copied directly from an internal string value that holds the parsed subject line. Directory entries are similarly generated on demand.

Example: authentication

Two file servers support authentication. One runs only on the authentication server, in a secured environment. The other runs on other Plan 9 nodes and acts as the authentication agent for users and servers, holding both sets of keys and all knowledge of authentication protocols.

`Keyfs` runs on the authentication server(s) for a Plan 9 authentication domain. The current Plan 9 authentication protocol is based on secret keys (eg, pass phrases). The secrets are stored, labelled by user, in a record- orientedfile, encrypted by a master key. (In fact, a hash of the secret is stored, not the original value.) `Keyfs` decrypts the file and serves the following name space:

```
username/
        key
        log
        status
        expire
...
```

In the two- level hierarchy, the top level directories are named after the registered users (eg, `forsyth`, `rog`, etc). The authentication data for each user is found in the corresponding directory. The file `key` contains the authentication key for that user (eg, a 'shared secret'). If the account is disabled, a read request returns an error. A key can be changed by writing to the file. The account's expiration date can be read or written in the file `expire`. The `keyfs` hierarchy is mounted on `/mnt/keys` in a name space created to run only authentication services. They can access the authentication data for users, but do not know the master key. Furthermore, several services can access the data simultaneously without risking its corruption, because `keyfs` acts as the master file's monitor. Applications in all other name spaces see an empty directory at `/mnt/keys` (because `keyfs` is not mounted in those name spaces).

`Factotum` is a general authentication agent, with at least one instance per user. It is the only application in the system that knows the details of a dozen or so authentication protocols, including Plan 9's own, and others such as `apop` or `ssh`. It serves the following two- levelname space, typically union- mounted on`/mnt`:

```
factotum/
        rpc
        proto
        confirm
        needkey
        log
        ctl
```

The `proto` file lists the authentication protocols implemented by this instance of `factotum`. The `ctl` file is written to install and remove keys:

```
cd /mnt/factotum
# add a key:
echo key 'user=badwolf' 'service=ftp' 'server=unit.org.uk' \
        '!password=buffalo' >ctl
# remove all matching keys:
echo delkey 'server=unit.org.uk' >ctl
```

When an application wishes to authenticate to another service, it opens the `rpc` file and follows a simple protocol (implemented by a C library function). The client writes `rpc` to tell `factotum` the desired authentication protocol, user, service etc., and then it acts as a proxy, forwding messages between the authenticating service and `factotum`, until authentication is complete. Only `factotum` knows the protocol details: the application simply follows `factotum`'s instructions on the phasing of read and write, and data contents. Furthermore, for secure protocols, the application cannot see the user's keys.

Of course, not all keys will be preloaded in general. It is sometimes necessary to prompt for them. The `needkey` file can be opened by another program, typically `auth/fgui`, and read to

receive requests for keys. The read blocks until `factotum` needs a key it does not yet hold; it then replies to the read with a description of the desired key, `fgui` prompts the user in any way desired and writes the key (if the user supplies one) to `/mnt/factotum/ctl` as above.

Example: window management

Graphics is ultimately provided through the `draw` device, a kernel file server that serves a three-level hierarchy:

```
draw
        new
        0/
                ctl
                data
                colormap
                refresh
        1/...
        ...
```

Each connection to the device is represented by a numbered directory, following a similar scheme to the network devices. A bitmap graphics application opens `/dev/draw/new` to allocate itself a new connection, reads the connection number, and opens the other files as needed. The `data` file implements the graphics protocol: messages are read and written by the client to allocate and free images and subfonts, draw on images (on and off screen), and so on. The `draw` device uses a variant of Pike's *layers*[21] to allow many applications to write to overlapping windows on the screen without confusion, thus multiplexing graphics output, but it does not provide window management or multiplexing of mouse and keyboard input.

