The Organization of Networks in Plan 9

Dave Presotto
Phil Winterbottom

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
USA

ABSTRACT

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

1. Introduction

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

A Plan 9 system is comprised of file servers, CPU servers and terminals. The File servers and CPU servers are typically multiprocessor machines with large memories in a central location. A variety of workstation class machines are used as terminals and are connected to the central servers using a variety of networks and protocols. The architecture of the system demands a hierarchy of network speeds matching the needs of the components. Connections between File servers and CPU servers are high bandwidth point to point fiber links. Connections from the central servers fan out to medium speed networks in the building at Murray Hill over Ethernet [Met80] and Datakit [Fra80]. Low speed connections via the Internet and the AT&T backbone serve users in Oregon and Illinois. Finally Basic Rate ISDN data service and 9.6K baud serial lines provide very slow links to users at home.

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. By hiding the system connectivity from the users Plan 9 provides this flexibility without constraining the choice. Simple commands allow a locally represented namespace to transparently span many machines and networks. 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 endgames. At home or connected over a slow network users tend to do most of the work on the CPU server to minimize network traffic. The goal of the network organization is to provide the same environment to the user wherever resources are used.

2. Kernel Network Support

The importance of the networking code within the kernel is reflected by its size. Of 25,000 lines of kernel code 12,500 are network and protocol related. Networks are being added all the time, so the fraction of code devoted to communications is growing. Networks play a central role in any distributed system; this is particularly true in Plan 9 where most resources are provided by servers external to the kernel. Moreover the network code is complex. Protocol implementations consist almost entirely of synchronization and

dynamic memory management and are an area where good error recovery is essential. The networks currently supported by the kernel are Datakit, point to point fiber links, an Internet (IP) protocol suite and ISDN data service. The diversity of networks and variety of machines which run Plan 9 has raised issues not addressed by other systems running on commercial hardware supporting only Ethernet or FDDI.

2.1. The File System protocol

A central idea in Plan 9 is the representation of a resource as a hierarchical file system. Each process assembles a view of the system by building a *namespace* [Needham] that connects those resources. File systems need not represent disc files; in fact most Plan 9 file systems have no permanent storage. A typical file system builds a dynamic representation of some resource, like a set of network connections or the process table. Communication between the kernel, device drivers and local or remote file servers uses a protocol called 9P. The protocol consists of 17 messages which describe operations on files and directories. Device and protocol drivers built into the kernel use a procedural version of the protocol while external file servers use an RPC form of the protocol. Nearly all traffic between Plan 9 systems consists of 9P messages. 9P makes several assumptions about the underlying transport protocol. The protocol assumes messages arrive reliably and in sequence. Delimiters between messages must be preserved. Where a protocol is unable to meet these requirements (for example TCP does not preserve delimiters) a means must be provided to marshal messages before they are handed to the system.

A kernel data structure, the *channel*, is a handle to a file server object. 9P messages are generated for each operation on a channel. The following is a brief description of the messages. 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, much like a kernel version of the dup system call. A *channel* may be moved to a file on the server using a `walk` message to descend each level in the hierarchy. The `stat` and `wstat` messages read and write the attributes of the file 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.

A kernel resident file server called the *mount* driver converts the procedural version of 9P into RPC's. The file descriptor is converted into a channel which can be a pipe to a user process or a network connection to a remote machine. After a mount, operations on the file tree below the mount point are sent as messages to the file server. The *mount* driver performs the associated buffer management, packs and unpacks parameters from messages and demultiplexes between processes using the file server.

2.2. Cyclone Fiber Links

The central file servers and CPU servers are connected together by point to point links. A link consists of two VME cards connected by a pair of optical fibers. The links allow high bandwidth communication between machines with VME buses. The VME cards use 33Mhz Intel 960 processors and AMD's TAXI fiber transmitter/receivers. The line speed is 125 Mbit/sec. Software in the VME card allows messages to be copied from system memory to fiber without intermediate buffering, so reducing latency.

2.3. Kernel Organization

The network code in the kernel can be divided into three layers; hardware interface, protocol processing, and program interface. A device driver typically provides the two interface layers using streams to provide the plumbing between the interfaces. Streams modules pushed onto the devices process the protocols. Each device driver is a kernel resident file system. Simple device drivers serve a single level file structure containing just a few files. For example the UART's on our processors are each represented by a data and a control file.

