6 — Network communication

6.1. Network connections

Plan 9 is a distributed system. But even if it was as its ancestor, UNIX, a centralized system that was designed just for one machine, it is very important to be able to use the network to provide services for other machines and to use services from others. All the operating systems that are in use today provide abstractions similar to the one whose interface is described here, to let you use the network.

This chapter may be hard to understand if you have not attended a computer networks course, but we try to do our best to explain how to use the network in any case. All the programs you have used to browse the Web, exchange electronic mail, etc. are implemented using interfaces that are similar to the ones described below (they use to be more complex, though).

In general, things work as for any other service provided by the operating system. First, the system provides some abstraction for using the network. As we will be seeing, Plan 9 uses also the file abstraction as its primary interface for using networks. Of course, files used to represent a network have a special meaning, i.e., behave in a particular way, but they are still used like files. Other operating systems use a whole bunch of extra system calls instead, to provide the interface for their network abstraction. Nevertheless, the ideas, and the programatic interface that we will see, are very similar.

Upon such system-provided abstraction, library functions may provide a more convenient interface for the application programmer. And of course, in the end, there many programs already installed in the system that, using these libraries, provide some services for the user.

A network in Plan 9 is a set of devices that provide the ability to talk with other machines using some physical medium (e.g, some type of wire or the air for radio communication).

A network device in Plan 9 may be an actual piece of hardware, but it can also be a piece of software used to speak some protocol. For example, most likely, your PC includes an ethernet card. It uses an RJ45 connector to plug your computer to an Ethernet network (just some type of cabling and conventions). The interface for the ethernet device in Plan 9 is just a file tree, most likely found at `/net/ether0`

```
; lc /net/ether0
0       1       2       addr    clone   ifstats stats
```

Machines attached to the wire have addresses, used by the network hardware to identify different machines attached to the wire. Networks using wireless communication are similar, but use the air as their “wire”. We can use the file interface provided by Plan 9 for our ethernet device to find out which one is its address:

```
; cat /net/ether0/addr
000c292839fc;
```

As you imagine, this file is just an interface for using your ethernet device, in this case, for asking for its address.

Once you have the hardware (e.g., the ethernet card) for exchanging messages with other machines attached to the same medium (wiring or air), your machine and exchange bytes with them. The problem remains of how to send messages to any machine in the Internet, even if it is not attached to the same wire your machine is attached at. One protocol very important to the Internet, IP (Internet Protocol), is provided in Plan 9 by a device driver called IP. This protocol is called a network protocol because it gives an address to each machine in the Internet, its IP-address, and it knows how to reach any machine, given its address. The interface for the IP network in Plan 9 is similar to the one we saw for Ethernet:

```
; lc /net/ipifc
0       1       clone   stats
```

This is not yet enough for communicating with programs across the internet. Using IP, you may talk to one machine (and IP cares about how to reach that machine through the many different wires and machines you need to cross). But you need to be able to talk to one *process*. This is achived by using another protocol, built upon the network protocol. This kind of protocol gives addresses for "mailboxes" within each machine, called *ports*. Therefore, an address for this protocol is a combination of a machine address (used to reach that machine through the underlying network protocol) and a *port* number.

In few words, the network protocol gives addresses for each machine and knows how to exchange messages between machines. Today, you are going to use IP as your network protocol. The transport protocol gives port numbers for processes to use, and knows how to deliver messages to a particular port at a particular machine. Think of the network address as the address for a building, and the port number as the number for a mailbox in the building.