That is done by a separate window manager, `rio`, which is also a file server. It works in a similar way to the older `8½`,[4] but more efficiently because the separate draw device multiplexes graphics output directly. Each window is given its own name space, in which `rio` serves the names `cons` and `mouse` (amongst many others). When a textual application such as the shell opens `/dev/cons`, it will open the one for that window, served by `rio`. When a graphical application opens `/dev/mouse`, it will open the one for that window. When it starts up, `rio` opens `/dev/cons` and `/dev/mouse` in the name space, and follows rules such as 'button 1 selects a window' to decide which client window should receive the data it reads from those files on the client window's `/dev/cons` and `/dev/mouse`. This recursive structure allows an instance of `rio` to run inside a `rio` window.

`Acme`[5] is an unusual application that centralises user interaction for text- orientedapplications. Its window on the screen is tiled with many text windows, labelled with a file or directory name. Each of the three mouse buttons selects text, but does something different with it. Briefly, button 1 selects text for editing, button 2 executes it as a command (perhaps built- into Acme), and button 3 takes the selected text something to locate or retrieve (eg, a file or directory, or text in a file). `Acme` serves a single name space containing a few top- levelfiles and directories, and a numbered directory for each such window:

```
/mnt/acme/
        acme/
        cons
        consctl
        draw
        editout
        index
        label
        new
        1/
                addr
                body
                ctl
                data
                editout
                errors
                event
                rdsel
                wrsel
                tag
                xdata
        2/
                ...
        ...
```

Its name space is quite different from ,`rio`'s even though both are window managers, because the level of interaction is different. `Acme` supports only text windows, and its clients interact with the user through high- leveloperations through Acme files corresponding to the client's text windows. For instance, a regular expression can be evaluated on the text in a given window by writing it as text to the corresponding `addr` file; a read returns a textual description of the start and end characters of the next match. Reading the `data` file retrieves the matching text; writing to `data` replaces it with the written text. The `event` file, however, is the key one for most clients. Changes to the text in the window (or its label), for instance by the user's editing actions, are reported as messages on the `event` file. Button 1 editing operations are handled as normal by `acme` and simply reported. Operations by button 2 (execute) and button 3 (search/retrieve) produce messages giving the selected text and other parameters, leaving it up to the application to implement. For example, the mail reader produces a list of mail messages in one window, and when one of those is selected by button 3, it receives a message on `event`, retrieves the selected mail message from `upas/fs`, and puts its text in a new Acme window. That window has some mail- specificcommands added to its label: `Reply`, `Delmesg` and `Save`. When the user clicks one of those with button 2, the client receives a suitable message on `event` and can take appropriate action, such as popping up a new text window to receive the reply text. (A client can also write back an `event` message to `acme` to have it take its default action.) `Acme`'s clients include general text editing, debugging, mail reading, Usenet news, and others all operating through the files in the name space above.

Example: storage formats

File servers are also used in a more conventional way to make the contents of various archive and file formats (eg, tar, zip, ISO9660, DOS) available through the name space. Less conventionally, the interface to the FTP protocol is also a file server, `ftpfs`, that makes the remote FTP archive visible in the local name space directly as files and directories, and subject to existing shell commands and system calls, rather than accessible only through a special command language.

The primary permanent (ie, conventional, disc- based) file storage is provided through a file server `fossil`.[7] It is an ordinary application program, not built in to the kernel. It has two unusual properties: it uses an underlying specialised block- oriented archive service, `venti`,[6] to store its data and, taking advantage of that, and an internal copy- on- writestorage structure,

efficiently maintains snapshots of its whole file hierarchy, made automatically at nominated times, where each snapshot is directly accessible through the name space. This provides automated backup and a simple way to see changes over time:

```
term% diff /n/dump/2005/0201/sys/src/cmd/p.c /sys/src/cmd
36c36
<                               fprint(2, "p: can't open %s0, argv[0]);
---
>                               fprint(2, "p: can't open %s - %r0, argv[0]);
```

As well as using `diff` and `grep`, and of course `cp` to fetch things from the backup, it is possible to `bind` whole sections of the past into the present name space, for instance to do regression checks on the compilers and libraries, or to compare present and past performance. The underlying Venti archives can be copied to guard against disaster.

