Lecture 21-22: Sockets

Everything is a File

The makers of Unix were quite clever when they made this overarching decision. It is interesting that most computer constructs can be simplified to the idea of a file. To this end, simply knowing a file descriptor can allow you to communicate successfully with all kinds of things, such as:

- The terminal, standard in, standard out and standard error
- Input and output files (basically just text saved in a file)
- Pipes (allowing communication between processes)

So, what else could we cram into this definition of 'a file'. Well, one thing we have not yet covered, something completely fundamental to computing is… processes.

"But" I hear you say, is not a pipe two processes communicating? Is not a pipe treating another process as a file? The answer is 'nearly', but 'not quite'. In the case of pipes both processes are communicating with a mini-construct-like thing (i.e. the buffer in the pipe) rather than directly with one another. Furthermore, the challenge of using a pipe is that so far while we have connected two processes, we have done so within the scope of a single program. We have not done any fancy 'connect to programs' together code yet.

So, before we get to sockets, we are first going to talk about connecting processes together (which is not unlike sockets).

Connecting Processes

Given the tools we already have at our disposal, connecting two processes is actually a completely possible task. The easiest way would be to have a temporary file which represents a buffer between the two processes. One process writes to it, the other reads to it. Of course, this solution is not without its problems (which are big). The first rather trivial problem is that reading to and writing from files is slow. The second, more substantial issue is how would you synchronize this mess? How could one process tell if the other process was actively writing, actively reading, had finished writing, had finished reading? There are not elegant answers to these questions.

As mentioned before, we can also connect processes via pipes. However, as also mentioned before, this only really works because both processes have access to the same pipe. This implies a parent/child relationship using forks which, as mentioned before is a rather harsh limitation.

fopen(), fdopen() and popen()

So, you should already be somewhat familiar with fopen. It is used for opening files either for reading or writing (or both). How does it actually work? It opens a file and then associates a stream with it.

By comparison, fdopen() associates a stream with an existing file descriptor (**f**ile **d**escriptor **open**).

popen(), which is syntactically very similar to the above (and stands for **p**rocess **open**) is used to open processes, rather than files. An example is below.

```
#include <stdio.h>
FILE * popen(const char * command, const char * type);
int pclose(FILE * stream);
```

In this case, the ‘command’ is the command to run in the system and the ‘type’ is “r” or “w” (i.e. read or write). Given that all processes have an input stream and an output stream, it is surprising how similar processes are to files (though this is hardly a coincidence).

So, what is popen actually doing? Under the covers, it is:

- Creating a pipe
- Forking
- Invoking the shell

It is not particularly different from the work you did in Assignment #2 in that case.

Reading and writing via popen has some default behaviour.

When you popen a process for **writing**, whenever you write to the stream defined by the popen call, you are simply writing to the processes standard input. The standard output of the process is matched to the initiating processes (i.e. the original standard out). This can be changed, but it is useful to know.

Likewise when you popen a process for **reading**, whenever you read from the stream defined by the poepn call, you are simply reading from the processes standard output (not surprising). The standard input of the command called by popen is… the same as the process that invoked popen. This is less desirable, but can equally be changed.

Interfaces

So before we get to socket, it is worth saying something about interfaces. Now from your studies in C++, you should be familiar with the concept of an interface such as in the functions:

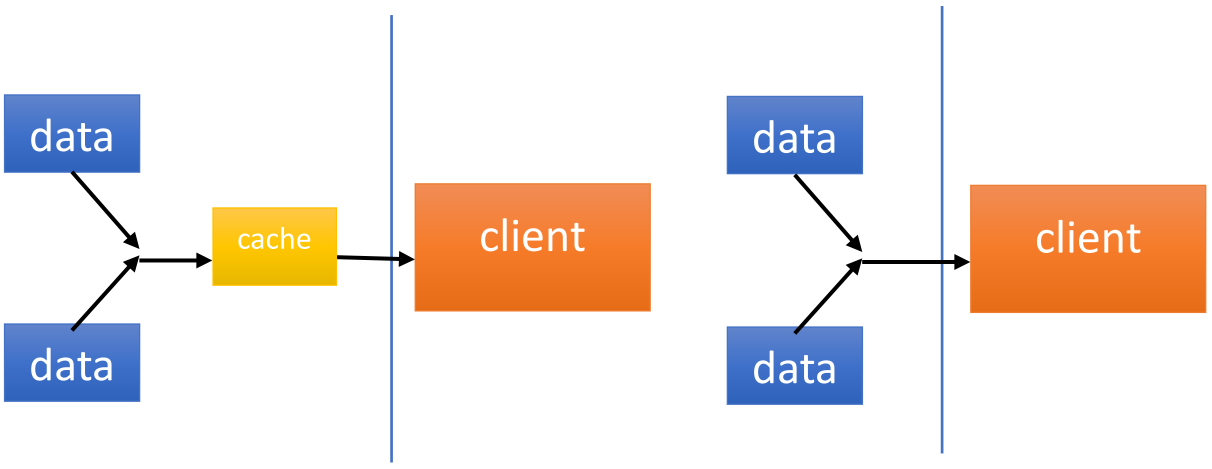
```
// returns sum of integers from a to b
int sum(int a, int b)
{
	int total = 0;
	for (int i = a; i < b; i++)
		total = total + i;
	return total;
}
```

