Networks & Sockets

CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

March 20, 2023

Lesson Plan

1. Communication & Networks
    - UDP vs. TCP
    - IP addresses and ports.
    - Endianess
2. Sockets
    - Server-Client Model

Acknowledgements

Some slides were adapted from Andi Bergen's; others are original.

Section 1

Communication & Networks

Communication: Humans vs. Computers

Communication can take many forms...

Human Communication

Human communication is governed by syntactic rules: vocabulary, sentence structure, spelling, grammar...which form semantics and context.

`How would you like it if a l l th elect ur e sli des we r e fo r ma ttt ed like thiiiiiis ?!.$`

Computer Communication

Computer communication is also governed by rules:

- *Encoding* of information, e.g., text, signed/unsigned integers, floating point
- *Ordering* of bytes, e.g., big endian, little endian
- *Message sequencing*, e.g., first send username, then send password
- *Message start and end boundaries*, e.g., `CRLF` (`\r\n`) to terminate messages

Communication: Newlines

Depending on your system, "lines" of data are separated differently.

- `LF` (`\n` or "line feed")
  - Equivalent to `^J` where `^` is the "control" key.
  - UNIX-like systems use `LF` to end lines.
- `CRLF` ("carriage return" + "line feed")
  - Equivalent to `^M` followed by `^J`.
  - Microsoft operating systems (such as DOS and Windows) end lines with `CRLF`.
- `CR`, `LF` could also take the form of 13, 10 (ASCII), or `\015`, `\012` (octal).

Because the vast majority of servers (back in the day) and general-purpose computers ran Microsoft operating systems, the `CRLF` is the adopted convention.

Conversion

Many applications and network interfaces implicitly convert, but you can use `unix2dos` and `dos2unix` to convert files between the two newline conventions.

Deeper Reading: Alan J. Rosenthal's CSC209

`www.teach.cs.toronto.edu/~ajr/209/notes/sockets/newline.html`

Networks: Transport Protocols

Two widespread models of *transport protocols* for computer communication over a network:

1. Connectionless: Exemplified by `UDP` protocol
2. Connection-oriented: Exemplified by `TCP` protocol

Protocols are a set of *rules*. Both TCP and UDP protocols are implemented by the *operating system*.

Networks: User Datagram Protocol (UDP)

- UDP is used for sending a *datagram* from one machine to another
- A datagram is a self-contained message with a beginning and end.
- The OS sends the datagram, but doesn't follow up to make sure that it got delivered.
- UDP is connectionless.

Networks: Transmission Control Protocol (TCP)

- TCP is used to establish a *socket* (similar to a pipe) to communicate between two processes.
    - Processes may be on the same computer, or two different computers connected by a network.
- The socket is created using a system call.
- The process sending the data writes a sequence of bytes to the socket.
- The OS guarantees that the bytes will be delivered over the network, in the correct order, to the receiving process.
- TCP is connection-oriented.

- Comparing UDP and TCP is like comparing SMS and WhatsApp.
- If you send an SMS to your friend, you have no way of knowing if they received your message.
    - Perhaps they may reply back to you confirming that they received your message.
- If you send a message over WhatsApp, **the app itself tells you whether or not the message was successfully delivered**.

Networks: Classic Bad Jokes

We were planning to tell a UDP joke on this slide...

Figure 1: But we weren't sure if you would get it.

Networks: CSC209 vs. CSC*58

In this course, you learn what's necessary to *use* TCP to communicate over a network. No UDP due to time constraints. If you want to learn more…

CSC358H5: Principles of Computer Networks

Introduction to computer networks and systems programming of networks. Basic understanding of computer networks and network protocols. Network hardware and software, routing, addressing, congestion control, reliable data transfer, and socket programming.

CSC457H1: Principles of Computer Networks (formerly CSC358H1)

Fundamental principles of computer networks and currently used network architectures and protocols. […] It will highlight the trade-offs (and approaches to navigate these trade-offs) in the design of computer network protocols. Theoretical but similar to CSC358H5.