3.1. Discussion

File servers in can act as the interface to all manner of resources, as suggested briefly by the examples above. In Plan 9, file servers can rely on the following properties:

- the server sees all operations by each client in its space
- clients can be given per- client views of the underlying service or data
- many clients can safely share the same data through the file server
- a single server can provide access to many independent data files for many clients
- complex parsing and concurrency control can be factored out of applications into the server, making both server and application easier to write
- as `upas/fs` and `keyfs` show, the actual format of underlying data files is hidden from applications
- files have ownership and permission, enforced as desired by the file server itself
- servers can support indexing, queries, replication and transaction control

Furthermore, the services presented through the name space can be securely exported to the network using standard mechanisms in the operating system, transparently to both application *and* server. The protocol itself is straightforward (except for `flush`), and there is a `9p` library that helps make file servers easier to write. Most of the work in `upas/fs` is handling RFC822 messages and MIME, not presenting the file system view.

The exposing of data interfaces through the name space has a further advantage. Since the hierarchy is in the normal name space, ordinary system commands can operate on it, including `ls`, `cmp`, and `cp`. This is helpful during implementation and debugging. At Vita Nuova, when we wish to develop client and server sides concurrently, we simply agree a common name space and suitable messages on the files therein. The server is often tested using shell commands or shell scripts; the client is tested by using a mixture of static files and a rapid prototype of the server's name space (eg, allowing its interaction to be scripted).

The biggest benefit, of course, was mentioned earlier: all the resources described thus far, all the devices and interfaces provided as file servers by the kernel, and other file serving applications not mentioned, can have their resources distributed on the network, using an appropriate combination of `mount` and `exportfs`. The `import` command parcels up one such instance:

```
import host remotefs [ mountpt ]
```

It dials `exportfs` running as a service on *host*, does mutual authentication, requests the service to apply `exportfs` to the name *remotefs* in the service's name space, then mounts the resulting network connection on *mountpt* locally. Thus, to turn use another machine's network interfaces:

```
import -b mygateway /net
```

The `-b` option puts the remote's `/net` in a local union mount, before the local machine's own

interfaces. Thus, a request to use a given protocol will use the remote's if it has it, and the local one otherwise. Similarly, one can import services from other machines (eg, by importing `/net/cs` or `/net/dns`).

Remote graphics is similar. Plan 9's `cpu` command is typically used to connect a window on a terminal to a cpu server. It dials a service on the server, authenticates, then uses `exportfs` to export the terminal's name space to the cpu server, which mounts it on `/mnt/term`. It then binds `/mnt/term/dev` over `/dev` so that the terminal's devices will be used, not the cpu server's own, and starts a shell. The shell opens `/dev/cons`, but that's bound from `/mnt/term/dev/cons`, so input and output appear in the terminal window. More important, running a graphics program will open a connection to `/dev/draw` and `/dev/mouse` which will go over the mounted connection via `exportfs` into the terminal's exported name space, thus doing the graphics in the terminal window. In particular, running `rio` in the window will start a window system within that window but the windows created there will have its programs running on the cpu server. Obviously the same extends to remote audio, other remote devices, `factotum`, mail boxes, grid scheduling, etc.

4. The protocol

The name space is implemented using a single protocol, *9P*. It is defined by a few pages in section 5 of the Plan 9 Programmer's Manual, Volume 1,[22] but here is a brief summary. The protocol provides 13 operations, shown in Table 1.

`Tversion tag msize version`	start a new session
`Rversion tag msize version`	
`Tauth tag afid uname aname`	optionally authenticate subsequent attaches
`Rauth tag aqid`	
`Tattach tag fid afid uname aname`	attach to the root of a file tree
`Rattach tag qid`	
`Twalk tag fid newfid nwname nwname*wname`	walk up or down in the file tree
`Rwalk tag nwqid nwqid*wqid`	
`Topen tag fid mode`	open a file (directory) checking permissions
`Ropen tag qid iounit`	
`Tcreate tag fid name perm mode`	create a new file
`Rcreate tag qid iounit`	
`Tread tag fid offset count`	read data from an open file
`Rread tag count data`	
`Twrite tag fid offset count data`	write data to an open file
`Rwrite tag count`	
`Tclunk tag fid`	discard a file tree reference (ie, close)
`Rclunk tag`	
`Tremove tag fid`	remove a file
`Rremove tag`	
`Tstat tag fid`	retrieve a file's attributes
`Rstat tag stat`	
`Twstat tag fid stat`	set a file's attributes
`Rwstat tag`	
`Tflush tag oldtag`	flush pending requests (eg, on interrupt)
`Rflush tag`	
`Rerror tag ename`	error reply with diagnostic