Some transport protocols provide an abstraction similar to the postal service. They deliver individual messages that may arrive out of order and may even get lost in the way. Each such message is called a *datagram*, which is the abstraction provided by this kind of transport. In the Internet, the datagram service is usually UDP. The IP device driver in Plan 9 provides an interface for using UDP, similar to the ones we saw for other protocols and network devices:

```
; lc /net/udp
0       1       clone   stats
```

Other transports use the ability to send individual messages to build a more convenient abstraction for maintaining dialogs, similar to a pipe. This abstraction is called a **connection**. It is similar to a pipe, but differs from it in that it can go from one port at one machine to another port at a different machine in the network. This type of communication is similar to a phone call. Each end has an address (a phone number), they must establish a connection (dial a number, pickup the phone), then they can speak to each other, and finally, they hangup. The analogy cannot be pushed too far, for example, a connection may be established if both ends call each other, which would not be feasible when making a phone call. But you get the idea. In the Internet, the most popular protocol that provides connections is TCP, it provides them using IP as the underlying transport protocol (hence the name TCP/IP for this suite of protocols). The IP device driver in Plan 9 provides the interface for using TCP. It has the now familiar file interface for using a network in Plan 9:

```
; lc /net/tcp
0       11      14      17      2       22      stats
1       12      15      18      20      23      26
10      13      16      19      21      24      clone
```

Each network is represented in Plan 9 as a directory, that has at least one `clone` file, and several other directories, called *line* directories. Opening the `clone` file reserves a new connection, and creates a directory that represents the interface for the new *line* used to establish a connection. Line directories are named with a number, and kept within the directory for the network. For example, `/net/tcp/14` is the interface for our TCP connection number 14. It doesn't need to be a fully established connection, it may be in the process of getting established. But in any case, the directory represents what can be a particular, individual, TCP connection. The program that opens `clone` may read this file to discover the number assigned to the line directory just created.

As shown in figure 6.1, for each connection Plan 9 provides at least a `ctl` file and a `data` file. For example,

```
; lc /net/tcp/14
ctl     data    err     listen  local   remote  status
```

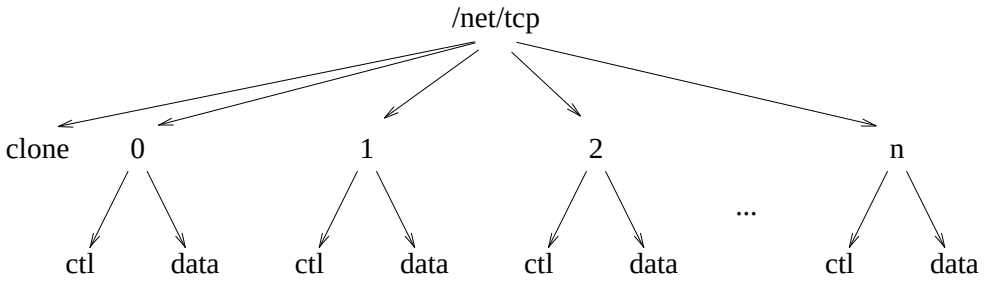

Figure 6.1: *The file interface for a network (protocol) in Plan 9.*

The file ctl can be used to perform control operations to the connection. For example, to hangup (break) this connection, we can just

```
; echo hangup >/net/tcp/14
```

The `data` file is used to send and receive bytes through the connection. It can be used very much like one end of a pipe. Writing to the data file delivers bytes through the connection that are to be received at the other end. Reading from the data file retrieves bytes sent from the process writing at the other end. Just like a pipe. Only that, if a transport provides datagrams, each write to a `data` file will send a different datagram, and it may arrive out of order or get lost.

There are more differences. An important one is that many transport protocols, including TCP, do not respect message boundaries. This means that data sent through a connection by several writes may be received at the other end by a single read. If your program has to receive messages from a network connection, it must know how much to read for each message. A single call to read may return either part of a message or perhaps more than one message.

In the line directory for our TCP connection, the `local` file has the local address (including the port number) for the connection. This identifies the local end of the *pipe*. The `remote` file serves the same purpose for the other end of the connection.

A network address in Plan 9 is a string that specifies the network (e.g., the protocol) to use, the machine address, and the port number. For example, `tcp!193.147.81.86!564` is a network address that says: Using the TCP protocol, the machine address is 193.147.81.86, and the port number is 564. Fortunately, in most cases, we may use names as well. For example, the address `tcp!whale!9fs` is equivalent to the previous one, but uses the machine name, `whale`, and the service name, `9fs`, instead of the raw addresses understood by the network software. Often, ports are used by programs to provide services to other programs in the network. As a result, a port name is also known as a **service** name.