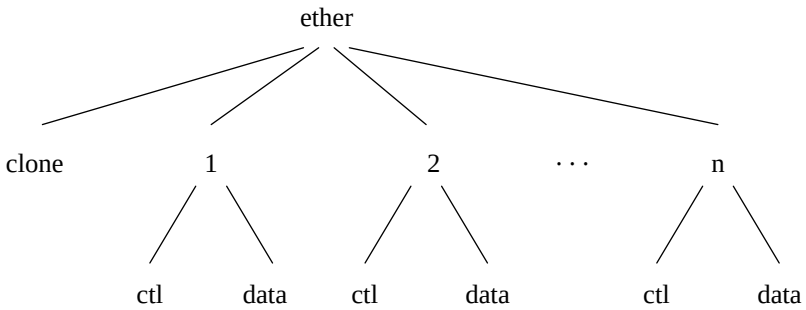
```
helix% cd /dev
helix% ls -l eia*
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia1
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia1ctl
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia2
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia2ctl
helix%
```

The control file is used to control the device. For example, writing *b1200* to `/dev/eia1ctl` will set that line to 1200 baud.

Multiplexed devices like the Ethernet interfaces present a more structured interface. The LANCE Ethernet driver provides an typical example. This driver is used to

- provide device control and configuration
- implement protocols like *arp* at user level.
- diagnose problems using snooping software.

The driver serves a two level file tree (Figure 1). The top directory contains a `clone` file and a directory for each conversation, numbered `1` to `n`. Each conversation directory corresponds to an Ethernet packet type. Opening the clone file finds an unused conversation directory and opens its `ctl` file. Reading the control file returns the ASCII number of the conversation chosen. This allows the user process to find the corresponding directory. Each connection directory contains files which represent the state of the connection. The LANCE driver supplies the files `ctl, data, stats` and `type.` Writing the string *connect 2048* to the `ctl` file sets the packet type to 2048. This configures a connection to receive all IP packets sent to the machine. Subsequent reads of the file `type` yields the string 2048. The `data` file accesses the media. A read from the `data` file will return the next packet of the type selected by the connection. A write to the file will append a packet header containing the source address and packet type to the data and then queue the packet for transmission on the wire. The `stats` file returns ASCII text containing the address of the interface, packet input/output counts, error statistics and general information about the state of the interface.



If more than one connection on an interface is configured for a particular type then each will receive a copy of the incoming packets. The special packet type, `-1`, is a wildcard matching any packet type. Writing the strings *promiscuous* and *connect -1* to `ctl` will cause that conversation to receive all packets on the Ethernet.

The driver interface may seem broader and more elaborate than necessary. However the representation of the device as a set of files using ASCII strings to communicate with the driver has several advantages. If we have a mechanism which allows files to be accessed remotely we immediately have the means to allow a remote machine to use our interfaces as gateways. By using ASCII strings to control the interface we avoid byte order problems and have a common representation which is easy to keep consistent between devices on the same machine and even devices accessed remotely. By representing dissimilar devices by the same set of files, common tools can be used on more than one network or interface. Programs like `stty` are eliminated and are replaced by `echo` and shell redirection.

2.4. Protocol devices

Network connections are represented as pseudo-devices called protocol devices. Protocol device drivers exist for the Datakit URP protocol and for each of the Internet IP protocols TCP, UDP, and IL. IL, described below, is a new communication protocol used by Plan 9 for transmitting file system RPC. All protocol devices look exactly the same so that programs need not include network specific code.

Each protocol device driver serves the same directory structure as the Ethernet driver. The top directory contains a `clone` file and a directory for each conversation numbered `1` to `n`. Each directory contains files needed to control one conversation and to send and receive information. A look at a tcp conversation provides the following:

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

The files `local, remote` and `status` are informational and represent the state of the connection. The `data` and `ctl` files are supplied by the streams code and provide the interface to the process end of the stream implementing the protocol. The `listen` file provides means to accept incoming calls from the network.

The following steps are used to establish a connection.

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

A process can then read and write the file descriptor to the `data` file to send and receive messages from the network. If the process opens the `listen` file the process will block until an incoming call is received. An address string written to the control file before the listen selects the ports or services the server is prepared to accept. After an incoming call is received the open completes. The file descriptor returned from the open of a `listen` file will point to the control file of the new connection. The path of the `data` file can be obtained by reading the control file. A connection remains established while any of the files in the connection directory are referenced or until a close is received from the network.

2.5. Streams

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

A stream is made up of a linear list of *processing modules*. Each module has both an upstream (toward the process) and downstream (toward the device) *put routine*. Data is inserted into the stream by calling the put routine of the module at either end of the stream. Each module calls the succeeding one to send data up or down the stream.

An instance of a processing module is represented by a pair of *Queue* structures, one for each direction. The queues point to the put procedures and can be used to queue information traveling along the stream.