AND

```
// returns sum of integers from a to b
int sum(int a, int b)
{
	return (a + b)/2 * (b – a + 1);
}
```

While these two functions differ in their implementation, they have the same overall purpose, results and ‘interface’ (that is an end user can’t tell the difference).

When it comes to having processes communicating, it is important to consider the interface, i.e. the rules that govern how the programs will interact. One advantage of having a static interface is that grants the processes on either end of the interface the flexibility to do things how they want to, as long as they comply with the interface.



Consider the above image. In both the left and right examples, the client doesn't really need to know whether the data it is receiving was generated a long time ago and just copied from cache, or whether it was created instantaneously on the spot.

Sockets

So now we know about connecting to processes, and the idea of interfaces, we can finally start discussing sockets. Similarly, to popen, we are going to read and write to another process as if it were a file. The main difference is that popen, like pipes was essentially working within a single program chain on a single computer. Sockets create an opportunity for two completely independent processes to communicate with each other. They must have an agreed interface, but other than that, there are not a lot of constraints on either side of the socket. It also means that sockets are going to be a lot more flexible in that 'dynamically doing things' way.

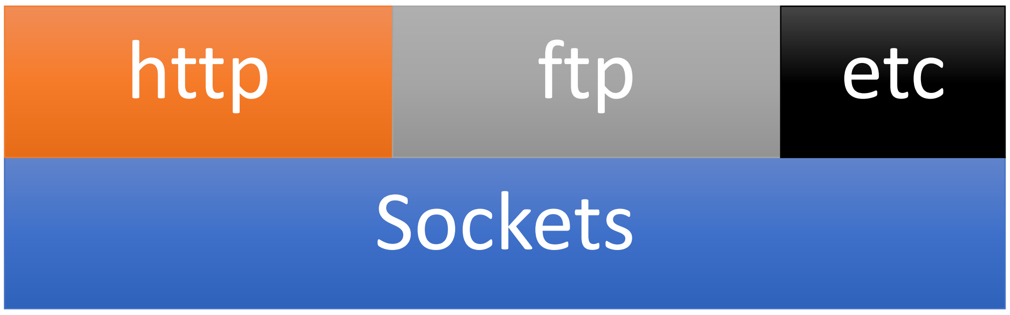
Common Uses for Sockets

Sockets can be used for all kinds of things, but traditionally find their most common usage in internet contexts including:

- Web servers
- Web clients
- FTP clients

Where do sockets sit?

Now, you may be familiar with lots of different protocols such as http and ftp. These different protocols are basically defining the formatting of information being sent. However, all of these protocols rely on the existence of sockets. If you think about a socket as a pipe between processes, then the protocols (http etc) are just different ways of sending bytes down that pipe.



Clients and Servers

So, these are again words with which you should be familiar. Let us have a quick think about why the internet and other systems are built using this framework. A server is a program which seems to be around for a while and is perpetually waiting for things (clients) to connect to it. A client is a program which is expecting to find a server at a particular location and requests a connection from it.

Is this the only way to connect two processes? In a way it is. If you have two independent things that want to connect to each other, at some point one of them is going to be waiting for the other. Therefore, embracing the fundamental fact, servers are placed in the role of 'waiter', and client as 'interrupter'. After this initial connection has been established the behaviour of server and client is quite arbitrary.

The server paradigm predates the internet and computers. A good example (which ceased to exist a few years ago) is the 'time'. It used to be possible to call (on your phone) 'the time'. This was a server that when called simply did the following:

At the beep, the time will be eleven twenty-three and forty seconds.

At the beep, the time will be eleven twenty-three and fifty seconds.

