Tutorial 7 - Sockets

Gidon Rosalki

2025-05-14

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Introduction

We are taking a break from purely OS oriented code, and will be discussing sockets, and networking this week. This is a subject of how computers communicate between each other, over a network. The internet is a type of such a network.

2 Networking basics - packets

2.1 Communication protocols

A communication protocol is a system of digital rules for data exchange between computers. Communication systems use well defined formats for exchanging messages, consider them languages. A language is a standardised system for communication between humans, and so too these protocols for between computers. A protocol defines the syntax, semantics, and synchronization of communication. Computer networking protocols implementation is called the protocol stack or network stack.
Communication occurs in several layers, but it will be mostly not discussed in this course.

- Physical: Bytes are streamed, physically, in a cable, or over the air
- Routing: Data is transferred from one entity to another, eventually reaching someone who can communicate directly with whom the data is intended to arrive.
- Management: The responsibility that the data does indeed arrive. Provides interaction with the application through a convenient API, and often supplies compression, encryption, and authentication.
- Application: The application understands the bytes, by converting the bytes into data (for example, into ASCII).

2.2 Packets

If computers can only send messages of a fixed size, we have problems, since messages can vary in size. WE also do not want to send messages that are too long, since these are more prone to errors, or being lost. Instead we have defined a unit of data called a **packet**, which is carried over a network. Packets have a minimum, and maximum size, and if we have a message that is longer than the maximum, we split it into many packets.
A packet contains:

- Header: This contains the source and destination addresses, often the packet identification number, and other protocol data
- Payload / Data: The information we want to send
- Trailer (sometimes): Comes at the end, after the data.

Packetisation is the system for splitting a larger amount of data into many packets. It is responsible for adding headers (and trailers) to packets, breaking the message into packets, and sending the packets / receiving incoming packets.
Errors can occur, for example the packet can be lost, and the transmission can be corrupted, and then the wrong bytes are received. For example, we could send `001`, and then instead `000` is received. Often, the communication protocol is also responsible for correct transmission. It ensures that data arrives, and that data is received in the correct order, since data can arrive in an arbitrary order. If data has arrived, it ensures that the data is accurate (this may include error correction, or a request to resend the data). It is important to note though, that some protocols do not take responsibility for some / all of these steps.

2.3 Routing

Since not all computers are directly connected one to another, we need routing to pass packets between intermediate stations. Another program is responsible for routing (sending to those intermediate stations, and eventually to the destination).

2.4 Protocol stack TCP/IP

The sender adds the headers to the packet, and since this is only used for transmission, the receiver drops it. We can for example add a header of a checksum for the packet, as an Error Correcting Code (ECC), so if the packet is corrupted this can be detected (and potentially fixed depending on the algorithm used.
The 4 layer model protocol stack (TCP/IP) is the model in which communication is performed in the following four layers:

- Application: The way the application "understands" the data (HTTPS, FTP, DNS, etc.)
- Transport: Makes sure that the communication is done smoothly, over protocols like TCP and UDP.
- Network/internet: The transmission of the data itself, routing, through protocols like IP
- Link/Physical: Actual bytes transfer (WiFi, Ethernet)

3 TCP and UDP

3.1 The transport layer

The network layer (one layer below the transport layer) is responsible for packet transmission from computer, to computer. Data is sent, but communication is *unreliable*. The transport layer is responsible for application- to-application transmission of packets. It may (but not always) supply a reliable, end to end communication for applications. This is how we can have multiple clients on a single server. The two main protocols on this layer are TCP and UDP.
However, many applications may want to communicate at the same time. How do we know to which application received data is intended? By using **ports**. A port is a unique number, identifying a connection end point. The packet contains incoming and outgoing port numbers. This is how the OS know with what application the data is associated. Ports are numbers from 1 to 65535. As we mentioned before, protocols of the application layer use the transport layer protocols. Here are some conventions:

- FTP: 20 and 21
- SSH: 22
- SMTP: 25
- HTTP: 80
- HTTPS: 443
- DNS: 53

3.2 UDP - User Datagram Protocol

UDP is the simplest transport layer protocol. The header contains the incoming, and outgoing ports, the length of the message, and a checksum (a simple integrity method). This is deeply unreliable, packets may be duplicated, lost, and arrive out of order. Consider UDP to be shouting into the void. If someone listens, great! You don't know that though. Additionally, they don't know if they heard you right.

3.3 TCP - Transmission Control Protocol

This is a reliable stream transport. It is more connection oriented, a "handshake" is mandatory before communication begins, where the sender and receiver agree on details. This takes care of lost / duplicated packets, packet ordering, long delays, and data buffering.