Some put routines queue data locally and send it along the stream at some later time either due to a subsequent call or an asynchronous event such as a retransmission timer or a device interrupt. Processing modules create helper kernel processes to provide a context to handle these asynchronous events. For example, a helper kernel process exists that awakens periodically to perform any necessary TCP retransmissions. The use of kernel processes instead of serialized run-to-completion service routines is a major deviation from Unix streams. The two major problems with Unix service routines are their inability to use any blocking kernel resource and their lack of local long lived state. Using kernel processes solves these problems and makes the stream code considerably less complex.

There is no implicit synchronization in our streams. Each processing module must ensure that concurrent processes using the stream are synchronized. While this allows a maximum of concurrency it creates the possibility of deadlock that did not exist in Unix streams. However deadlocking has not caused us problems.

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

```
typedef struct Block    Block;
struct Block
{
        Block   *next;
        uchar   *rptr;          /* first unconsumed byte in buffer */
        uchar   *wptr;          /* first empty byte in buffer */
        uchar   *base;          /* start of the buffer */
        uchar   *lim;           /* 1 past the end of the buffer */
        uchar   type;           /* M_DATA, M_CTL, M_HANGUP */
        uchar   flags;
};
```

2.5.1. User Interface

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

Like UNIX streams [bltj Plan 9 streams can be dynamically configured. The stream system intercepts the control blocks which control configuration of the stream. These control blocks are:

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

pop to remove the top module of the stream.

hangup to send a hangup message up the stream from the device end.

Other control blocks are stream specific and interpreted by each module as they pass through.

Our system has no `ioctl` system call. The syntax and semantics of `ioctl` in UNIX are so uncontrolled that we left it out. Writing to the `ctl` file takes the place of `ioctl`. Writing to the control file is the same as writing to a data file except that the blocks created are of type control. A processing module

parses each control block put to it. The commands in the control blocks are simple ASCII strings. Therefore, there is no problem with byte ordering when one system is controlling streams in a name space implemented on another processor. The time to parse control blocks is not important since control operations are rare.

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

2.5.2. Device Interface

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

2.5.3. Multiplexing

The conversations belonging to a protocol device need to be multiplexed onto a single physical device. We group the conversations by pushing a multiplexing processing module onto the physical device stream. The device end modules on the conversations add the necessary header onto downstream messages and then put them to the module downstream of the multiplexor. The multiplexing module looks at each message moving up its stream and puts it to the correct conversation stream after stripping whatever header it used to do the demultiplexing.

This is similar to how Unix streams implement multiplexors. The major difference is that we have no general structure that corresponds to a multiplexor. Each attempt to introduce one made the code more complicated. Therefore, each multiplexor is written from scratch.

2.5.4. Reflections

After 5 years' experience with streams and many programmers writing stream code, we are still dissatisfied with the mechanism. Performance is not an issue. The time to process protocols and drive device interfaces continues to dwarf the time for the stream mechanism itself, i.e., allocating, freeing, and moving blocks of data. The problem is that the mechanism remains one of the most complex in our kernel. Much of the complexity exists to make streams dynamically configurable and to make processing modules reusable on different devices. This includes synchronization in the kernel to ensure that data structures don't disappear under foot. This is irritating since we seldom use these properties. The same was true on our research Unix systems. Only the tty module was ever heavily reused and Plan 9 needs no such module.

Streams remain in our kernel because we are unable to devise a better alternative. Larry Peterson's X-kernel [Pet89a] is the closest contender but doesn't offer enough advantage to switch.

3. The IL Protocol

None of the standard IP protocols are suitable for transmission of 9P messages over an Ethernet or the Internet. TCP has a high overhead and does not preserve delimiters. UDP while cheap does not provide reliable sequenced delivery. Early versions of the system used a protocol called `Nonet` on local Ethernets. The protocol was efficient but had some problems. Packets were addressed by Ethernet source and destination addresses. Most routing hardware does not preserve Ethernet addresses. We were unable to supply a service across the Internet or AT&T backbone because the protocol was not encapsulated by IP. Finally, implementations of the protocol were unreliable because the protocol was ill defined. When IP, TCP and UDP were implemented we decided to look around for a suitable replacement. None met our needs:

- Reliable datagram service with sequenced delivery
- Run over IP
- Low complexity, high performance
- Adaptive timeouts to allow good performance over the Internet