From the shell, it is very easy to create connections. The `srv` program dials a network address and, once it has established a connection to that address, posts a file descriptor for the connection at `/srv`. This descriptor comes from opening the `data` file in the directory for the connection, but you may even forget this. Therefore,

```
; srv tcp!whale!9fs
post...
```

posts at `/srv/tcp!whale!9fs` a file descriptor that corresponds to an open network connection from this machine to the port named `9fs` at the machine known as `whale`, in the network speaking the protocol `tcp`.

To connect to the web server for LSUB, we may just

```
; srv tcp!lsub.org!http
post...
```

Here, `tcp` is just a shorthand for `/net/tcp`, which is the real (file) name for such network

in Plan 9. Now we can see that `/srv/tcp!lsub.org!http` is indeed a connection to the web server at `lsub.org` by writing an HTTP request to this file and reading the server's reply.

```
; echo GET /index.html  >>/srv/tcp!lsub.org!http          Get the main web page
; cat /srv/tcp!lsub.org!http
<html>
<head>
<title> Laboratorio de Sistemas --- ls </title>
<link rev="made" href="mailto:ls@plan9.escet.urjc.es">
</head>
<body  BGCOLOR=white>
<h1>   ls --- Laboratorio de Sistemas [ubicuos] del GSyC  </h1>
...and more output omitted here...
;
```

If we try to do the same again, it will not work, because the web server hangs up the connection after attending a request:

```
; echo GET / >>/srv/tcp!lsub.org!http
; cat /srv/tcp!lsub.org!http
cat: error reading /srv/tcp!lsub.org!http: Hangup
; echo GET / >>/srv/tcp!lsub.org!http
echo: write error: Hangup
```

And, as you can see, it takes some time for our machine to notice. The first write seemed to succeed. Our machine was trying to send the string `GET...` to the web server, but it couldn't really send it. The connection was closed and declared as hung up. Any further attempt to use it will be futile. What remains is to remove the file from `/srv`.

```
; rm /srv/tcp!lsub.org!http
```

There is a very popular command named `telnet`, that can be used to connect to servers in the Internet and talk to them. It uses the, so called, *telnet protocol*. But in few words, it dials an address, and thereafter it sends text from your console to the remote process at the other end of the connection, and writes to your console the text received. For example, this command connects to the e-mail server running at `lsub.org`, and we use our console to ask this server for help:

```
; telnet -r tcp!lsub.org!smtp
connected to tcp!lsub.org!smtp on /net/tcp/52
220 lsub.org SMTP
help
250 Read rfc821 and stop wasting my time
Del
```

We gave the option `-r` to `telnet`, to ask it not to print *carriage-return* characters (its protocol uses the same convention for new lines used by DOS). When telnet connected to the address we gave, it printed a diagnostic message to let us know, and entered a loop to send the text we type, and to print the text it receives from the other end. Our mail server wrote a salutation through the connection (the line starting `220...`), and then we typed `help`, which put our mail server into a bad mood. We interrupted this program by hitting *Delete* in the terminal, and the connection was terminated when `telnet` died. A somewhat abrupt termination.

It is interesting to open serveral windows, and connect from all of them to the same address. Try it. Do you see how *each* `telnet` is using its own connection? Or, to put it another way, all the connections have the *same* address for the other end of the connection, yet they are *different* connections.

To name a connection, it does not suffice to name the address for one of its ends. You *must* give both addresses (for the two ends) to identify a connection. It is the four identifiers local address, local port, remote address, and remote port, what makes a connection unique.

It is very important to understand this clearly. For example, in our `telnet` example, you cannot know which connection are you talking about just by saing "The connection to `tcp!lsub.org!smtp`". There can be a dozen of such connections, all different, that happen to reach that particular address. They would differ in the addresses for their other extremes.

6.2. Names

Above, we have been using names for machines and services (ports). However, these names must be translated into addresses that the network software could understand. For example, the machine name `whale` must be translated to an IP address like `193.147.81.86`. The network protocol (IP in Internet) knows nothing about names. It knows about machine addresses. In the same way, the transport protocol TCP knows nothing about the service with name `http`. But it does know how to reach the port number `80`, which is the one that corresponds to the HTTP service.