Table 1. 9P messages

T- messagesare the requests sent from client to server, with a *tag* unique amongst all outstanding messages. The R- messagesare sent in reply, with the *tag* from the original T- message(that tag value can then be reused). A *fid* is a number chosen by the client to identify an active file (directory) on the server. The association is made by `Tauth`, `Tattach`, or `Twalk`, and lasts until it is

cancelled by `Tclunk` or `Tremove`. The *fid* in `Tattach` is associated with the server's root; the *newfid* in `Twalk` is associated with the result of walking from directory *fid* using each *wname* in turn. The other requests use a *fid* to identify the file or directory to which they apply.

The server can respond to requests out of order, and might defer a reply until some other event occurs (eg, a read of `/dev/cons` waits until the user types something). The client can cancel any outstanding request using `Tflush`, giving the *tag* of the cancelled request. The protocol is defined to handle correctly cases such as a flush arriving after reply, or a *tag* being flushed more than once.

A server's files are uniquely identified by a 13- byte*qid*, which it determines (it has some substructure and attributes not discussed here). Two files are identical if they have the same *qid*. (Note that unlike Unix i- node numbers, one cannot access afile using only a *qid*.) The *qid* can be used to detect stale data when caching data or files.

Outside the kernel, over pipes and on the network, the protocol has a well- definedand compact representation as a stream of bytes. The protocol is not an Internet protocol, but does need reliable, in- sequencedelivery. Each message is preceded by a 4 byte message length to delimit it in the stream, the message type is one byte, tags are two bytes, fids are four bytes, counts are four bytes, qids are 13 bytes, and strings and stat data are preceded by a two- bytecount. The code to marshal and unmarshal the structures is simple, obvious and small.

An incoming connection is optionally authenticated, with encryption and digesting engaged if desired. That operation lies outside 9P proper. Within the protocol, the server can require that each `Tattach` be authenticated. If so, the client does a `Tauth` which creates an *authentication file* on the server, referred to by *afid*, which can then be read and written using `Tread` and `Twrite` to exchange the data in any agreed authentication protocol. If successful, the *afid* can then be presented in a subsequent `Tattach` to authorise it. Both connection level and `Tauth` authentication is typically done using `factotum`.

5. Implementation environment

We saw above that Plan 9 has 'file servers' at the heart of its design and implementation. They rely, however, on aspects of the system that are available to all applications. The remaining sections look briefly at the programming environment, the support for concurrency, and the Plan 9 kernel.

Apart from a few venerable utilities derived from Research Unix, all the code is new and unrelated to the commercial Unix code; even the older programs have usually been changed, for instance to support Unicode, Plan 9's native character set. In particular, the Plan 9 kernel code, including the networking code, is completely new. It was designed with symmetric multiprocessor support from the start. Even so, the kernels have seen several significant revisions, mainly in the networking code. For example, the first two editions provided a Stream I/O subsystem similar to that of 8th Edition Unix (and much simpler than System V's), which was used in various device drivers, including the protocol stack. The Third Edition replaced that by a simpler, more efficient abstraction for queued I/O. The networking subsystem also became more modular.

The system is written to be portable. Assumptions about architectural details, such as the structure of the memory- management unit, are confined to a small amount of machine- dependent code. Indeed, unlike some 'portable' systems, the aim is to map the software requirements as efficiently as possible into the real processor, not to reflect the processor's nature into the higher-level software. It is also trivial to compile for any architecture on any others, and debug one platform from any other.

The kernel and nearly all applications are implemented in Plan 9's dialect of C. It is not POSIX compatible, because the libraries and programming conventions were done anew for Plan 9. For instance, the structure of libraries and header files is much more straightforward. There are new libraries: `libbio` for buffered IO; `lib9p` for file server implementation; `libsec` and `libmp` for encryption; `libauth` for authentication; `libplumb` to interact with the message exchange; `libdraw` for bitmap graphics; `libthread` for concurrent programming (see below); and others.