IL is a lightweight protocol designed to be encapsulated by IP. It provides reliable transmission of sequenced messages between machines. The protocol is connection based. No provision is made for flow control since the protocol was designed to transport RPC messages between client and server. A small outstanding message window prevents too many incoming messages from being buffered. Messages outside of the window are discarded and must be made up by retransmissions. Connection setup uses a two way handshake to generate initial sequence numbers at each end of the connection. Each data message increments the sequence number. Messages out of order are resequenced by the receiver. In contrast to other protocols IL does not do blind retransmission. If a message is lost and a timeout occurs, a querey message is sent. The query message is a small control message containing the current sequence numbers as seen by the sender. The receiver responds to a query by retransmitting missing messages. This allows the protocol to behave well in congested networks, where blind retransmission would cause further congestion. Like TCP, IL has adaptive timeouts. A round-trip timer is used to calculate acknowledge and retransmission times in terms of the network speed. This allows the protocol to perform well on the Internet and on local Ethernets.

In keeping with the minimalist design of the rest of the kernel, IL is small. It has the minimum mechanism to fulfill our needs. The entire protocol is 847 lines of code, compared to 2200 lines for TCP. IL has become our protocol of choice.

4. Network Addressing

Defining a uniform interface to the protocols and devices is not sufficient to allow the transparency we require. Since each of the networks uses a different addressing scheme the ASCII strings written to a control file have no common format. That implies that each tool needs to know the specifics of the networks it is capable of addressing. Moreover since each machine supplies a different subset of the available networks each user would have to be aware of the networks supported by each terminal and server machine. This is obviously an unacceptable situation.

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

If the tools are to be network independent a third party server must be used to resolve network names. A server on each machine, with local knowledge, can select the best network for any particular machine or service. Since the network devices have a common interface the only operation which differs between networks is name resolution. A symbolic name must be translated into an ASCII string containing a path to the clone file of a protocol device and an address string which can be written to the `ctl` file. A connection server (CS) provides this service. Translation is performed by writing a symbolic name to */net/cs* and reading back a translated address. Symbolic names have the form *network!machine!service.* In most cases both the *network* and *service* can be omitted. A default network is selected by CS and the service, for instance

rlogin, is implied by the tool.

4.1. Network Database

Network information on most systems is held in a number of files such as `/etc/hosts`, `/etc/networks`, `/etc/services`, `/etc/hosts.equiv`, `/etc/bootptab`, `/etc/named.d`, etc. A major component of administering a computing environment is maintaining these files and keeping them mutually consistent. This has been aided by a number of tools that try to automatically derive one or more of the files from information in other files. Despite such aids maintenance continues to be difficult and error prone. The only real reason for this plethora of files is that each appeared as an adjunct to some new tool.

Since we wrote our world from scratch, we decided to avoid this nightmare. One database on a shared server contains all the information needed for network administration. The main database is composed of two ascii files, `/lib/ndb/local` containing information locally administered and `/lib/ndb/global` containing information imported from elsewhere. The files contain sets of attribute/value pairs of the form *attr=value*. An unindented line starts a new set. Both attribute and value can be any alphanumeric string. Systems are described by multi-line entries giving names, addresses, properties, etc. For example, the set for our main CPU server specifies a domain name, an IP address, an Ethernet address, a Datakit address, a boot file, and protocols supported.

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

If there are many systems sharing the same entries such as network mask and gateway, we specify that information with the network or subnetwork instead of the system. The following entries define a class B IP network and a few subnets derived from it. The entry for the network specifies the ipmask, file system, and authentication server to be used by all systems on the network. Each subnetwork specifies the default IP gateway for that subnetwork.

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

The mapping of service names to port numbers for TCP, UDP, and IL is also contained in the database.

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

All programs using the database read it directly. There are no intermediate files or binary format. Therefore, consistency problems are rare. However the database files can become quite large. Our global file, which contains all information about both Datakit and Internet systems in AT&T, is 43000 lines long. To speed up searches, we build hash table files for each attribute that we expect to search a lot. The hash files contain pointers to entries in the master files. Since hash files contain the modify time of the master file they pertain to, we never use out of date hash tables. Searches for attributes that aren’t hashed or whose hash table is out of date still work, they just take longer.

4.2. Connection Server

Each system runs a user level connection server process, CS, to perform symbolic name to address translation. CS uses information about available networks, the network database, and other servers (such as DNS) to translate names. As we stated earlier, CS is a file server that presents one file `/net/cs`. For any symbolic name written to `/net/cs` CS responds with one line for each matching destination that might be reachable from this system. The lines are of the form *filename message*, where *filename* is the clone file to open for a new connection and *message* is what to write into it to make the connection.