Translating names into addresses (including machine and service names) is done in a different way for each kind of network. For example, the Internet has a name service known as DNS (domain name service) that knows how to translate from a name like `whale.lsub.org` into an IP address and viceversa. Besides, for some machines and services there may be names that exist only within a particular organization. Your local system administrator may have assigned names to machines that work only from within your department or laboratory. In any case, all the information about names, addresses, and how to reach the Internet DNS is kept in a (textual) database known as the *network database*, or just `ndb`. For example, this is the entry in our `/lib/ndb/local` file for `whale`:

```
dom=whale.lsub.org ip=193.147.81.86 sys=whale
```

When we used `whale` in the examples above, that name was translated into `193.147.81.86` and that was the address used. Also, this is the entry in our `/lib/ndb/common` file for the service known as `9fs` when using the TCP protocol:

```
tcp=9fs port=564
```

When we used the service name `9fs`, this name was translated into the port number `564`, that was the port number used. As a result, the address `tcp!whale!9fs` was translated into `tcp!193.147.81.86!564` and this was used instead. Names are for humans, but (sadly) the actual network software prefers to use addresses.

All this is encapsulated into a program that does the translation by itself, relieving from the burden to all other programs. This program is known as the *connection server*, or `cs`. We can query the connection server to know which address will indeed be used when we write a particular network address. The program `csquery` does this. It is collected at `/bin/ndb` along with other programs that operate with the network data base.

```
; ndb/csquery
> tcp!whale!9fs
/net/tcp/clone 193.147.81.86!564
>
```

The "`>`" sign is the prompt from `csquery`, it suggests that we can type an address asking for its translation. As you can see, the connection server replied by giving the path for the `clone` file that can be used to create a new TCP connection, and the address as understood by TCP that corresponds to the one we typed. No one else has to care about which particular network, address, or port number corresponds to a network address.

All the information regarding the connections in use at your machine can be obtained by looking at the files below `/net`. Nevertheless, the program `netstat` provides a convenient way for obtaining statistics about what is happening with the network. For example, this is what is happening now at my system:

```
; netstat
tcp  0    nemo       Listen       audio      0          ::
tcp  1               Established  5757       9fs        whale.lsub.org
tcp  2    nemo       Established  5765       ads        whale.lsub.org
tcp  3    nemo       Established  5759       9fs        whale.lsub.org
tcp  4    nemo       Listen       what       0          ::
tcp  5    nemo       Established  5761       ads        whale.lsub.org
tcp  6    nemo       Established  5766       ads        whale.lsub.org
tcp  7    nemo       Established  5763       9fs        whale.lsub.org
tcp  8    nemo       Listen       kbd        0          ::
...many other lines of output for tcp...
udp  0    network    Closed       0          0          ::
udp  1    network    Closed       0          0          ::
```

Each line of output shows information for a particular line directory. For example, the TCP connection number 1 (i.e., that in `/net/tcp/1`) is established. Therefore, it is probably being used to exchange data. The local end for the connection is at port 5757, and the remote end for the connection is the port for service `9fs` at the machine with name `whale.lsug.org`. This is a connection used by the local machine to access the 9P file server at `whale`. It is being used to access our main file server from the terminal where I executed `netstat`. The states for a connection may depend on the particular protocol, and we do not discuss them here.

In some cases, there may be problems to reach the name service for the Internet (our DNS server), and it is very useful to call `netstat` with the `-n` flag, which makes the program print just the addresses, without translating them into (more readable) names. For example,

```
; netstat -n
tcp  0    nemo       Listen       11004      0          ::
tcp  1               Established  5757       564        193.147.71.86
tcp  2    nemo       Established  5765       11010      193.147.71.86
tcp  3    nemo       Established  5759       564        193.147.71.86
tcp  4    nemo       Listen       11003      0          ::
tcp  5    nemo       Established  5761       11010      193.147.71.86
...many other lines of output
```

It is very instructive to compare the time it takes for this program to complete with, and without using `-n`.

