Lecture 13 - TCP/IP Networking

(last updated: 11/03/22)

Networking allows programs on different computers to communicate with one another. Networking usually take place over a "stack" of protocols, with each layer of the stack handling some aspect of communication. There are multiple ways to characterize the layers and responsibilities in a protocol stack; here is one such characterization:

Layer	Purpose	Examples
Application	interpret transported data	HTTP, SSH
Transport	manage flow of network packets	TCP, UDP
Network	route packets over a network of links	IP, ICMP
Link	send packets over a physical connection	Ethernet
Physical	provide a physical connection	IEEE 802.3ab

In this course, we will treat the implementation of everything under and including the transport layer as a black box. We will interact with the transport and network layers from the command line using netcat, and from C using the POSIX sockets API (application programming interface).

Be forewarned that there is quite a lot of boilerplate (repetitive) code that you need to write when using the POSIX sockets API. Rather than explain that code in detail, this lecture note will outline the steps taken to establish a TCP connection, and focus on context and background useful for navigating the sockets API. Make sure to study the accompanying code examples for more details about how the API should be used.

Also note that in these lecture notes, when I write "IP" I am referring to IPv4 (Internet Protocol, version 4) in particular; IPv6 addresses are structured slightly differently and are beyond the scope of this course.

TCP/IP connections

TCP/IP networking provides a reliable, two-way connection between computers (usually called "hosts" or "peers" in the context of networking). Here, "reliable" means that, as long as two hosts are connected, bytes sent between them are never dropped, duplicated, corrupted, or reordered (as might happen with other networking protocols); "two-way" means that both hosts can send or receive data.

Hosts are identified by an IP address, a 4-byte (32-bit) integer usually written in "dotted-quad" notation, where four decimal numbers, each representing a byte, are separated by periods, e.g., 34.145.159.110 and 127.0.0.1.

Hosts are also identified via hostnames, including domain names such as clac.cs.columbia.edu. Domain names are resolved to IP addresses using DNS; e.g., clac.cs.columbia.edu resolves to 34.145.159.110.

Different programs running on the same hosts are distinguished by port numbers, a 2-byte (16-bit) integer. Well-known applications typically use fixed port numbers, e.g., port 80 for HTTP web servers, port 22 for SSH servers. Port numbers below 1024 are typically reserved for well-known applications, so when experimenting, you should choose port numbers greater than 1024.

TCP/IP follows a client-server model, where one host acts a server and the other as a client. At a high level, a TCP connection is established as follows:

- a server program "listens" on a certain port number
- a client program "connects" to the server using the server's IP address and the port number the server program is listening on
- from that point onward, a bidirectional, two-way connection is established

netcat: TCP/IP swiss army knife

nc ("netcat") is a handy command-line tool for TCP/IP networking that exposes TCP connections to stdin/stdout. nc is convenient for quickly making TCP connections and is perfect for experimenting with networking.

You can use nc as either the server or the client in a TCP connection. For clarity, I will write "server$" as the prompt for the server machine and "client$" for the client machine (though they may be the same machine).

To use nc as a server, use the -l flag (short for "listen"):

```
    server$ nc -l <port-number>
```

To use nc as a client, simply specify the hostname and port number:

```
    client$ nc <hostname> <port-number>
```

If you connect two instances of nc, what you type on one end should be visible on the other.

Some other useful flags:

- -N: close the network connection upon EOF.
- -C: translate newlines ("\n", also known as LF) to carriage returns followed by newlines ("\r\n", also known as CRLF); useful for protocols that expect lines ending with CRLF.
- -k: keep the server alive after a connection is completed and listen for subsequent connections.

Sockets API

On POSIX systems, networking capabilities are made available to programs in the form of internet sockets. In particular, stream sockets represent an endpoint for reliable, bidirectional connections such as TCP connections. Stream sockets are associated with file descriptors that you can use with I/O system calls such as read(), write(), and close(). (There are also other kinds of sockets used for other kinds of connections, beyond the scope of this course.)

You can create a socket and obtain a file descriptor for it using the socket() system call (arguments omitted for brevity; see attached examples for details):

```
    int fd = socket(...);
```

fd is a file descriptor referring to the newly created socket, though this socket is not yet associated with any TCP connection. How we connect that socket depends on whether we are using it as a TCP client or server.

A TCP client forms a socket connection with a server by connect()ing the socket file descriptor to the server address:

```
    int fd = socket(...);

    connect(fd, ... /* server address */);

    // Communicate with the server by read()ing from and write()ing to fd.

    close(fd);
```

A TCP server is a little more complicated. We first need to bind() the socket to a server address and port, and then tell it to listen() for incoming connections; finally, we can accept() incoming connections:

```
    int serv_fd = socket(...);

    bind(serv_fd, ... /* server address */);

    listen(serv_fd, ... /* max pending connections */);

    for (;;) {
        int clnt_fd = accept(serv_fd, ...);

        // Communicate with the client by read()ing from and write()ing to
        // clnt_fd, NOT serv_fd.

        close(clnt_fd);
    }
```

Several things to note here:

- accept() will block until a client connects to the server.

