

Client-side Networking

CSE 333

Instructor: Hannah C. Tang

Teaching Assistants:

Deeksha Vatwani Hannah Jiang Jen Xu

Justin Tysdal Leanna Nguyen Sayuj Shahi

Wei Wu Yiqing Wang Youssef Ben Taleb

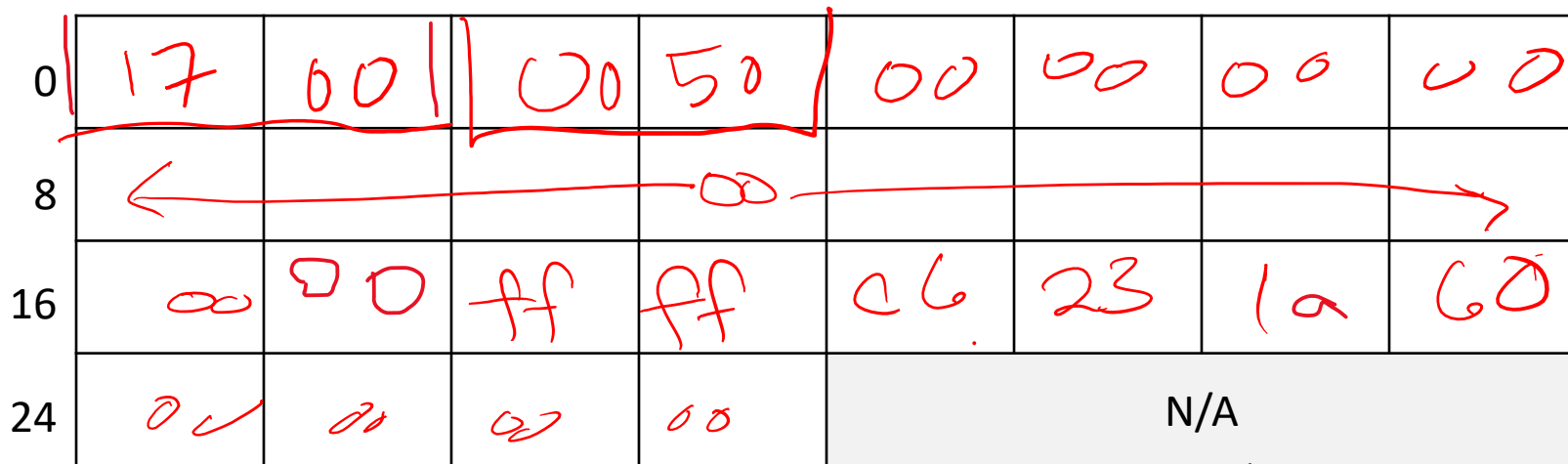
- ❖ Address: 198.35.26.96 (::ffff::c6:23:1a:60). Port 80 (0x50).

AF_INET6 = 23 (0x17)

- ❖ Little-endian host. Zero-fill flow and scope

```

struct sockaddr_in6 {
    sa_family_t    sin6_family;    // Address family: AF_INET6
    in_port_t      sin6_port;      // Port number
    uint32_t       sin6_flowinfo;  // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    uint32_t       sin6_scope_id;  // Scope ID
};
    
```



Administrivia

- ❖ HW2 grades released; regrade requests open Saturday afternoon and remain open for 7d
 - (more generally, regrade requests always open ~24h after grades are released)
- ❖ HW3 due in 4 days
- ❖ Ex15 released today, due on Wednesday (sorry)

Lecture Outline

- ❖ **Finish IP addresses and DNS**
- ❖ Reading/writing to a socket

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

The entire previous lecture was this single step!

Step 1: DNS Lookup

- ❖ (from last time; details/examples in sections yesterday)
- ❖ See `dnsresolve.cc`

```
struct addrinfo {
    int      ai_flags;           // additional flags
    int      ai_family;         // AF_INET, AF_INET6, AF_UNSPEC
    int      ai_socktype;       // SOCK_STREAM, SOCK_DGRAM, 0
    int      ai_protocol;       // IPPROTO_TCP, IPPROTO_UDP, 0
    size_t   ai_addrlen;        // length of socket addr in bytes
    struct sockaddr* ai_addr;    // pointer to socket addr
    char*    ai_canonname;      // canonical name
    struct addrinfo* ai_next;    // can form a linked list
};
```

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;
        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

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

- 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

Connect Example

❖ See `connect.cc`

```
// Get an appropriate sockaddr structure.
struct sockaddr_storage addr;
size_t addrlen;
LookupName(argv[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;
    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;
}
```

- ❖ How do we *error check* `read()` and `write()` ?
 - A. `ferror()`
 - B. Return value less than expected
 - C. Return value of 0 or NULL
 - D. Return value of -1
 - E. We're lost...

Lecture Outline

- ❖ Finish IP addresses and DNS
- ❖ **Reading/writing to a socket**

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;
        close(socket_fd);
        return EXIT_FAILURE;
    }
    if (wres == -1) {
        if (errno == EINTR)
            continue;
        cerr << "socket write failure: " << strerror(errno) << endl;
        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