Operating Systems and Architecture (CO527) Spring 2016

Using Networks

Fred Barnes and Radu Grigore

Socket Programming

A connected socket is similar to an open file. A connected socket is identified by an integer, just like an open file is identified by a file descriptor which is an integer. We can use a connected socket in most places where we can use a file descriptor. In particular, we can use the functions read, write, and select.

To obtain a connected socket we must establish a network connection. Establishing a network connection is asymmetric: the client initiates the connection, while the server waits for connections.

Here is what the client must do:

```
1 int s = socket(AF_INET, SOCK_STREAM, 0);
2 connect(s, (struct sockaddr *) &address, sizeof address);
```

Line 1 creates a socket; line 2 connects to a server. We will see later how address is set. After line 2, the client can use the functions read and write as if s were a file descriptor.

Here is what the server must do:

```
1 int listening_socket = socket(AF_INET, SOCK_STREAM, 0);
2 bind(listening_socket, (struct sockaddr *) &address, sizeof address);
3 listen(listening_socket, connection_queue_size);
4 int connected_socket = accept(listening_socket, NULL, NULL);
```

Line 1 creates a socket; line 2 binds the socket to a local address; line 3 prepares this socket to accept connections; line 4 accepts a connection. In line 1, we choose which type of socket to use. In line 2, we choose where to wait for connections: a computer may have multiple network interfaces, and even single network interfaces have several ports. In line 3, we choose how many clients can queue waiting for a connection; if more clients try to connect, they will be summarily refused. In line 4, we obtain a connected socket, which is *different* from the listening socket: we use connected_socket as if it were a file, to communicate with the client; we use listening_socket to accept further connections.

At this point, both the client and the server can communicate using the functions read and write, as if they would be interacting with an open file. See the accompanying miniclient.c and miniserver.c.

IP, TCP, UDP

Both the client and the server create sockets by calling

```
1 socket(AF_INET, SOCK_STREAM, 0)
```

The result is what POSIX calls an 'internet domain byte-stream socket for use with IPv4 addresses'. In practice, it means that we get to use the TCP/IP protocol, version 4.

IP (**i**nternet **p**rotocol) is the network protocol used by Internet. It is responsible for delivering packets from one address to another. Each network card connected to the network has an IP address. In version 4, an address is 32 bits long; in version 6, an address is 128 bits long.[1] To be more precise, each IP address identifies a *network interface*, which may be a real network card or may be a special, virtual network card. For example, each Linux computer has a so-called *local loopback* network interface, corresponding to the IPv4 address 127.0.0.1. On top of IP come several protocols, including TCP and UDP.

[1] We are transitioning from IPv4 to IPv6 because there are more than 2^{32} network cards in the Internet. The only reason the Internet still works is that only some IPv4 addresses are visible globally. This is because the network has a hierarchical structure. If you want more details, you'll have to take a module about networks.

TCP (**t**ransport **c**ontrol **p**rotocol), the protocol used in the example above, gives the illusion of a reliable stream of bytes. The user does not know (1) that the stream of bytes is split up into packets, (2) that packets may overtake each other but TCP takes care of reordering them, and (3) that packets may get lost in transit but TCP takes care to retransmit those. UDP (**u**ser **d**atagram **p**rotocol) does not try so hard as TCP: (1) it does not hide that data is split into chunks called datagrams, (2) it does not reorder datagrams that overtake each-other, and (3) it does not retransmit lost or corrupted datagrams.

Both TCP and UDP add the notion of a port on top of that of IP address. Each IP address gets 2^{16} ports, from 0 to $2^{16}-1$. The purpose of ports is to allow several servers to use the same network interface at the same time. For example, port 80 is traditionally reserved for http servers, while port 22 is traditionally reserved for ssh servers.

If TCP does so much more than UDP, why would you ever used UDP? The standard use-case for UDP is transmitting audio or video data. When you talk on Skype, it is not a big problem is the sound drops for 20 ms — you might not notice even. However, if the sound is delayed by 500 ms, then the perceived quality of the call goes down sharply. That is, for audio, it is more important that data arrives at its destination quickly, and less important that all data arrives.

From the point of view of programming, the main difference between using TCP and using UDP, is that UDP does not use *connected* sockets. This means (1) that there is no special connection phase (no connect/accept), and (2) that you specify the destination of a message every time you send one. Symmetrically, you don't find out the origin of some data by looking on which connection it comes, but by looking at the message itself. In turn, this means that you do not use read and write to communicate. Instead, you use sendto and recvfrom: the function sendto has extra arguments to specify the destination; the function recvfrom has extra arguments to specify the source. (The above contains a small lie. You *can* in fact use connect and write. The effect of connect is local: it simply sets a default destination, to be used with subsequent calls to write. So, no connection actually takes place.)

IP Addresses

To connect or bind a socket, we need an address. If we know the 32 bit address, then we need to put it into a struct sockaddr. For example, miniclient.c does the following:

```
struct sockaddr_in address = {
  .sin_family = AF_INET,
  .sin_addr = htonl(INADDR_LOOPBACK)
  .sin_port = htons(12345),
};
```

The sin_family member says this is an IPv4 address; the sin_addr member says we use the special loopback address 127.0.0.1, and the sin_port member chooses the port 12345. The functions htonl and htons adjust the byte order from the convention of the host to that of the network. The convention of the host is often little-endian, while the convention of the network is big-endian. In case htonX is called on a computer that happens to use the same convention as the network, then htonX just returns its argument.