CSC458/2209H1: Computer Networking Systems (similar to CSCD58H3)

Computer networks with an emphasis on network systems, network programming, and applications. Networking basics: layering, routing, congestion control, and the global Internet. Network systems design and programming: Internet design, socket programming, and packet switching system fundamentals. Additional topics include network security, multimedia, software-defined networking, peer-to-peer networking, and online social networks.

Networks: Internet Protocol (IP) Addresses

An *IP address* identifies a specific host (computer) on a specific network.

- IPv4 addressing (most widespread) identifies hosts by four decimal 8-bit integers separated by dots, e.g., `128.100.3.30`. This is the current standard/default.
- IPv6 addressing identifies hosts by eight groups of four hex digits, separated by colons, e.g., `fe80:1234:0432:a2d8:61ff:fe8b:8924:c23f`.
- IPv4 is tractibly limited in the number of unique addresses, so IPv6 is the prospective replacement (adoption is quite slow).

You can use `dig` to find the IP address of a given URL. For example, Deerfield lab machines have IP address in the range `142.1.X.Y`.

```
$ dig dh2010pc11.utm.utoronto.ca | grep IN
;dh2010pc11.utm.utoronto.ca.    IN  A
dh2010pc11.utm.utoronto.ca. 8835 IN A   142.1.46.80
```

Networks: `localhost`

You may test your client and server programs by running both on the same machine.

- `127.0.0.1` is the "loopback" IP address, for when your program needs to communicate with another program on the same computer
- `localhost` is a *hostname* "aliased" to `127.0.0.1`

Networks: TCP and UDP Ports

An IP address only identifies a host, but not the program running on the host.

- To communicate with a specific program on a host, you must specify a *port number* between 0 and 65535.
- For some of your homework, your client and server programs must use the same port number; otherwise, they cannot communicate.

Port Number Conventions

- Ports in range 0–1023 are *well-known* or *reserved* (e.g., 22 for SSH, 80 for HTTP, 443 for HTTPS).
- Ports in range 1024–49151 are *registered* (e.g., 3724 for World of Warcraft, 1313 for `hugo` servers).
- Ports in range 49152–65535 are *dynamic*
  - These are the ones you should typically pick, to avoid conflict.

See IANA for list of port assignments

Networks: Useful Commands Addresses & Ports

dig - DNS lookup utility

Performs DNS lookups and displays answers returned by the queried name server(s).

ifconfig - show/manipulate routing, network devices, interfaces and tunnels

The command `ifconfig` provides a rich source of information. They list the local (`l*`), cabled (`e*`) and wireless (`w*`) connections to your machine. Combine with other UNIX commands like `grep` to find what you're looking for, like IP addresses.

- If `ifconfig` doesn't exist, try `ip addr` (although output format is less pleasing).

ping - send ICMP ECHO_REQUEST to network hosts

To see if a server is online and reachable, use `ping` to send small packets to diagnose the connection. It'll show IP addresses of the server URLs. Use `ping6` or `ping -6` for IPv6.

```
ping math.toronto.edu # 128.100.68.3; what's coexeter?
ping utm.utoronto.ca  # 142.150.1.50; what's erin?
ping openai.com       # 13.107.238.51; why does it take so long?
ping google.com       # 142.251.41.78; why doesn't this take long?
ping6 google.com      # try ping -6 if ping6 isn't found
```

You might see some unexpected/interesting/creative results...

A byte (unit of 8 bits) is internally atomic (its own bits are consistently ordered). But how a sequence of bytes is ordered/interpreted depends.

- *Little-endian* orders bytes from least to most significant. (Intel)
- *Big-endian* orders bytes from most to least significant. (most others)

By convention, data being sent over a network must first be converted into *big endian* (also known as *network byte order*).

- Any data received is thus assumed to be in network byte order.
- The endianness of the host is referred to as *host byte order*.

Example – IEEE-754 Floating Point

Consider the approximation $\tau \approx 6.28318530718$.