To add yet another tool to your network survival kit, the `ip/ping` program sends particular messages that behave like probes to a machine (to an IP address, which is for a network interface found at a machine, indeed), and prints one line for each probe reporting what happen. It is very useful because it lets you know if a particular machine seems to be alive. If it replies to a probe, the machine is alive, no doubt. If the machine does not reply to any of the probes, it might be either dead, or disconnected from the network. Or perhaps, it is your machine the one disconnected. If only some probes get replied, you are likely to have bad connectivity (your network is loosing too many packets). Here comes an example.

```
; ip/ping lsub.org
sending 32 64 byte messages 1000 ms apart
0: rtt 152 µs, avg rtt 152 µs, ttl = 255
1: rtt 151 µs, avg rtt 151 µs, ttl = 255
2: rtt 149 µs, avg rtt 150 µs, ttl = 255
...
```

In the output, `rtt` is for *round trip time*, the time for getting in touch and receiving the reply.

6.3. Making calls

For using the network from a C program, there is a simple library that provides a more convenient interface that the one provided by the file system from the network device. For example, this is our simplified version for `srv`. It dials a given network address to establish a connection and posts a file descriptor for the open connection at `/srv`.

srv.c

```
#include <u.h>
#include <libc.h>

void
main(int argc, char* argv[])
{
        int     fd, srvfd;
        char*   addr;
        char    fname[128];

        if (argc != 2){
                fprint(2, "usage: %s netaddr\n", argv[0]);
                exits("usage");
        }

        addr = netmkaddr(argv[1], "tcp", "9fs");
        fd = dial(addr, nil, nil, nil);
        if (fd < 0)
                sysfatal("dial: %s: %r", addr);

        seprint(fname, fname+sizeof(fname), "/srv/%s", argv[1]);
        srvfd = create(fname, OWRITE, 0664);
        if (srvfd < 0)
                sysfatal("can't post %s: %r", fname);
        if (fprint(srvfd, "%d", fd) < 0)
                sysfatal("can't post file descriptor: %r");
        close(srvfd);
        close(fd);
        exits(nil);
}
```

Using `argv[1]` verbatim as the network address to dial, would make the program work only when given a complete address. Including the network name and the service name. Like, for example,

```
; 8.srv tcp!whale!9fs
```

Instead, the program calls `netmkaddr` which is a standard Plan 9 function that may take an address with just the machine name, or perhaps the network name and the machine name. This function completes the address using default values for the network and the service, and returns a full address ready to use. We make `tcp` the default value for the network (protocol) and `9fs` as the default value for the service name. Therefore, the program admits any of the following, with the same effect that the previous invocation:

```
; 8.srv tcp!whale
; 8.srv whale
```

The actual work is done by `dial`. This function dials the given address and returns an open file descriptor for the connection's data file. A write to this descriptor sends bytes through the connection, and a read can be used to receive bytes from it. The function is used in the same way for both datagram protocols and connection-oriented protocols. The connection will be open as long as the file descriptor returned remains open.

```
; sig dial
        int dial(char *addr, char *local, char *dir, int *cfdp)
```

The parameter `local` permits specifying the local address (for network protocols that allow doing so). In most cases, given `nil` suffices, and the network will choose a suitable (unused) local port for the connection. When `dir` is not nil, it is used by the function as a buffer to copy the path for the line directory representing the connection. The buffer must be at least 40 bytes long. We changed the previous program to do print the path for the line directory used for the connection:

```
        fd = dial(addr, nil, dir, nil);
        if (fd < 0)
                sysfatal("dial: %s: %r", addr);
        print("dial: %s0, dir);
```

And this is what it said:

```
; 8.srv tcp!whale!9fs
dial: /net/tcp/24
```

The last parameter for dial, `cfdp` points to an integer which, when passing a non-nil value, can be used to obtain an open file descriptor for the connection. In this case, the caller is responsible for clossing this descriptor when appropriate. This can be used to write to the control file requests to tune properties for the connection, but is usually unnecessary.