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

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

Like CS, the domain name server is a user level process providing one file, `/net/dns`. A client writes a request into the file of the form *domain-name type*, where *type* is a domain name service resource record type. DNS will then perform a recursive query through the Internet domain name system and return one line per resource record found. Like other domain name servers, DNS caches information learned from the network. DNS also has a UDP interface to answer requests from other systems. It runs as a multiprocess shared memory application.

5. Library routines

The details of making and receiving connections across a network were described in the section on protocol devices. While straightforward, the dance is complicated enough to be tedious. The routines described here are provided to relieve the programmer of most of the tedium.

5.1. Connecting

Making a connection to a remote destination uses a single library call, *dial*(2). *Dial* returns a file descriptor for the open data file of the connection.

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

dest is the symbolic name/address of the destination.

local is the local address. Since most networks do not allow this to be specified, it is almost always zero.

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

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

Most programs call *dial* with only a destination name and all the other arguments 0. Using CS, *dial* translates the symbolic name into all possible destination addresses. It then attempts connecting to each one in turn until one works. By using the special name `net` as a network, the user requests the system to pick a network/protocol in common with the destination host and for which the requested service is valid. For example, assume that the system *research.att.com* has the Datakit address *nj/astro/research* and the IP addresses *135.104.117.5* and *129.11.4.1*. The call

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

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

A programmer may also use addresses instead of symbolic names to dial out. For example, the

destinations *tcp!135.104.117.5!513* and *tcp!research.att.com!login* are both valid arguments to *dial* for the same machine. Since we use unicode representations everywhere, no commands or library routines need make a distinction between the use of addresses and names.

5.2. Listening

Four routines are used to listen for incoming connections. A program first *announce*(2)'s its intention to receive connections. It then *listen*(2)'s for calls and finally *accept*(2)'s or *reject*(2)'s them. *Announce* returns a file descriptor for the control file and the path name of the protocol directory (in *dir*) for the announcement.

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

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

Listen returns a file descriptor for the control file and the path name of the protocol directory (in *ldir*) for the received connection. It is passed *dir* from the announcement.

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

Accept and *reject* are called with the control file descriptor and *ldir* from the call to *listen.* Some networks such as Datakit allow the reason for the rejection. Networks such as IP ignore the argument.

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

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

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

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

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

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

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

        }
}
```

6. User Level

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

By convention the protocol and device driver file systems are mounted in a directory called */net*. Since the namespace is per process a user may configure any view of the system he chooses. However in practice most people's profiles build a similar namespace.

6.1. Exportfs

Exportfs is invoked by an incoming network call. The profile of the user requesting the service is run by the listener (Plan 9 equivalent of inetd). The profile constructs a namespace for the user. After the profile has executed the *exportfs* program is started. An initial protocol establishes the root of the file tree being exported. The connection to *exportfs* is then mounted by the remote process. *Exportfs* then acts as a

relay file server; operations in the imported file tree are executed on the remote server and the results returned. This gives the appearance of exporting a name space from a remote machine into a local file tree.

The *import* command calls *exportfs* on a remote machine and mounts the result in the local namespace. After the connection has been mounted the import exits. No process is required locally to serve mounts. 9P messages are generated by the mount driver in the kernel and sent directly over the network.

Exportfs must be multithreaded since the system calls *open, read* and *write* may block. Plan 9 does not implement the *select* system call but does have mechanisms for sharing file descriptor tables, memory and other resources between processes. This eases the burden of writing multithreaded servers. Together with the namespace *Exportfs* provides a powerful means of sharing resources between machines. It is used as a building block for complex namespaces served from many machines. The simplicity of the interfaces enables naive users to exploit the potential of a richly connected environment.

Using these tools it is possible to gateway between networks. For example, on a terminal connected only to the Datakit:

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

The *import* command makes a Datakit connection to the machine helix. An *exportfs* process is started to serve */net* from helix. The import command mounts the remote */net* directory after (the -a option to import) the existing local contents of */net*. The contents of the directory has become the union of the local and remote versions of */net*. Duplicate entries in the directories will be matched by the local entry. Networks supplied by the local machine will continue to be chosen in preference to those supplied remotely. Unique entries in the remote directory however are now visible in the local */net*. The networks helix supports over and above the Datakit are now available in the terminal.

7. Performance

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

Table 1 - Performance

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

8. Conclusion

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

9. References

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

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

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

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

[Fra80] A. G. Fraser, ‘‘Datakit - A Modular Network for Synchronous and Asynchronous Traffic’’, *Proc. International Conference on Communication , Boston Mass., June 1980.*

[Pet89a] L. Peterson, ‘‘RPC in the X-Kernel: Evaluating new Design Techniques’’, *Proc. Twelfth Symposium on Operating Systems Principles,* Lictfield Park AZ, December 1990.

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