3.4 TLS - Transport Layer Security

None of what was discussed so far was secured. There are however protocols for securing transportation of data. For example, TLS. These protocols sit above a transport layer protocol. The TLS packet contains the data of the user (encrypted, or authenticated, for example). The whole TLS packet is the data of a TCP packet, and the whole TCP packet is the data of a IPv4 packet. TLS requires a handshake, much like TCP.

Property	UDP	TCP
Reliable	No	Yes
Connection type	Connectionless	Connection oriented
Flow control	No	yes
Latency	Low	High
Applications	VOIP, Games, video, etc.	HTTP, HTTPS, FTP, SMTP, Telnet, SSH

Table 1: UDP vs TCP

3.5 QUIC

We often use both TLS and TCP (for example, the internet), but that requires two handshakes to initiate a conversation (and packet overhead). A relatively new protocol named QUIC replaces the combination of TLS+TCP. It relies on UDP (to avoid the TCP handshake), but supplies both TLS security, and TCP integrity, with only a single handshake.

4 Sockets

4.1 Berkeley Sockets API

The sockets API is a set of system calls, which provide convenient communication for processes. A socket is a door / gate for communication. A socket is "a host-local application-created OS-controlled interface into which application process can both send and receive messages to/from another application process". It follows a client/server paradigm, and supports both UDP and TCP. It effectively abstracts away dealing with TCP, and the exact implementations to the OS, and provides a simpler method to access the data.

4.1.1 Client/Server model

There's a single server, and multiple clients (consider Moodle, and all the students). The routine is as follows:

- A client initiates contact with a server
- The server is "ready" (also known as listening), for such new communications
- When a server is contacted by a client, a separate socket is created, to communicate with a specific client. The client is unaware of this socket, and the server continues to listen for new clients

4.1.2 Client/Server model - TCP

The server creates a socket, binds, and listens. The client creates a socket, and connects to the host IP, on port `x`. The server, which was waiting fro such a connection, accepts. At this point, if this is TCP, a handshake is performed. The client sends a request, which the server reads, and sends responses (this section loops, until the message is finished). Then, everyone closes their sockets. We read and write to sockets, much like files, but the OS handles the fact that these are subtly different.

4.1.3 Client/Server model - UDP

The server creates a socket, of port `x`, and the client creates a socket, sends a request to the server, which the server (probably) reads, and may then respond (this section loops). The client will read a reply, and then the sockets are closed.

4.2 Address

Let us consider the code for implementing this. We need to begin by specifying an address. The address will be from an address family (IPv4 or IPv6). We also need any additional data. To do this, we have an address struct:

```
struct sockaddr
{
    unsigned short sa_family;
    char          sa_data[];
};
```

`sa_family` is usually `AF_INET` or `AF_INET6`, and the `sa_data` contains relevant data, such as the port. Using this address struct is complicated, in what place do we put the port? To make life easier, we have a specialised struct for IPv4 (very common) addresses, and for IPv6, we sue `sockaddr_in6`.

```
struct sockaddr_in
{
    short            sin_family;
    unsigned short int sin_port;
    struct in_addr   sin_addr;
    unsigned char    sin_zero[8];
};

struct in_addr
{
    uint32_t         s_addr;
};
```

A pointer to `struct sockaddr_in` can (and should be, when used) cast into `struct sockaddr`. The field `sin_zero` is used for padding, and must be set to all 0s, using `memset`. Additionally, both `sin_port`, and `sin_addr` must be in **Network Byte Order**.

4.2.1 Little/Big endian

These are 2 ways of writing numbers, and the difference is where the most significant bit is located. For big-endian, more significant bytes are located in lower memory locations (consider this, starting from the biggest number first). In little endian, the most significant bytes are located in higher memory locations (consider this, starting from the smallest number first). For example, if we take the number 6, this would be `110` in big-endian, and `011` in little endian.
Big-endian is the **only** approach in networking, whereas most computers are little endian (but not all). Therefore, we need to convert our host address into networking address order. However, you must be careful to ensure that the address and ports are in order before using networking functions to achieve this.
Since there are 2 types that we want to convert, `short` (2 bytes), and `long` (4 bytes), we have the functions:

- `htons` - "Host to network short"
- `htonl` - "Host to network long"
- `ntohs` - "Network to host short"
- `ntohl` - "Network to host long"

IPv4 addresses should be converted from the "human friendly" notation, into bytes. We have the function `inte_aton` for this purpose. **WARNING**: This function returns a 0 for failure, and **not** success.

4.2.2 DNS

We do not know the IPv4 addresses of websites off the top of our heads (and if you do, what is *wrong* with you? They change all the time!). To do ths, we instead use the Domain Name Service (DNS). This protocol allows a translation between a "human readable" address (for example, google.com) into an IP address (for example, 142.251.209.14). There's a function named getaddrinfo that can run a DNS query and return the result.

