6 Application interface: Sockets

Lecturer's note: We jump ahead in our travel up the OSI model up to the interface between the Transport Layer (Layer 4) and Application Layer (Layer 5+) to cover some material that is useful for the lab practicals.

6.1 Goal of this section

At a simplistic level, this section is intended to help set readers up for their first lab practical 😊.

The goal of this section is for readers to understand the sockets interface, both how it works, and some of the background thinking that leads to why it is as it is. We'll also touch on some other relevant concepts for the practical (that are covered in detail in later sections) and cover some useful "gotchas" that might catch you out in the area of sockets.

6.2 Network Applications

Networked applications are necessarily distributed (with all the complications/race conditions/etc. that distribution brings). As such, one key paradigm used over and over again to simplify the interactions between networked applications is "Client-Server".

6.2.1 The Client-Server paradigm

Servers are on all the time, waiting for incoming requests to provide a specific service to a client. They have a well-known advertised address that clients can discover or know before starting communication with the server. For an example of a "server", consider a Website, which has a known address and pages of data that it can provide to you when you connect and ask for them.

Clients are ephemeral. They control initiating the connection to the server – to an address that they have somehow determined already. (The server does not need to know the client's address beforehand, as the client makes the initial request, and the server replies to the client's address.) In the web browsing example above, your browser is acting as a web client. You provide the server's address, when you type in the URL, and your web client connects to the web server and requests the web page information (before closing the connection).

A server typically is able to handle many client connections. While a client may connect to many servers, it typically connects to a single server to provide each specific service.

In decentralised peer-to-peer communications, there is no single "always-on" server. Each peer acts as both client and server, and hosts come and go and change addresses, connecting to other hosts as a client to gather information that they can then relay acting as a server. However, the connections between peers still work as client-server interactions.

6.3 Sockets

The standard service abstraction from the application layer down to the transport layer is done by **sockets**, and note that this is an abstraction that is used regardless of actual networking. This section will cover where sockets comes from, and how its used in networking to provide client and server processes with what they need to connect and communicate.

As a side note: the lab practical is about getting into and understanding the sockets interface by building and using it live. The sockets interface is so ubiquitous that there are a bunch of good libraries out there that handle sockets for almost any language. You're unlikely to need to directly write sockets code ever again.

6.4 Where Sockets come from: Files and Pipes

A process is a running program and its data. Inter-process communications within a single host are determined by the operating system. There are two "obvious" ways to pass data between processes – shared state (where both processes use and edit the same data) and message passing (where a block of data written by one process is passed to another process, with the initial process losing control/ownership of that data). The latter works better for distributability.

6.4.1 Unix Files

Unix files are a way for processes to write and read coherent blocks of data. Processes do not need to know where or how the data is actually being stored or retrieved.

The "Files" interface has 4 simple functions:

```
int open(const char *pathname, int flags, mode_t mode);
```

open prepares a file for reading or writing. The `pathname` gives the OS a description that defines the file uniquely, and the call returns a File Descripter (a small positive integer that we can provide on future calls to uniquely define the file we're talking about). `flags` and `mode` provide additional rules of engagement that go beyond the scope of our interest here. (For example, what permissions we wish to have and or assign to others on a new file (read/write), or if the file does not already exist whether to create it or return an error.)

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

read attempts to read up to `count` bytes of data from the file `fd`, and store them into `buf`. It returns the number of bytes actually read (which may be less than count if the file has less in it.

```
ssize_t write(int fd, const void *buf, size_t count);
```

write attempts to write up to `count` bytes of data from `buf` into the file given by file descriptor `fd.` It returns the number of bytes actually written.

```
int close(int fd);
```

close registers the end of our interest in the file `fd`, and importantly unlinks `fd` so we can no longer use it.

When you look closer, these files are just streams of bytes identified by a number. There is nothing in this concept that requires a file to be a physical space on a disk. Writing to a file is outputting a block of bytes, and reading from a file is taking a block of bytes as input.

6.4.2 Unix Pipes

The next step beyond processes being able to use files to read and write data is to connect these together. Unix provides a method for processes to communicate with each other using Files – Pipes.

```
int pipe(int pipefd[2]);
```

pipe sets up a pair of file descriptors that a process can use. When called, it returns two connected file descriptors in the `pipefd` array . Anything written into `pipefd[1]` can be read from `pipefd[0]` . (Remember that in C, arrays number from 0 😊).

This then allows files to pass any information to each other – which can be used to synchronise, to trigger remote procedure calls, or to request a return of some calculated or stored data.

6.5 Sockets

Extending the "Pipes" idea further, we get the Unix Socket API. This combines pipe-like file writing, with client-server like protocols, and the ability to coordinate addressing across other arbitrary processes.

Sockets are designed to cover all possible types of inter-process message passing. They are connected together after creation, and messages are passed (potentially through a network) to the connected socket.