But, more often, we remember Internet server names rather than addresses. For example, we prefer to remember www.google.com, rather than 216.58.213.100. The Internet contains servers whose goal is to convert between names and addresses.[2] POSIX specifies a simple way to access this service: the function getaddrinfo. Here is a typical call:

```
getaddrinfo("www.google.com", "80", &hints, &candidates);
```

Both hints and candidates are pointers to struct addrinfo. In hints we select, for example, whether we want TCP or UDP. From candidates we read the address to use with connect or bind.

See the accompanying dns.c for an example of how to use getaddrinfo.

[2] Your computer must know the IP address of a DNS (**d**omain **n**ame **s**ystem) server if you want to use names like www.google.com. The IP address of a DNS server is acquired via a protocol called DHCP (**d**ynamic **h**ost **c**onfiguration **p**rotocol). Again, you'll have to take a networks course if you want to know more.

The addresses obtained via getaddrinfo are so-called *unicast* addresses: they refer to a single network interface. There exist also *multicast* addresses, which refer to some set of network interfaces, and a *broadcast* address (255.255.255.255), which refers to all network interfaces. Multicast is intended to be used for distributing audio and video. The basic idea is to send less data over the network by taking advantage of having multiple viewers. For example, if 10 students from University of Kent want to watch live some event at the Olympic games in Rio, then surely it's enough to send *once* the video data from Brazil to UK, and split the stream into 10 copies once UK is reached. But, multicast is not used much: live events are rare, and joining/leaving multicast groups involves some overhead. The function of multicast (exploit geographic location to reduce network traffic) is nowadays performed by CDNs (**c**ontent **d**elivery **n**etworks). Broadcast, on the other hand, is sometimes used. However, broadcast does not mean that a message reaches the whole of Internet: broadcast packets are not relayed by routers. So, broadcast really means 'all network interfaces in the *local* network'.

In Linux, only unicast addresses can be used with TCP.

Network Administration

Before we move on to some implementation details, let us look at several tools for network administration: ss, ip, tc. These tools are not part of POSIX — they are Linux specific. Our goal is not to become adept at network administration. Rather, we'll use these tools to start peeking under the hood, at the implementation of TCP/IP.

You can use ss to inspect the sockets on your system, which were created and not yet closed. For example, ss -t lists the connected **T**CP sockets.

You can use ip to inspect and to change routing tables.

```
rg@rg-2016:network$ ip route show dev wlan0
default via 192.168.0.1  proto static
192.168.0.0/24  proto kernel  scope link  src 192.168.0.4  metric 9
```

Above, we see two entries in the routing table. The first one applies when no other does (default), and it says that packages should be routed to 192.168.0.1. (That's the address of a WiFi router, by the way.0 The second one applies for destinations that match 192.168.0.0/24; that is, for destinations that are the same as 192.168.0.0 when we ignore the last 24 bits of the address. In this second entry there is no routing involved (via is missing) which means that the packet should be sent directly. We can also inspect a table that caches our neighbours — those to whom we can send packets directly:

```
rg@rg-2016:notes$ ip neighbour
192.168.0.9 dev wlan0 lladdr 3c:a9:f4:4a:f4:20 STALE
192.168.0.1 dev wlan0 lladdr 90:21:06:0a:05:c0 REACHABLE
```

We see two entries. One is STALE, which means that address was not seen for awhile. The other is reachable, and is the address of the WiFi router. The lladdr gives the link-layer address, also called physical address, also called MAC address. This table that tracks IP–MAC correspondence is sometimes called *ARP table*, because it is built using ARP (**a**ddress **r**esolution **p**rotocol).

Finally, you can use tc to configure traffic control. There are four kinds of traffic control:

1. *Traffic shaping* refers to delaying the transmission of outgoing packets. Without shaping, traffic tends to be quite bursty: long periods of inactivity interspersed with short periods of intense traffic. This bursty nature of the traffic causes all kinds of problems. For example, if the intense traffic exceeds network capacity, this will lead to lost packets, which need to be retransmitted, hence causing even more traffic.

2. *Traffic policing* is applied to incoming traffic: If the rate exceeds a threshold then a special action is taken. That action could be dropping the packet, or it could be reclassifying it into a lower priority queue.

3. *Traffic scheduling* refers to reordering packets on transmission. Without scheduling, a download could completely ruin a Skype call. With scheduling, the kernel could make sure that high-priority but low-bandwidth applications don't get squeezed out by low-priority but high-bandwidth applications.

4. *Traffic dropping* may happen both for incoming and for outgoing packets. The goal is in general to avoid network congestion.

TODO: add qdisc and filters

Exercises

1. ► What is a *socket*?
2. Look at Java's implementation of java.net.Socket and java.net.DatagramSocket.
3. To see all the network interfaces on your system, run 'ip link'.
4. To see which servers and on which ports are running on your computer, run 'nmap 127.0.0.1'. Figure out how to do something similar by using ss.
5. ► Give a list of four websites that are likely to use UDP. Connect to them, and check whether they do indeed start UDP connections. [Hint: Use ss.]
6. For an example of how to use UDP, look at 'man getaddrinfo'.
7. ► What is the difference between multicast and broadcast?
8. Look at 'man ss', 'man 8 ip', and 'man tc'.