There is a lot of useful information that we may obtain about a connection by calling `getnetconninfo`. This function returns nothing that could not be obtained by reading files from files in the line directory of the connection, but it is a very nice wrap that makes things more convenient. In general, this is most useful in servers, to obtain information to try to identify the other end of the connection, (i.e., the client). However, because it is much easier to make a call than it is to receive one, we prefer to show this functionality here instead.

Parameters for `netconninfo` are the path for a line directory, and one of the descriptors for either a control or a data file in the directory. When nil is given as a path, the function uses the file descriptor to locate the directory, and read all the information to be returned to the caller. The function allocates memory for a `NetConnInfo` structure, fills it with relevant data, and returns a pointer to it

```
typedef struct NetConnInfo NetConnInfo;
struct NetConnInfo
{
        char    *dir;           /* connection directory */
        char    *root;          /* network root */
        char    *spec;          /* binding spec */
        char    *lsys;          /* local system */
        char    *lserv;         /* local service */
        char    *rsys;          /* remote system */
        char    *rserv;         /* remote service */
        char    *laddr;         /* local address */
        char    *raddr;         /* remote address */
};
```

This structure must be released by a call to `freenetconninfo` once it is no longer necessary. As an example, this program dials the address given as a parameter, and prints all the information returned by `getnetconninfo`. Its output for dialing `tcp!whale!9fs` follows.

conninfo.c

```
#include <u.h>
#include <libc.h>

void
main(int argc, char* argv[])
{
        int     fd, srvfd;
        char*   addr;
        NetConnInfo*i;
        if (argc != 2){
                fprint(2, "usage: %s netaddr\n", argv[0]);
                exits("usage");
        }

        addr = netmkaddr(argv[1], "tcp", "9fs");
        fd = dial(addr, nil, nil, nil);
        if (fd < 0)
                sysfatal("dial: %s: %r", addr);
        i = getnetconninfo(nil, fd);
        if (i == nil)
                sysfatal("cannot get info: %r");
        print("dir:\t%s\n", i->dir);
        print("root:\t%s\n", i->root);
        print("spec:\t%s\n", i->spec);
        print("lsys:\t%s\n", i->lsys);
        print("lserv:\t%s\n", i->lserv);
        print("rsys:\t%s\n", i->rsys);
        print("rserv:\t%s\n", i->rserv);
        print("laddr:\t%s\n", i->laddr);
        print("raddr:\t%s\n", i->raddr);
        freenetconninfo(i);
        close(fd);
        exits(nil);
}
```

```
; 8.out tcp!whale!9fs
dir:    /net/tcp/46
root:   /net
spec:   #I0
lsys:   212.128.4.124
lserv:  6672
rsys:   193.147.71.86
rserv:  564
laddr:  tcp!212.128.4.124!6672
raddr:  tcp!193.147.71.86!564
```

The line directory for this connection was `/net/tcp/46`, which belongs to the network interface at `/net`. This connection was using `#I0`, which is the first IP interface for the machine. The remaining output should be easy to understand, given the declaration of the structure above, and the example output shown.

6.4. Providing services

We know how to connect to processes in the network that may be providing a particular service. However, it remains to be seen how to provide a service. In what follows, we are going to implement an echo server. A client for this program would be another process connecting to this service to obtain an *echo service.* This program provides the service (i.e., provides the echo) and is therefore a *server.* The echo service, surprisingly enough, consists on doing echo of what a client writes. When the echo program reads something, writes it back through the same connection, like a proper echo.

The first thing needed is to **announce** the new service to the system. Think about it. To allow other processes to *connect* to our process, it needs a port for itself. This is like allocating a "mailbox" in the "building" to be able to receive mail. The function `announce` receives a network address and announces it as an existing place where others may send messages. For example,

```
announce("tcp!alboran!echo", dir);
```

would allocate the TCP port for the service named `echo` and the machine named `alboran`. This makes sense only when executed in that machine, because the port being created is an abstraction for getting in touch with a local process. To say it in another way, the address given to announce must be a local address. It is a better idea to use

```
announce("tcp!*!echo", dir);
```

instead. The special machine name "`*`" refers to any local address for our machine. This call reserves the port `echo` for any interface used by our machine (not just for the one named `alboran`). Besides, this call to `annouce` now works when used at any machine, no matter its name.