- Divide by 2 twice to get the mantissa 1.57079637050628662.
- The (biased) exponent is encoded as $129 = 10000001_2$
- The mantissa is encoded $4788187 = 10010010000111111011011_2$.

Try writing this IEEE-754 approximation in little-endian order.

Endianness: Assumption & Conversion

To ensure portability of code, we always convert *host byte order* to *network byte order* even if they're the same. This behaviour is abstracted by conversion functions.

```c
#include <arpa/inet.h>     // most systems use this
#include <netinet/in.h>    // some systems require this instead

uint32_t htonl(uint32_t hostlong);     // host to network long
uint16_t htons(uint16_t hostshort);    // host to network short
uint32_t ntohl(uint32_t netlong);      // network to host long
uint16_t ntohs(uint16_t netshort);     // network to host short
```

Even port numbers must be converted, but ASCII text does not; why?

Deeper Reading: Beej's Guide to Network Programming (3.2)

`beej.us/guide/bgnet/html/split/ip-addresses-structs-and-data-munging.html`

Networks: Summary

Transport Protocols

- UDP sends a *datagram* (bundle of bounded data) between machines without checking receipts. (connectionless)
- TCP communicates between processes with *sockets* using `write`/`read`. Receipt is implicitly guaranteed. (connection-oriented)

IP Addresses & Ports

- IP addresses identify hosts on a network; standards are IPv4 and IPv6.
- `127.0.0.1`/`localhost` refers back to the host itself; useful for development.
- Applications communicate externally on ports of the host they're running from.

Endianness

Network byte order is big-endian, and data is converted with system-dependent functions to ensure consistency and allow assumption.

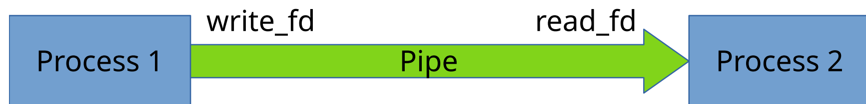
Deeper Reading: Beej's Guide to Network Programming (Section 3)

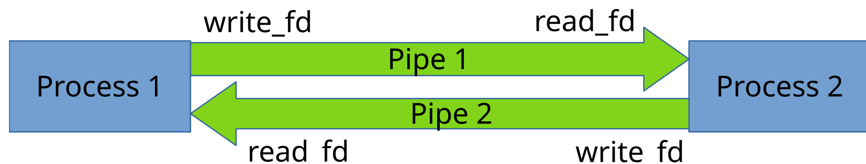
`beej.us/guide/bgnet/html/split/ip-addresses-structs-and-data-munging.html`

Section 2

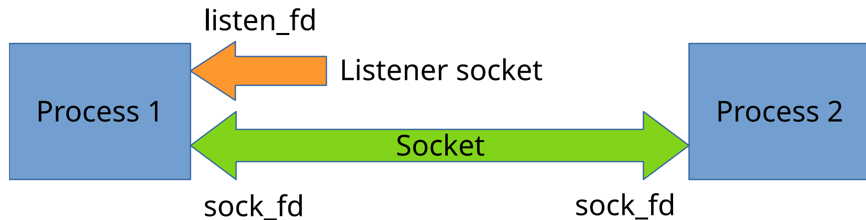
Sockets

Pipes: Unidirectional vs. Bidirectional Communication





Sockets



- A server must have a *listener socket* to accept new connections
- A separate socket is created to communicate with each client

Sockets: System Calls for Setting up a Server

```
#include "sys/socket.h"
int socket(int family, int type, int protocol);
int setsockopt(int sockfd, int level, int optname,  const void *optval.
                    socklen_t optlen);
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *restrict addr,
                    socklen_t *restrict addrlen);
```

A server is typically created and used like this...

1. `socket`: Creates a socket; configure with `setsockopt`.
2. `bind`: Assigns an address and port to the socket. (must assign an IP address that actually belongs to your systems).
3. `listen`: Establish a queue for incoming connections.
4. `accept`: Accept queued incoming connection and create a new socket.
5. `read`/`write` (or `recv`/`send`): Receive/send data on socket.
6. `close`: Close resources and the socket itself.