- accept() returns the file descriptor for a NEW socket (clnt_fd) for communicating with the connected client.

- We can terminate the TCP connection with the client by close()ing clnt_fd.

- The server accept()s new connections in an infinite loop because that’s how servers typically operate; we don’t need to loop if we only care about connecting with a single client.

Socket address structures and network byte order

The connect() and bind() system calls require you to specify the server’s internet address and port using a socket address structure, defined as follows:

```
    struct sockaddr_in {
        sa_family_t    sin_family;          // address family: AF_INET
        uint16_t       sin_port;            // port in network byte order
        struct in_addr sin_addr;            // internet address
    };
    struct in_addr {
        uint32_t       s_addr;              // address in network byte order
    };
```

Here, "network byte order" refers to the endianness of the 4-byte address and 2-byte port number. Network byte order is always big-endian, and contrasts "host byte order" which differs from machine to machine. The byte order of the address and port number must be explicitly specified so that it is consistent across networks where hosts do not all have the same endianness. The following functions convert 2- and 4-byte integers between network and host byte order:

```
    uint16_t htons(uint16_t host);          // "host-to-network, short"
    uint16_t ntohs(uint16_t net);           // "network-to-host, short"

    uint32_t htonl(uint32_t host);          // "host-to-network, long"
    uint32_t ntohl(uint32_t net);           // "network-to-host, long"
```

On big-endian hosts, these functions do nothing; on little-endian hosts, these functions convert the byte order accordingly.

We do not normally need to populate socket address structures ourselves, since nowadays helper functions will do so for us (e.g., getaddrinfo()), but it is still illustrative to see how one might do so (as one would "in the old days"):

```
    // Use 3157 as an example port number:
    uint16_t ip_port = 3157;

    // Construct 34.145.159.110 as a 4-byte integer:
    uint32_t ip_addr = 34 << 24 | 145 << 16 | 159 << 8 | 110;

    struct sockaddr_in addr;                // Define socket address struct

    addr.sin_family = AF_INET;              // Set internet address family
    addr.sin_port = htons(ip_port);         // Set port number
    addr.sin_addr.s_addr = htonl(ip_addr);  // Set IP address
```

Socket address polymorphism in C (optional)

Astute readers may have noticed from the man pages that the type signature of connect() does not take internet socket addresses of type struct sockaddr_in:

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

Instead, the second parameter accepts pointers to struct sockaddr! You will find a similar type mismatch with the bind() system call.

The sockaddr structure exists because sockets can be used to form connections other than IPv4 connections (e.g., IPv6), which use different kinds of socket address structures. Rather than make a different connect() and bind() system call for each kind of socket connection, POSIX exposes a single "polymorphic" interface that supports all socket address types, whose pointers are to be cast to the generic sockaddr structure:

```
    struct sockaddr {
        sa_family_t sa_family;      // address family
        char        sa_data[14];    // interpretation depends on sa_family
    };
```

The "address family" (i.e., sa_family) of all specific socket address structures (e.g., struct sockaddr_in, whose address family is AF_INET) is recorded in their first field, and is used by connect() and bind() to distinguish socket address types from one another (e.g., if they see addr->sa_family == AF_INET, they will cast the pointer type to struct sockaddr_in *).

Thus, when we manually build an internet socket address structure, we cast its pointer to struct sockaddr * before passing it to connect() or bind(), e.g.,:

```
    // Same ip_port and ip_addr as before

    struct sockaddr_in addr;                 // Define socket address struct
    memset(&addr, 0, sizeof(addr));          // Zero-initialize addr’s bytes

    addr.sin_family = AF_INET;               // Set internet address family
    addr.sin_port = htons(ip_port);          // Set port number
    addr.sin_addr.s_addr = htonl(ip_addr);   // Set IP address

    // Obtain stream socket for TCP/IP connections, and connect() it to addr:
    int fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    connect(fd, (struct sockaddr *) &addr, sizeof(addr));
```

As an extra precaution against memory errors, we zero out the bytes allocated for addr using memset() to ensure that any struct padding bytes are also zero-initialized (since those bytes aren’t accessible via struct fields).