This function returns an open file descriptor to the `ctl` file of the line directory used to announce the port. The second parameter is updated with the path for the directory. Note that this line directory is an artifact which, although has the same interface, is *not* a connection. It is used just to maintain the reservation for the port and to prepare for receiving incomming calls. When the port obtained by a call to `announce` is no longer necessary, we can close the file descriptor for the `ctl` file that it returns, and the port will be released.

This program announces the port 8899, and sleeps forever to let us inspect what happen.

ann.c

```
#include <u.h>
#include <libc.h>

void
main(int argc, char* argv[])
{
        int     cfd;
        char    dir[40];

        cfd = announce("tcp!*!9988", dir);
        if (cfd < 0)
                sysfatal("announce: %r");
        print("announced in %s\n", dir);
        for(;;)
                sleep(1000);
}
```

We may now do this

```
; 8.ann &
; announced in /net/tcp/52       We typed return here, to let you see
; netstat | grep 9988
tcp  52   nemo       Listen       9988       0          ::
```

According to `netstat`, the TCP port number 9988 is listening for incomming calls. Note how the path printed by our program corresponds to the TCP line number 52.

Now let's try to run the program again, without killing the previous process.

```
; 8.out
announce: announce writing /net/tcp: address in use
```

It fails! Of course, there is another process already using the TCP port number 9988. This new process cannot announce that port number again. It will be able to do so only when nobody else is using it:

```
; kill 8.ann|rc
; 8.ann &
; announced in /net/tcp/52
```

Our program must now await for an incomming call, and accept it, before it could exchange data with the process at the other end of the connection. To wait for the next call, you may use `listen`. This name is perhaps misleading because, as you could see, after `announce`, the TCP line is already listening for calls. Listen needs to know the line where it must wait for the call, and therefore it receives the directory for a previous announce.

Now comes an important point, to leave the line listening while we are attending a call, calls are attended at a *different* line than the one used to listen for them. This is like an automatic transfer of a call to another phone line, to leave the original line undisturbed and ready for a next call. So, after `listen` has received a call, it obtains a new line directory for the call and returns it. In particular, it returns an open file descriptor for its `ctl` file and its path.

We have modified our program to wait for a single call. This is the result.

listen.c

```
#include <u.h>
#include <libc.h>

void
main(int argc, char* argv[])
{
        int     cfd, lfd;
        char    adir[40];
        char    dir[40];

        cfd = announce("tcp!*!9988", adir);
        if (cfd < 0)
                sysfatal("announce: %r");
        print("announced in %s (cfd=%d)\n", adir, cfd);
        lfd = listen(adir, dir);
        print("attending call in %s (lfd=%d)\n", dir, lfd);
        for(;;)
                sleep(1000);    // let us see
}
```

When we run it, it waits until a call is received:

```
; 8.listen
announced in /net/tcp/52 (cfd=10)
```

At this point, we can open a new window and run `telnet` to connect to this address

```
; telnet tcp!$sysname!9988
connected to tcp!alboran!9988 on /net/tcp/46
```

which makes our program receive the call:

```
attending call in /net/tcp/54 (lfd=11)
```

You can see how there are two lines used. The line number 52 is still listening, and the call received is placed at line 54, where we might accept it. By the way, the line number 46 is the other end of the connection.

Now we can do something useful. If we accept the call by calling `accept`, this function will provide an open file descriptor for the `data` file for the connection, and we can do our echo business.

netecho.c

```
#include <u.h>
#include <libc.h>

void
main(int argc, char* argv[])
{
        int     cfd, lfd, dfd;
        long    nr;
        char    adir[40];
        char    ldir[40];
        char    buf[1024];

        cfd = announce("tcp!*!9988", adir);
        if (cfd < 0)
                sysfatal("announce: %r");
        print("announced tcp!*!9988 in %s\n", adir);
        for(;;){
                lfd = listen(adir, ldir);
                if (lfd < 0)
                        sysfatal("listen: %r");
                dfd = accept(lfd, ldir);
                if (dfd < 0)
                        sysfatal("can't accept: %r");
                close(lfd);
                print("accepted call at %s\n", ldir);
                for(;;){
                        nr = read(dfd, buf, sizeof buf);
                        if (nr <= 0)
                                break;
                        write(dfd, buf, nr);
                }
                print("terminated call at %s\n", ldir);
                close(dfd);
        }
}
```

