Client-side Networking CSE 333 Autumn 2025

Instructors: Naomi Alterman, Chris Thachuk

Teaching Assistants:

Ann Baturytski Derek de Leuw Blake Diaz

Rishabh Jain Chendur Jel Jayavelu Lucas Kwan

Irene Xin Jie Lau Nathan Li Maya Odenheim

Advay Patil Selim Saridede Deeksha Vatwani

Angela Wu Jiexiao Xu

Administrivia

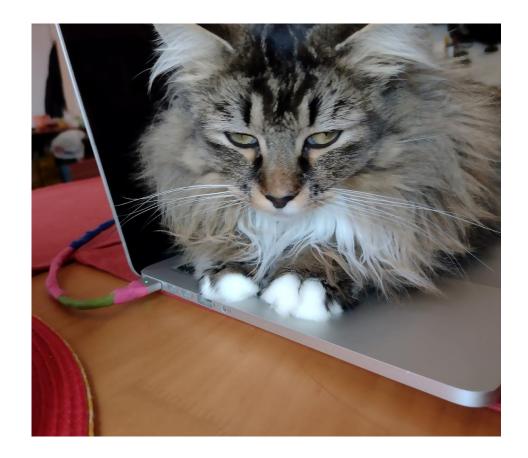
- The HW3 bell tolls for thee
 - Sorry we didn't have office hours yesterday
- Ex15 also out
 - Stitch together the code chunks from Monday and Wednesday

Socket API: Client TCP Connection

- There are five steps:
 - 1) Figure out the IP address and port to connect to
 - 2) Create a socket
 - 3) Connect the socket to the remote server
 - 4) read() and write() data using the socket
 - 5) Close the socket

Our networking multitool

- Netcat ("nc")
 - Does what "cat" does, but over the net
 - Specifically:
 - Opens a socket
 - Writes bytes from stdin to the socket
 - Prints bytes read from the socket to stdout
- Usage (TCP):
 - Client: nc HOST PORT
 - Server: nc -k -l PORT
 - To exit either of them: Ctrl+C



Step 1: DNS Lookup

- (from last time; details/examples in sections this week)
- * See dnsresolve.cc

Step 2: Creating a Socket

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

- Creating a socket doesn't bind it to a local address or port yet
- Returns file descriptor or -1 on error

socket.cc

```
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <iostream>
int main(int argc, char** argv) {
  int socket fd = socket (AF INET, SOCK STREAM, 0);
 if (socket fd == -1) {
     std::cerr << strerror(errno) << std::endl;</pre>
     return EXIT FAILURE;
  close(socket fd);
  return EXIT SUCCESS;
```

Step 3: Connect to the Server

- The connect() system call establishes a connection to a remote host
 - - sockfd: Socket file description from Step 2
 - addr and addrlen: Usually from one of the address structures returned by getaddrinfo in Step 1 (DNS lookup)
 - Returns 0 on success and -1 on error
- connect() may take some time to return
 - It is a blocking call by default
 - The network stack within the OS will communicate with the remote host to establish a TCP connection to it
 - This involves ~2 round trips across the network

How long are two "round trips"

 Exact numbers change somewhat over time, but you should know the order-of-magnitudes here

Numbers Everyone Should Know

L1 cache reference	0 .	.5 ns
Branch mispredict	5	ns
L2 cache reference	7	ns
Mutex lock/unlock	25	ns
Main memory reference	100	ns
Compress 1K bytes with Zippy	3,000	ns
Send 2K bytes over 1 Gbps network	k 20,000	ns
Read 1 MB sequentially from memo:	ry 250,000	ns
Round trip within same datacente:	500,000	ns
Disk seek	10,000,000	ns
Read 1 MB sequentially from disk	20,000,000	ns
Send packet CA->Netherlands->CA	150,000,000	ns

Connect Example

* See connect.cc

```
// Get an appropriate sockaddr structure.
struct sockaddr storage addr;
size t addrlen;
LookupName (arqv[1], port, &addr, &addrlen);
// Create the socket.
int socket fd = socket(addr.ss family, SOCK STREAM, 0);
if (socket fd == -1) {
  cerr << "socket() failed: " << strerror(errno) << endl;</pre>
 return EXIT FAILURE;
// Connect the socket to the remote host.
int res = connect(socket fd,
                  reinterpret cast<sockaddr*>(&addr),
                  addrlen);
if (res == -1) {
  cerr << "connect() failed: " << strerror(errno) << endl;</pre>
```

Step 4: read()

- If there is data that has already been received by the network stack, then read will return immediately with it
 - read() might return with *less* data than you asked for
- If there is no data waiting for you, by default read() will block until something arrives
 - This might cause deadlock!
 - Can read() return 0?

Step 4: write()

- write() enqueues your data in a send buffer in the OS and then returns
 - The OS transmits the data over the network in the background
 - When write () returns, the receiver probably has not yet received the data!
- If there is no more space left in the send buffer, by default write ()
 will block

Read/Write Example

```
while (1) {
  int wres = write(socket fd, readbuf, res);
  if (wres == 0) {
    cerr << "socket closed prematurely" << endl;</pre>
    close(socket fd);
    return EXIT FAILURE;
  if (wres == -1) {
    if (errno == EINTR)
      continue;
    cerr << "socket write failure: " << strerror(errno) << endl;</pre>
    close(socket fd);
    return EXIT FAILURE;
  break;
```

- * See sendreceive.cc
 - Demo

Step 5: close()

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

- Nothing special here it's the same function as with file I/O
- Shuts down the socket and frees resources and file descriptors associated with it on both ends of the connection

Extra Exercise #1

- Write a program that:
 - Reads DNS names, one per line, from stdin
 - Translates each name to one or more IP addresses
 - Prints out each IP address to stdout, one per line