Sockets: System Calls for Setting up a Client

```
#include "sys/socket.h"
int socket(int family, int type, int protocol);
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

// analogs of read()/write() that take flags
ssize_t recv(int __fd, void *__buf, size_t __n, int __flags);
ssize_t send(int __fd, const void *__buf, size_t __n, int __flags);
```

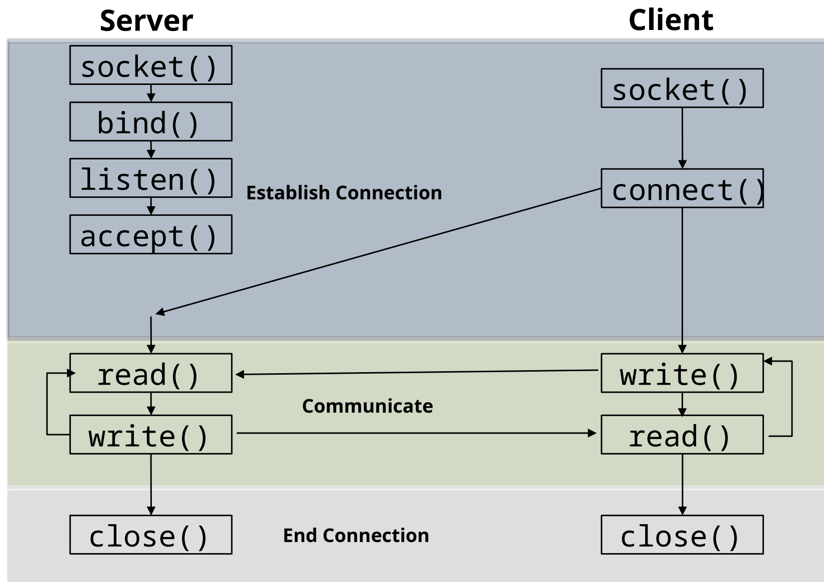
An accompanying client can be created to connect to such a server...

1. `socket`: Creates a socket; configure with `setsockopt`.
2. `connect`: Connects to a remote server using an IP address and port.
3. `read`/`write` (or `recv`/`send`): Receive/send data on socket.
4. `close`: Close resources and the socket itself.

Look familiar?

Recall the system calls `read`/`write`/`close` from `unistd.h`; you use them here just like with regular files and pipes! `recv`/`send` are analogs of `read`/`write` provided by `sys/socket.h` that can take flags (bit vector `int __flags`). See `man` pages!

Sockets: The Server-Client Model



Sockets: Identifying & Multiplexing

A socket can be identified by its source/destination addresses/ports, and its protocol.

`{SRC-IP, SRC-PORT, DEST-IP, DEST-PORT, PROTOCOL}`

This uniqueness allows some useful properties.

- A server or client can have multiple sockets, connecting to clients/servers.
- Multiple clients can connect to the same server socket, and it's up to the server to dispatch child processses (or threads) to process them.
  - This is how a shared machine can be used by multiple users at once.

What the community says...

`stackoverflow.com/questions/3329641/how-do-multiple-clients-connect-simultaneously-to-one-port-say-80-on-a-server`

Sockets: IP Addresses

System calls expect the IP address to be passed in as an `in_addr` struct.

```
#include "sys/socket.h"
struct in_addr { uint32_t s_addr; }; // 32 bits/4 bytes
struct in6_addr { unsigned char s6_addr[16]; }; // how big is this?
```

There are system calls to help you convert back and forth between `in_addr` structs and human-readable strings (`a.b.c`).

```
#include "arpa/inet.h"
int inet_aton(const char *cp, struct in_addr *inp); // a.b.c -> struct in_addr
char *inet_ntoa(struct in_addr in); // in_addr -> a.b.c
```

To support IPv6, there are analogs `inet_pton`/`inet_ntop`.

```
#include "arpa/inet.h"
int inet_pton(int af, const char *src, void *dst);
const char *inet_ntop(int af, const void *src,char *dst, socklen_t size);
```

The latter is favoured over the deprecated `inet_ntoa`. See `man 3 inet_ntop`.