Plan 9 uses its own C compiler suite, written by Ken Thompson, which itself has unusual structure. See Thompson's paper for details[11] There is good support for development for a system with heterogeneous architectures (both compilation and debugging). In general, the source code to an application is invariant across architectures. There is no use of `#ifdef` or `./configure` to parametrise the source code for any particular target architecture. Application code (and most kernel code) is simply written not to depend on architectural details. Programs such as the debuggers that need to manipulate executables and other machine- dependent data, do so through the library `libmach`. Remote debugging is done by importing the `/proc` file system from the remote machine into the debugger's name space. They can of course be different architectures.

A bigger difference is the good support at both library and kernel level for concurrent programming. That support is essential and well- used:many applications are concurrent programs. File servers are often concurrent programs to allow them to handle many clients at once, particularly when some requests might block. For example, `ndb/cs` creates a new process for each request that requires use of DNS or similar services. DNS also creates a process for each request than cannot be answered immediately from its cache. The window system `rio`, the text window system `acme`, `exportfs` and the `plumber` are all concurrent programs.

Two system calls, `rfork` and `rendezvous`, provide all the necessary kernel support. `Rfork` takes a bit mask that controls the sharing (or not) of resources of the calling process with its parent and children:

```
pid = rfork(RFPROC|RFMEM|RFFDG);
```

Resources include file descriptors, name space, environment variables, rendezvous group (see below), and memory. `Rfork` can optionally create a new child process. The parent also chooses which of its other resources will be shared with that child. In particular, a child can optionally share the parent's memory segments. It always has a private copy of the stack to that point, avoiding the need for assembly- language linkage,and also providing some process- privatememory.

`Rendezvous` allows two processes that share the same rendezvous group to synchronise:

```
void* rendezvous(ulong tag, void *value);
```

One process calls `rendezvous` with an agreed *tag* value; it blocks until another process does the same. At that point, the *values* are exchanged between the processes, and each call returns the other process's *value*. The meaning of the *value* is up to the programmer, although it is often a pointer to a shared data structure. All other concurrent programming operations are built on `rendezvous`.

Two styles of concurrent programming have library support. The Plan 9 C library provides spin locks, queued locks, and reader/writer locks, for primitive shared memory programming. Most concurrent applications, however, use a newer library that allows processes in a shared address space (as produced by `rfork`) to communicate and synchronise by sending and receiving values on typed *channels*, in the style of Hoare's Communicating Sequential Processes (as subsequently implemented by `occam` and a few other languages). The basic operations are *send* on a channel, *receive* from a channel, and *alt* to send or receive to one of a number of channels (the first that is ready, allowing a process to interact with several senders on different channels). The library provides both synchronous (unbuffered) and buffered channels. Plan 9 itself has only one kind of process (the aim is to keep that sufficiently lightweight). The library adds support for *threads*, but they are coroutines, not processes, and scheduled non- preemptively(within a given Plan 9 process, which can itself be preempted). Threads and processes can however interact through channels. The debugger `acid` has a library in the `acid` language that supports debugging concurrent applications. Most new non- trivial concurrent programs use the channel- based library.

6. Kernel implementation

The kernel has an invariant core that includes the implementation of kernel and user processes, memory allocation, virtual memory, queued I/O support, and name space primitives, in about 36 source files (including some optional modules), A further 27 files provide platform- independent kernel file servers, including for essential functionality.

The kernel supports symmetric multiprocessing, using kernel processes (which can be pre-empted) cooperating through spin locks, basic queued locks, queued locks providing multiple readers and exclusive writing, and multiprocessor sleep and wakeup. Processes are assigned dynamically to one of the available processors, taking account of their previous affinity, to reduce cache and TLB flushing on some hardware. Processes are preemptable, and scheduled by priority. There are 20 priority levels, split into bands (user, kernel and root kernel processes), with dynamic priority adjustment in the user process band by default, though that can be adjusted through a control file. Real- time support is provided by a variant of EDF (Earliest Deadline First), using deadline inheritance to handle shared resources.[15] It is enabled and controlled for a process or process group using the `/proc` file.