Obtaining socket addresses with getaddrinfo()

Internet addresses are typically written in human-readable forms: IP addresses are usually written in dotted quad notation (e.g., 34.145.159.110), or given as domain names (e.g., clac.cs.columbia.edu) that must be resolved using DNS.
To convert these human-readable addresses to the socket address structures that connect() and bind() expect, we can use the getaddrinfo() function provided by the POSIX API:

```
    int getaddrinfo(const char *node,              // human-readable address
                    const char *service,           // human-readable port
                    const struct addrinfo *hints, // hints to limit the lookup
                    struct addrinfo **res);        // res will point to result
```

getaddrinfo()’s interface is complicated by the fact that it’s sometimes possible to obtain multiple socket addresses from the same human-readable address; for instance, it’s possible for the same domain name to resolve to multiple IP addresses (perhaps some are IPv4 addresses and others are IPv6 addresses). Thus, we can specify some hints to limit the kinds of address(es) it will obtain for us.

Here is what the above connect() example looks like, using getaddrinfo():

```
    struct addrinfo hints;

    memset(&hints, 0, sizeof(hints));   // Zero-initialize all fields by default

    hints.ai_family = AF_INET;          // IPv4 internet address family
    hints.ai_socktype = SOCK_STREAM;    // Stream socket
    hints.ai_protocol = IPPROTO_TCP;    // TCP/IP protocol
    struct addrinfo *res;               // Where to report results

    getaddrinfo("34.145.159.110", "3157", &hints, &res);

    // res now points to linked list of struct addrinfo containing address info;
    // the first result is sufficient for our purposes, but you can access the
    // other results by following the .ai_next field of each struct addrinfo.

    int fd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    //       ^should be same as socket(AF_INET, SOCK_STREAM, IPPROTO_TCP),
    //        since we explicitly specified those fields in our hints

    // getaddrinfo() populates the socket address structure for us
    connect(fd, res->ai_addr, res->ai_addrlen);

    // We're done with res, so we can free() it.
    freeaddrinfo(res);
```

Note that when we call getaddrinfo(), we specify the IP address and port as strings, rather than as numbers. We can also specify a domain name, and getaddrinfo() will perform the DNS lookup for us:

```
    getaddrinfo("clac.cs.columbia.edu", "3157", &hints, &res);
```

To obtain an address for the server side to bind() to, you will need to pass a NULL pointer as the first parameter (instead of a human-readable address string), and additionally specify the AI_PASSIVE flag in hints.ai_flags; make sure to study the accompanying code examples to see how this is done.

Sockets I/O with file descriptors and FILE pointers

Once you've established a TCP/IP connection on a socket, you can communicate with your peer using the socket file descriptor with the write() and read() system calls, e.g.:

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

Since sockets encapsulate more complicated behaviors than regular files, POSIX also provides the send() and recv() system calls, which work like write() and read(), except they take an additional flags argument:

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

All of these system calls can be used interchangeably with TCP stream sockets; write() and read() are equivalent to send() and recv(), with the flag set to 0. We can also pass non-zero flags to send() and recv() to customize their behavior with respect to the underlying socket connection. For example, the MSG_WAITALL flag tells recv() to block until the entire buffer is filled (or until the connection is interrupted or disconnected); the MSG_DONTWAIT tells recv() not to block, and only read from buffered TCP packets the kernel has already received.

However, file descriptors and direct system calls can be more cumbersome to use than their FILE pointer counterparts. For instance, there isn't any system call equivalent to fgets() or fprintf(). To overcome this limitation, we can "wrap" a file descriptor using fdopen(), which will give us a FILE pointer we can use with those functions:

```
    FILE *sock_fp = fdopen(sock_fd, "wb");

    fprintf(sock_fp, "Sending a formatted number: %4d\n", 42);

    // ...

    fclose(sock_fp);
```

When we fclose() the socket FILE pointer, fclose() will close() the underlying socket file descriptor for us, so there's no need to do so separately.

Note that FILE pointers are block-buffered by default, so you may need to call fflush() to ensure any buffered output is actually sent through the socket connection to your peer (or just turn off buffering altogether using setbuf()).

One wrinkle with using FILE pointers to encapsulate socket file descriptors is that FILE pointers aren't really designed for duplex file streams. In particular, mixing read and write operations with the same FILE pointer can be problematic in the presence of output buffering, so it's best to create separate FILE pointers, one for reading, and one for writing:

```
    FILE *sock_fpw = fdopen(sock_fd, "wb");       // write-only FILE pointer
    FILE *sock_fpr = fdopen(dup(sock_fd), "rb");  // read-only FILE pointer

    // ...

    fclose(sock_fpr);
    fclose(sock_fpw);
```

Note that we first duplicate the socket file descriptor by calling dup() on sock_fd; this creates a new file descriptor referring to the same stream socket, which we give to fdopen(). Since we now have two file descriptors, we need to make sure we close() both of them by calling fclose() on their wrapping FILE pointers.