Establishing a Connection

The traditional way of connecting to another computer across a network is via an IP address. This will be covered in greater detail in other courses but briefly, an IP address is an address similar to a house address. It tells you where the information is meant to go to/come from. Along with the IP address is the port number. The port number is used to determine what is being communicated, a bit like a radio frequency.

So… IP Address:

- Unique for a computer
- Several processes can communicate using the same IP address (i.e. same computer)

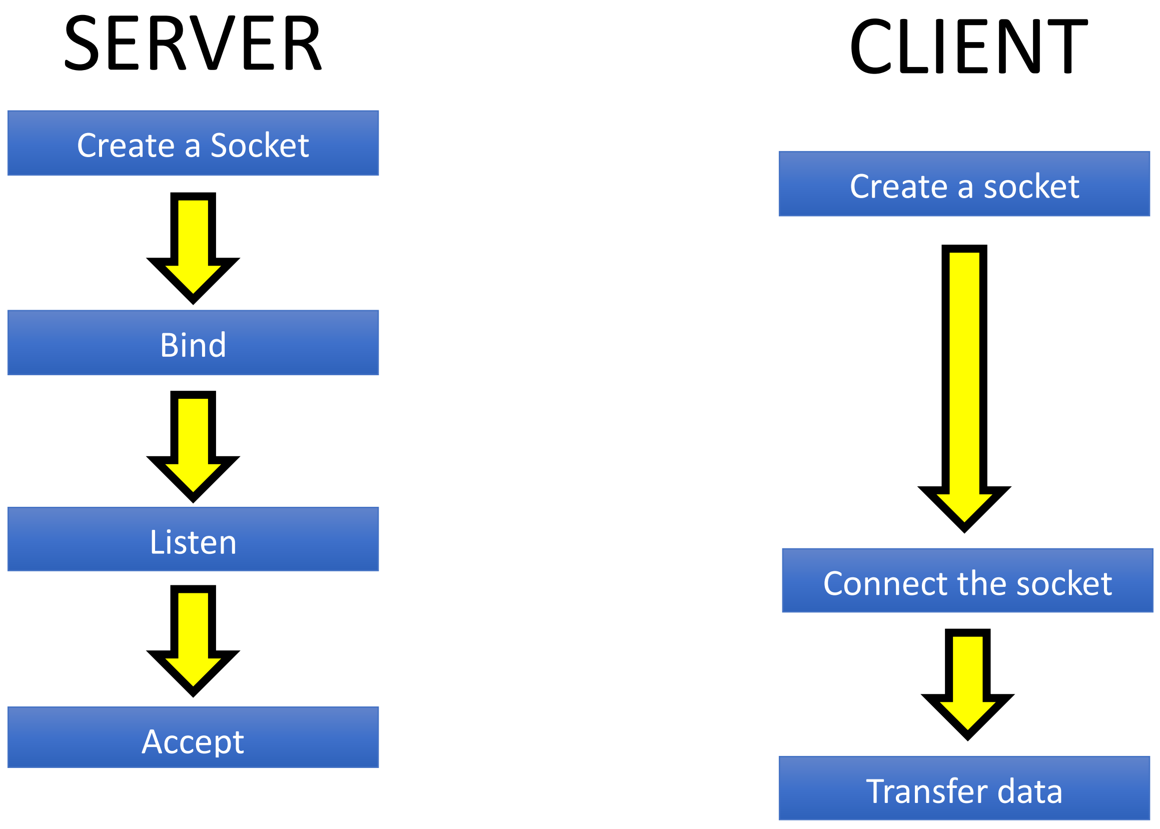
Port Number:

- Unique for an application
- Several computers can communicate using the same Port number (i.e. same application)

Connecting a Server and Client

Below is a figure depicting the order of actions that occur when connecting. Briefly, the server creates a socket. The server needs to specify a port but given it doesn't know who will connect there is no need for an IP address at this stage. It does however need to have a name for itself. The server then needs to bind the socket and begin listening on it.

At this point it is assumed that some period of time passes before the client is ready to connect. If the client were to connect before this, it would get the standard "NO SERVER". Once the socket has been created by the server, the next step is for the client to create its socket (the address and port should match the server). The client then attempts to connect to the listening server. The server then will either accept or reject the connection at which point the socket (i.e. pipe) will be operational and the server and client can do whatever they want.