If we do as before, and use `telnet` to connect to our server and ask for a nice echo, we get the echo back. After quitting `telnet`, we can connect again to our server and it attends the new call.

```
; telnet -r tcp!$sysname!9988
connected to tcp!alboran!9988 on /net/tcp/46
Hi there!
Hi there!
Del
; telnet -r tcp!$sysname!9988
connected to tcp!alboran!9988 on /net/tcp/54
Echo echo...
Echo echo...
Del
;
```

And this is what our server said in its standard output:

```
; 8.netecho
announced tcp!*!9988 in /net/tcp/52
accepted call at /net/tcp/54
terminated call at /net/tcp/54
accepted call at /net/tcp/55
terminated call at /net/tcp/55
```

The program is very simple. To announce our port, wait for call, and accept it, it has to call just `announce`, `listen`, and `accept`. At that point, you have an open file descriptor that may be used as any other one. You just read and write as you please. When the other end of the connection gets closed, a reader receives an EOF indication in the conventional way. This means that connections are used like any other file. So, you already know how to use them.

Out program has one problem left to be addressed. When we connected to it using `telnet`, there was only one client at a time. For this program, when one client is connected and using the echo, nobody else is attended. Other processes might connect, but they will be kept on hold waiting for this process to call `listen` and `accept.` This is what is called a **sequential server**, because it attends one client after another. You can double check this by connecting from two different widows. Only the first one will be echoing. The echo for the second to arrive will not be done until you terminate the first client.

A sensible thing to do would be to fork a new process for each client that connects. The parent process may be kept listening, waiting for a new client. When one arrives, a child may be spawned to serve it. This is called a **concurrent server**, because it attends multiple clients concurrently. The resulting code is shown below.

There are some things to note. An important one is that, as you know, the child process has a copy of all the file descriptors open in the parent, by the time of the fork. Also, the parent has the descriptor open for the new call received after calling `listen`, even though it is going to be used just by the child process. We close `lfd` in the parent, and `cfd` in the child.

We might have left `cfd` open in the child, because it would be closed when the child terminates by calling `exits`, after having received an end of file indication for its connection. But in any case, it should be clear that the descriptor is open in the child too.

Another important detail is that the child now calls `exits` after attending its connection, because that was its only purpose in life. Because this process has (initially) all the open file descriptors that the parent had, it may be a disaster if the child somehow terminates attending a client and goes back to call `listen`. Well, it would be disaster because it is *not* what you expect when you write the program.

cecho.c

```
#include <u.h>
#include <libc.h>

void
main(int argc, char* argv[])
{
        int     cfd, lfd, dfd;
        long    nr;
        char    adir[40];
        char    ldir[40];
        char    buf[1024];

        cfd = announce("tcp!*!9988", adir);
        if (cfd < 0)
                sysfatal("announce: %r");
        print("announced tcp!*!9988 in %s\n", adir);
        for(;;){
                lfd = listen(adir, ldir);
                if (lfd < 0)
                        sysfatal("listen: %r");
                switch(fork()){
                case -1:
                        sysfatal("fork: %r");
                case 0:
                        close(cfd);
                        dfd = accept(lfd, ldir);
                        if (dfd < 0)
                                sysfatal("can't accept: %r");
                        close(lfd);
                        print("accepted call at %s\n", ldir);
                        for(;;){
                                nr = read(dfd, buf, sizeof buf);
                                if (nr <= 0)
                                        break;
                                write(dfd, buf, nr);
                        }
                        print("terminated call at %s\n", ldir);
                        exits(nil);
                default:
                        close(lfd);
                }
        }
}
```

6.5. System services

How some machines may provide services automatically. Which one is the program for the tcp echo service? how does this work?