Device driver routines are invoked by a process context (either a kernel process or the kernel context of a user- modeprocess in a system call). Thus a single device driver serves numerous client processes. Interrupts are invoked outside process context (either on the current process stack or on a special stack). It is up to the machine- dependentcode how and when rescheduling takes place on an interrupt.

Within the kernel, the 9P protocol is implemented directly by function calls; that constitutes the only interface between the kernel and its devices and protocols. The functions in the current interface are shown below:

```
struct Dev
{
        int     dc;
        char*   name;

        void    (*reset)(void);
        void    (*init)(void);
        void    (*shutdown)(void);
        Chan*   (*attach)(char*);
        Walkqid*(*walk)(Chan*, Chan*, char**, int);
        int     (*stat)(Chan*, uchar*, int);
        Chan*   (*open)(Chan*, int);
        void    (*create)(Chan*, char*, int, ulong);
        void    (*close)(Chan*);
        long    (*read)(Chan*, void*, long, vlong);
        Block*  (*bread)(Chan*, long, ulong);
        long    (*write)(Chan*, void*, long, vlong);
        long    (*bwrite)(Chan*, Block*, ulong);
        void    (*remove)(Chan*);
        int     (*wstat)(Chan*, uchar*, int);
        void    (*power)(int);  /* power mgt: power(1) => on, power (0) => off */
        int     (*config)(int, char*, DevConf*);
};
```

A `Chan` corresponds to a *fid* in the protocol. The separate structure `Walkqid` represents a variable- lengtharray of `Qid` values. There are a few extra functions to allow the file server to initialise its own state and that of a device it is driving; for power management and dynamic configuration; and `bread` and `bwrite` to read and write data directly in the `Block` representation used by the network subsystem and other device drivers. A kernel library helps implement name spaces, parse control messages, and manage queues of data.

Some kernel file systems have their own interfaces below them. For example, a single `devether.c` serves the standard name space expected of the `ether` medium, and implements

its control messages. Each physical implementation has its own device- specificcode, which the file server invokes through the following interface:

```
typedef struct Ether Ether;
struct Ether {
        ...

        void    (*attach)(Ether*);      /* filled in by reset routine */
        void    (*detach)(Ether*);
        void    (*transmit)(Ether*);
        void    (*interrupt)(Ureg*, void*);
        long    (*ifstat)(Ether*, void*, long, ulong);
        long    (*ctl)(Ether*, void*, long); /* custom ctl messages */
        void    (*power)(Ether*, int);  /* power on/off */
        void    (*shutdown)(Ether*);    /* shutdown hardware before reboot */
        void    *ctlr;

        Queue*  oq;

        ...
};
```

It separates the card- specificcode from the file- servingcode. `Devether.c` itself uses a library of functions that helps implement the bulk of a generic 'network interface' name space, leaving it to deal with the Ether- specificwork. A similar approach is taken with some other device drivers, such as those for UARTs and USB.

7. The distribution

The Plan 9 distribution includes all the current compiler suites, and the source code for the kernel and applications. It is self- supportingand does not require any other software to run, compile or develop it.

The Plan 9 compiler suite[11] currently supports 680x0, various MIPS, ARM/Thumb, Alpha, x86, AMD64, SPARC, and PowerPC. (Other implementations have existed but are now obsolete.)

The free distribution includes the following kernels (in `/sys/src/9`):

`alphapc`	Alpha PC 164
`bitsy`	iPAQ 36xx
`mtx`	PowerPC 603
`pc`	Intel/AMD/Other x86
`ppc`	PowerPC 8260 or 750/755

The kernels other than `pc` are included as samples of the system running on non- Intelarchitecture; they usually target specific hardware.

8. Resources

> ... but as he climbed in over the rail, so a waft of air took the frigate's sails back, a breath of heavy air off the land, with a thousand unknown scents, the green smell of damp vegetation, palms, close- packedhumanity, another world.
>
> — Patrick O'Brian, *HMS Surprise*

The whole system can be downloaded free of charge in PC- bootable ISO9660 format; see `plan9.bell-labs.com/plan9`. The site also offers online updates to Plan 9 systems once installed, using an authenticated 9P connection.