Code Time - Server

Includes

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

Functions

```
int sockfd = socket(domain, type, protocol)
```

The socket function takes a **domain** (which for our purposes will be PF_INET), a **type** (SOCK_STREAM) and protocol (0 – internet). This is used to create our socket and return a **file descriptor** which we can use for pipey things. We could choose a different type (which is a stand in for protocol). The main two are TCP (SOCK_STREAM) and UDP (SOCK_DGRAM).

```
int error = bind(fd, sockaddr, socklen)
```

Binding is where a lot of the 'work' gets done in setting up a socket on the server side. It takes a **sockaddr** as its second parameter which is a struct containing a lot of useful information like addresses and ports. The first parameter **fd** is just the file descriptor we received when we called socket. The **socklen** is just a **sizeof**(sockaddr)… because C is... C. Like pipes, threads and all things C, bind also returns a success/error code.

```
int listen(fd, backlog)
```

It listens! **fd** is the same file descriptor we have been passing around for a while. The **backlog** is used to declare how many can be in the listening queue. This hasn't been covered before but basically you can imagine a server being inundated with client requests (think Facebook OR Google). The backlog is a way of handling an overabundance of requests. The extra client wannabes are placed in a queue.

```
int accept(fd, sockaddr, socklen_t)
```

Assuming a request has come in we can then accept and begin doing things. The parameters here are the same as the ones used in **bind**.

Structs

```
sockaddr_in
{
      sa_family_t             sin_family;       // Address Family
      in_port_t               sin_port;         // Port in network
      struct in_addr          sin_addr;         // Internet address
}
```

Here we will set **sin_family** to AF_INET. The **sin_addr** and **sin_port** will be set to our address and port name in a bit of a round-about way. To get the **sin_addr** we need to use the function **gethostbyname** and then to get the **sin_port** we need to use **htons**.

Code Time - Client

Includes

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

Functions

```
int connect(fd, sockaddr, socklen_t)
```

Very familiar parameters. We do need to set up the sock address on the client as well.

```
int read(fd, void * buf, size_t count)
```

Here the buffer is where the read data is going to go (i.e. a string). **size_t** is again really just a **sizeof** call on your buffer (or depending on how you define your buffer a separately saved variable).

Webserver

So now we have established the basic logic behind a socket, now it is time to think about how to do something slightly more complicated. While it is nice to be able to connect two computers over the internet, what we have so far is still a bit limited. Ideally, what we would like is to separate two parts of the server. What two parts am I talking about? Well, conceptually, a server has two main things it does. The first is sort of like a receptionist, it takes phone calls. The second is actual work. However, what we have now is a receptionist that both takes phone calls and does actual work. It would be akin to calling the right person first time.

However, this is rarely how things work in practice. Often you will call a business (let us say a bank) without really knowing the phone number of the person you actually need to talk to (say an investment adviser). You just call the bank, and the bank works out what you want and then passes you off to that person. Similarly, a web server probably wants to separate one of the functions 'the receptionist', from the 'doing work' parts of the server.

If the model works this way, the receptionist is immediately available after 'handing off' a call to deal with another potential client. This allows a server to be a big multi-process affair within 'holding up the line'. If we couldn't do this, you could imagine that lots of people would be waiting for one person who hasn't quite chosen the google search they want yet while everyone waits for the 'one queue' google.

What we want to do now is similar to the normal server:

- Make a socket
- Bind the socket
- Listen on the socket

Likewise the client is basically the same:

- Make a socket
- Connect to the server

What we want to next on the server, is instead of engaging in read/write operations with the socket (and hence the client), instead we want to fork, duplicate some file descriptors and the go back to what we were doing before.

Simple Code Example:

```
process_request(fd)
{
      int pid = fork();
      if (pid == -1) return;
      if (pid != 0)
      {
            wait(NULL);
            return;
      }
      dup2(fd, 1);
      close(fd);
      execl("date", "date", NULL);
      exit(1);
}
```

Here we process a request on a file descriptor by simply forking, duping the file descriptors and then calling another process via **exec**. If we set this up right, the socket will now actually be talking to a completely separate process from the one that called it (i.e. the original server). Now the client is directly talking to some other process (be it 'ls' or whatever).

Where to from here?

Well now that we can establish sockets and pass off connections to other processes, we have reached the end of the scope of this course. Obviously, to set up some complex webserver will require a lot more work such as dictating what communications are now sent down this pipe (socket). This could be http (i.e. hypertext), or some other custom made protocols with headers and bodies so that we can interpret what the client wants the server to do. Of course, if you are doing this in C, I feel pity for you (it is laborious). In other languages a lot of this work is abstracted away and then you call libraries which do all the fancy text encoding stuff for you.