4.3 Sockets programming (TCP)

First, we must create a socket to listen for connections (in TCP). This is done using the `socket` system call.

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

The `domain` parameter determines the communication protocol family. We use `AF_INET` for IPv4 and `AF_INET6` for IPv6. `type` determines the socket type, for example `SOCK_STREAM` for TCP, and `SOCK_DGRAM` for UDP. The parameter `protocol` is almost always 0. It is the default protocol for the socket type.

The socket system call returns an integer. A negative number is considered as an error, and a non-negative value is a number called "file descriptor". The file descriptor is an ID given to the user, and it is used to identify the socket when we ask the OS for operations. This is analogous to opening a file.
We now have a socket, but it is not yet connected to any port or address. The next step is called binding, and is done by the server. To achieve this, we use the bind system call:

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

which gets the socket file descriptor, the address struct value, and the length of the address struct. It ”identifies” the socket with an address.

A machine can be a part of multiple networks, these are called interfaces. For example, consider the loopback interface (sending data to oneself), ethernet, WLAN, and so on. In each network, it might have multiple IP addresses, but you cannot have the same IP on different interfaces. In binding, we supply an address, and a port. This way we can determine who can address us, and how (what address and port).

Now we are bound to an address, it is time to wait for requests. The TCP stack can queue incoming connection requests, and if we're busy handling a connection, then another incoming request waits until we can deal with it. We have the function

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

which gets the socket file descriptor, and the maximum number of allowed clients that will be waiting to be accepted (see later).

```
int
start_listening (unsigned short portnum)
{
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY; // No binding to a specific interface, instead listening to ALL
    addr.sin_port = htons (portnum);
    memset (addr.sin_zero, 0, sizeof (addr.sin_zero));

    int welcome_fd = socket (AF_INET, SOCK_STREAM, 0);
    bind (welcome_fd, (struct sockaddr *)&addr, sizeof (addr));
    listen (welcome_fd, 3);
    return welcome_fd;
}
```

It is of utmost importance to check the return values!

Once we have started to listen, we wait for calls to that socket using the `accept` system call. This is analogous to picking up the telephone when it rings. `accept` returns a new socket which is connected to the caller. This allows a conversation between the two, and is transparent to the client. The server may open a different thread, to communicate with the client, and keep accepting new requests.

In response, we call

```
int accept(int sockfd, struct sockaddr *cli_addr, socklen_t *cli_addrlen);
```

This gets the socket ID, with which `listen` was called, and returns a new file descriptor, of a socket for communicating with the client. This sets 2 new parameters:

- `cli_addr` - The address of the client
- `cli_addrlen` - The length of the address struct that was returned

```
int
get_connection (int s)
{
    int s = start_listening (PORT); /* earlier
                                */
    int t;                      /* socket of connection */
    if ((t = accept (s, NULL, NULL)) < 0)
        return -1;
    return t;
}
```

So we have now initialised the server, and it awaits a call, but what about the client? The client starts by ”having a phone” before it calls someone, and we use the socket system call, as before. Now the client connects to the server socket. It supplies the socket file descriptor, and the address struct which should be the same as the server's.

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

The connection is now established! After the client calls `connect`, the server accepts the connection and creates a new socket for communication between the two. Then, the `read` and `write` system calls are used to write and read data, just like reading and writing from files.

4.3.1 Read and write

To read from a file descriptor into a buffer we use

```
ssize_t read(int fd, void *buff, size_t bytes);
```

which reads **at most** `bytes` bytes into the buffer given in `buff`, from `fd`. It returns the number of bytes actually read, or negative values in errors, and blocks until at least 1 byte is read. It is important to note that it does not necessarily read all the bytes! You have to loop until you read everything you want.
To write to a file descriptor, given a buffer, we use the following:

```
ssize_t write(int fd, const void *buff, size_t bytes);
```

For example:

```
int
read_data (int s, char *buf, int n)
{
    int bcount; /* counts bytes read */
    int br;   /* bytes read this pass */
    bcount = 0;
    br = 0;
    while (bcount < n)
        { /* loop until full buffer */
            br = read (s, buf, n - bcount);
            if (br > 0)
                {
                    bcount += br;
                    buf += br;
                }
            if (br < 1)
                {
                    return -1;
                }
        }
    return bcount;
}
```

In actuality, when using sockets, two different system calls are common:

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

`flags` allow advanced options: For example, non blocking receive, raising a signal if the peer closed the connection, and more. They are equivalent to write and read (respectively), if `flags=0`.