The documents and manual pages are included in machine- readableform; they are also available in printed form, ordered from `www.vitanuova.com/plan9`.

References

1. Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey Phil Winterbottom, ''Plan 9 from Bell Labs,'' *Computing Systems* **8**(3), pp. 221- 254(Summer 1995).
2. Dave Presotto Phil Winterbottom, ''The Organization of Networks in Plan 9,'' *Proceedings of the Winter 1993 USENIX Conference*, San Diego, California, pp. 271- 280 (1993).
3. Dave Presotto Phil Winterbottom, ''The IL Protocol,'' *Plan 9 Programmers Manual, Fourth Edition* **2**, Bell Labs Lucent Technologies (2002).
4. Rob Pike, ''8½, the Plan 9 Window System,'' *Proceedings of the Summer 1991 USENIX Conference*, Nashville, pp. 257- 265 (1991).
5. Rob Pike, ''Acme: A User Interface for Programmers,'' *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, California, pp. 223- 234 (1994).
6. Sean Quinlan Sean Dorward, ''Venti: a new approach to archival storage,'' *First USENIX Conference on File and Storage Technologies*, Monterey, California (2002).
7. Sean Quinlan, Jim McKie Russ Cox, *Fossil, an Archival File Server,* Lucent Technologies Bell Labs, Unpublished memorandum (September 2003).
8. Sean Quinlan, ''A Cached WORM File System,'' *Software: Practice and Experience* **21**(12), pp. 1289- 1299 (1991).
9. Ken Thompson, ''The Plan 9 File Server,'' *Plan 9 Programmers Manual, Fourth Edition* **2**, Bell Labs Lucent Technologies (2002).
10. Rob Pike, ''Plumbing and Other Utilities,'' *Proceedings of the USENIX Annual Conference*, San Diego, California, pp. 159- 170 (June 2000).
11. Ken Thompson, ''Plan 9 C Compilers,'' *Proceedings of the Summer 1990 UKUUG Conference*, London, pp. 41- 51 (1990).
12. Phil Winterbottom, ''Acid: A Debugger Built From A Language,'' *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, California, pp. 211- 222 (1994).
13. Rob Pike, Dave Presotto, Ken Thompson Gerard Holzmann, ''Process Sleep and Wakeup on a Shared- memory Multiprocessor,'' *Proceedings of the Spring 1991 EurOpen Conference*, Tromsø, Norway, pp. 161- 166 (1991).
14. Rob Pike, ''Lexical File Names in Plan 9, or Getting Dot- Dot Right,'' *Proceedings of the USENIX Annual Conference*, San Diego, California, pp. 85- 92 (June 2000).
15. Pierre Jansen, Sape Mullender, Paul J M Havinga Hans Scholten, *Lightweight EDF Scheduling with Deadline Inheritance,* University of Twente (2003).
16. Russ Cox, Eric Grosse, Rob Pike, Dave Presotto Sean Quinlan, ''Security in Plan 9,'' *Proceedings of the 11th USENIX Security Symposium*, San Francisco, pp. 3- 16 (2002).
17. Rob Pike Ken Thompson, ''Hello World, or Καλημέρα κόσμε, ...,'' *Proceedings of the Winter 1993 USENIX Conference*, San Diego, pp. 43- 50 (1993).
18. Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey Phil Winterbottom, ''The Use of Name Spaces in Plan 9,'' *Proceedings of the 5th ACM SIGOPS European Workshop*, Mont Saint- Michel (1992).
19. R Sandberg, D Goldberg, S Kleimann, D Walsh B Lyon, ''Design and Implementation of the Sun Network File System,'' *Proceedings of the Summer 1985 USENIX Conference*, Portland, Oregon, pp. 119- 130 (June 1985).
20. Rob Pike Dennis M Ritchie, ''The Styx Architecture for Distributed Systems,'' *Bell Labs Technical Journal* **4**(2), pp. 146- 152 (April- June 1999).
21. Rob Pike, ''Graphics in Overlapping Bitmap Layers,'' *Transactions on Graphics* **2**(2), pp. 135- 160 (1982).

22. *Plan 9 Programmers Manual, Fourth Edition (Manual Pages),* Bell Labs Lucent Technologies (2002).