6.5.1 Unix Socket API

```
int socket(int domain, int type, int protocol);
```

To create a socket in the API you use `socket()`. This returns a file descriptor for your socket. The parameters here are as follows:

- **domain** indicates what kind of thing you want your socket for. Setting this to AF_UNIX gives you sockets that are effectively pipes (see above). Setting this to AF_INET gives you IPv4 sockets – that use IPv4 connectivity to connect across networks.
- **type** gives you options within the domain. Remember, sockets are the interface to the transport layer (which can give services and on top of just raw packet connectivity). SOCK_STREAM gives TCP sockets and SOCK_DGRAM gives UDP sockets. (We'll cover TCP and UDP when we cover the transport layer).
- **protocol** is used to identify the protocol to use. For AF_INET sockets you can set to 0, and socket will use TCP or UDP depending on the type above.

6.6 Addressing for network sockets (IP address and port)

We've already covered IPv4 addresses in some detail earlier in the course – which are used for addressing hosts. As more than one application per host may want a connection, sockets need a way to identify the destination with something beyond just the destination host IP addresses. They use Ports. A port is a 16-bit value, which is a unique identifier for a socket within a host.

Humans write IP address+Port combinations with the ports in decimal after a colon. For example: '172.19.8.52:5060'.

There are several "standard" known ports, defined by IANA. For example :80 is web servers, :22 is for ssh, :25 for smtp mail, etc. Higher numbered ports (above 12k or so) are free for assignment by the operating system – called ephemeral ports.

Coming back to server and client behaviour, servers listen on the well-known port. A client chooses an ephemeral port to connect out of, and attempts to establish a connection with the server (addressing its known port). The server responds from its own ephemeral port (leaving the known port available for future incoming requests).

6.7 Client and Server Sockets

6.7.1 Client code

The client opens a connection to a server and sends and receives data.

Create a socket:

```
int socket(int domain, int type, int protocol);
```

As above, this returns a file descriptor, that we use in future calls.

Connect to a well-known server socket:

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

This returns success (or not!)

The `addr` passed in contains a host (IP) number, and a service (port) number.

In the practical, you can hardcode these, but in the real world, these well come from some further calls such as

```
struct hostent *gethostbyname(const char *name);
struct servent *getservbyname(const char *name, const char *proto);
```

which look up host names and service/protocol name pairs in some database which might (after some bootstrapping) be provided by network services.

Send and receive data:

A connected socket is very like any other UNIX file descriptor, and supports reading and writing (see the File section above).

```
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

The C sockets library provides variants of write and read for use with sockets that we actually use instead of read and write:

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

These have an extra flags argument. These are an integer which is interpreted as a set of bits that control parameters of the service. When set to zero (no flags set), the control parameters are default for the socket type. (You can set these to 0 in the practical and ignore them.)

Close the socket:

Closing a socket dismantles the connection, as well as returning the resources which it is consuming.

```
int close(int fd);
```

6.7.2 Server code

The server listens for incoming connections from clients.

Create a socket:

```
int socket(int domain, int type, int protocol);
```

Bind the socket to a well-known address:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

This is informing the operating system which local address information we wish to use.

Listen for incoming connections:

```
int listen(int sockfd, int backlog);
```

This socket is *not* the socket that will receive data from clients. This is a socket that is waiting to handle incoming connection requests – it's effectively expecting to receive a stream of calls from clients. The queue of waiting clients is allowed to get as long as backlog before clients are turned away. (There are no guarantees of service!)

Accept incoming connection requests:

The server takes requests from calls of connect from a client.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

The value returned from this call is a *new* socket identifier, which is a socket connected to the client that made the corresponding connect. The client address is stored at the addr argument.

If there are no incoming requests, the accept() call blocks waiting for an incoming request (and if there are too many concurrent incoming requests, they may be rejected).

Send and receive data:

The server can send and receive data on the newly returned socket just as the client does.

Then the server can use read, write and close on that new socket to engage in a conversation with the client.

How does the server handle multiple incoming requests? Either it can handle serially (one cashier at a post office) can have a pool of tasks or spawn a new process off each call of **accept()** to hand off incoming connections (multiple cashiers) – probably up to some limit!

6.8 Byte ordering

UNIX files are streams of bytes. Any other data structure has to be flattened into an array of bytes before being written/read. Even integers (bigger than a byte) are data structures. How do you convert these into a set of bytes?

Intel PCs use least significant byte first, so a pointer to an integer of any length can have the same address if the representation has enough trailing zeroes. Sparc chips (now ~defunct) were reversed. If we're going to have different hosts talking to each other, we need consistency across the internet!

The network byte order on the internet is most significant byte first, which is best used with fixed-length representations. Network byte order is used even for things like IP numbers, so conversion is necessary on PCs. There is a suite of fucntions provided to convert. You should always use these when doing the conversion for portability of your code (they are implemented by the OS to perform the correct conversion based on the underlying chipset).

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Note that the names aren't gibberish (or rather they're not *random* gibberish. For all of these h stands for host, n for network, and l & s for long & short respectively – so those names unpack to "host to network long".

6.9 Handy for the practical: Process creation

In UNIX, a new process is created by

```
pid_t fork(void);
```

which (called as part of one process) returns in two processes.

In the calling process, it returns a positive numerical identifier for the new (child) process.

In the child process, it returns zero (so your process can tell whether it is the original parent or a child).

If no process can be created it returns a negative number.

Both processes inherit the file descriptors of the parent, so in your practical, the parent can go on to accept another connection while the child serves this client with the created socket.

6.10 Practical…

You should now have enough information to get you started on the echo-server practical, that involves fleshing out the sockets code for a client and server.