Sockets: Socket Addresses for IPv4

Socket addresses contain 16 bytes, mostly for the protocol address.

```
#include "sys/socket.h"
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

However, when interfacing with IPv4 or IPv6, we only need to know the `AF_INET`, internet address and port number (which are part of the protocol address)!

```
// (IPv4 only--see struct sockaddr_in6 for IPv6)
struct sockaddr_in {
    short int          sin_family;  // address family, AF_INET
    unsigned short int sin_port;    // port number
    struct in_addr     sin_addr;    // internet address
    unsigned char      sin_zero[8]; // padded to sizeof(struct sockaddr)
};
```

But if `connect()` requires a `sockaddr`, how can we use `sockaddr_in`? They're the same size so you can cast either to the other.

Sockets: Socket Addresses for IPv6

For IPv6, we use sockaddr_in6. How big is this?

```
#include "sys/socket.h"
// (IPv6 only--see struct sockaddr_in and struct in_addr for IPv4)
struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};
```

If we want portable code, we'd abstract socket addresses for both IPv4 and IPv6 for as much interaction as possible. For that, there is the sockaddr_storage struct, a union-like that is padded/aligned to be cast-compatible with both IPv4 and IPv6.

Further Reading: Beej's Guide to Network Programming (3.3)

beej.us/guide/bgnet/html/split/ip-addresses-structs-and-data-munging.html

Sockets: Incomplete Messages & Boundaries

Messages aren't always perfect (i.e. incomplete), and sometimes there are many of them to process.

Suppose that a sender sends a sequence of bytes to a receiver over a TCP socket.

- TCP guarantees that the receiver will receive the entire sequence, eventually.
- But it's possible that when the receiver calls `read()` on the socket:
  - The entire message wasn't received yet.
  - The `read()` call was interrupted (e.g., see `EINTR` in `man 2 read`).

You would somehow need to know the correct sequence length (in other words, a known message boundary) to wait for or count up to. What is "correct" could be standardized, or declared in a prefix of the sequence.

Sockets: Buffering

What if you receive more messages faster than you're able to handle?

- Buffering is an extremely common technique, especially in networking.
- The operating system does its own buffering, but programmers typically also use buffering in many areas of systems/network/web programming.

Arnold provided a server-client example where clients send text to the server, which returns it back with alphabetical letters upper-cased.

```
mcs.utm.utoronto.ca/~209/23s/lectures/src/sockets/server.c
mcs.utm.utoronto.ca/~209/23s/lectures/src/sockets/client.c
```

Compile the server and client separately.

```
$ gcc -o server server.c
$ gcc -o client client.c
```

Then start the server and connect a client.

```
$ ./server &
$ ./client
```

Notice that the server is treating each client individually/independently...

What's next? Multiplexing & `select`

Can we build useful multi-user programs like file-sharing or chat applications? Not so fast! We'll discuss `select` and I/O multiplexing next week!

Section 3

Trivia Questions

Messages: Questions

1. How do we know when we have received a complete (not partial) message?
2. Are there alternative techniques for determining that we have received a complete message?

Messages: Questions & Answers

1. How do we know when we have received a complete (not partial) message?
   - *We define a byte sequence that indicates the end of a message. In text-based protocols, the most common convention is to signify an end-of-message with a CRLF (carriage-return + newline, or `\r\n`) sequence. Actual message content must not contain any instances of this sequence.*
2. Are there alternative techniques for determining that we have received a complete message? *Yes. Two common techniques:*
   - *Define a fixed-length message format (i.e., every message must be identical in length). Then check the number of bytes received in `read`/`recv`.*
   - *Define a fixed-length "header" that contains an integer representing the length of the remainder of the message. Also check the number of bytes received.*
3. Follow-up: in the header solution, would you still need any length standardization?
   - *Yes, but much less! You need only standardize the size of the header, which is much smaller than the rest of the message (which can now be of variable length indicated in the header).*

Sending/Receiving: Questions

3. Is it possible for the server (or client) to call `read()` on a socket and receive more than one message? Hint: how would pipes work?
4. What happens if your PC receives data from the network, but your program isn't ready to call `read()`/`recv()` yet, because it is busy doing something else.

Sending/Receiving: Questions & Answers

3. Is it possible for the server (or client) to call `read()` on a socket and receive more than one message? Hint: how would pipes work?
    - *Yes. This might happen if the sender is sending the messages faster than you are reading them. In this case, you must save the messages in a* ***buffer*** *and handle them one at a time.*
4. What happens if your PC receives data from the network, but your program isn't ready to call `read()`/`recv()` yet, because it is busy doing something else.
    - *The OS saves it in a buffer, until your program calls `read()`/`recv()`.*

Section 4

The TCP/IP Model

TCP/IP Model

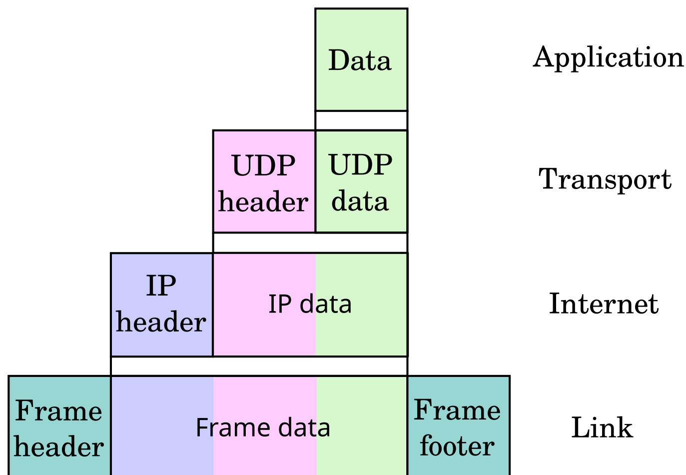


Figure 3: Layers in the TCP/IP Model

TCP/IP: Layers

The functionality of the TCP IP model is divided into four layers.

Application Layer

In CSC209, when we define a message format, what we're really doing is defining an *application-layer protocol*, governing communication between server and client.

Transport Layer

Transport protocols, such as TCP and UDP, govern how your OS "packages up" your application data to send it to another host over the network, and check to make sure that it arrived at the destination.

Internet Layer

Internet protocols such as IP, RIP, and OSPF govern how your data gets transferred from one Internet Service Provider (ISP; e.g., Bell, Rogers, UofT) to another.

Link Layer

Link-layer protocols deal with how your device physically transmits the data, e.g., wirelessly, or over a copper or fibre-optic cable.

TCP/IP: Why Layers?

Suppose you're writing an application that uses local or remote connections. Imagine if you had to write different versions of code based on whether the client is connected,

- over a WiFi connection, LTE, or an Ethernet cable...
- to the local area network (LAN), a wide area network (WAN), or beyond...

The layers of the TCP/IP model separate different code to perform discrete networking functions. Learn more in CSC*58!

Further Reading: TCP/IP Stack from GURU99

`www.guru99.com/tcp-ip-model.html#2`

TCP/IP: Additional Features

TCP has many more features that are beyond the scope of our discussions for this course. Here are a couple of solutions to data loss issues...

- Flow control: If a computer is sending data too fast for the receiver to handle, TCP will automatically slow down to avoid data loss.
- Congestion control: If the network is too congested, TCP will automatically slow down to avoid data loss.

Networking theory is quite statistical; that's why CSC*58 include an introductory probability/statistics course as their prerequisites.

Common TCP/IP-modelled protocols

In addition to the literal marriage of TCP and IP, other protocols modelled this way include DNS, FTP, HTTP/S, SMTP, SNMP, TELNET.