CS118 Discussion 1A, Week 1

Zengwen Yuan (zyuan [at] cs.ucla.edu)
Boelter Hall 9436, Friday 12:00—1:50 p.m.
TA

• Zengwen Yuan ( zyuan [at] cs.ucla.edu )

• Discussion session (1A):
  • Boelter Hall 9436
  • Friday 12:00 – 1:50 p.m.

• Office hours:
  • BH 2432 Monday 4:30 – 6:30 p.m.
  • By appointment

• Class website: http://web.cs.ucla.edu/~zyuan/teaching/cs118.html
Logistics

• Turn in your signed Academic Integrity Agreement

• Grade decomposition:
  • Homework: 20% (due 6:00 p.m. next Wednesday)
  • Project 1: 8% (due Friday, Feb. 3rd)
  • Project 2: 12% (due Friday, Mar. 17th)
  • Midterm: 30% (Thursday, Feb. 16th, in-class)
  • Final: 30% (Wednesday, Mar. 22nd)
Logistics: Homework

• Turn it to the dropbox located at BH2432 before 6:00 p.m. the next Wednesday.

• Late submission will not be graded.

• Submission guidelines:
  
  • 1. Only paper copy of the submitted homework solution is accepted.
  
  • 2. Make sure that you do not write/print answers on both sides of the paper. We will only scan single side for grading.
  
  • 3. We recommend you to use the US letter size paper. Paper size must not exceed the US letter size.
  
  • 4. Do not staple multiple pages of your homework solution! If the homework solution includes multiple pages, you are responsible for writing/printing your names on every page you turned in.
Logistics: Project

• Two projects (C/C++ only):
  • simple web server — introduction to network programming;
  • reliable data transfer — a simple user-level TCP-like transport protocol

• Form a team of 2 persons.

• Test environment:
  • Pre-configured VirtualBox image:
    • http://metro.cs.ucla.edu/cs118/CS118_OVF.rar
    • Password: cs118

• Demo code: ~/workspace/ClientServer_Example/
Network programming

• What is the model for network programming?
• Where are we programming?
• Which APIs can we use? How to use them?
Client-server model

- Asymmetric communication
  - Client — Requests data:
    - Initiates communication
    - Waits for server’s response
  - Server (Daemon) — Responds data requests:
    - Well-known by clients (e.g. IP address + port)
    - Waits for clients connection
    - Processes requests, send replies
Client-server model

- Client and server are not disjoint
  - A client can be a server of another client
  - A server can be a client of another server
  - Example?

- Server’s service model
  - Concurrent: server processes multiple clients’ requests simultaneously
  - Sequential: server processes clients’ requests one by one
  - Hybrid: server maintains multiple connections, but responses sequentially
Network programming

- What is the model for network programming?
- **Where are we programming?**
- Which APIs can we use? How to use them?
Which layer are we at?

- “Clients” and “servers” are programs at application layer.
- Transport layer is responsible for providing communication services for application layer.
- Basic transport layer protocols:
  - TCP
  - UDP
TCP: Transmission Control Protocol

• A connection is set up between client and server

• Reliable data transfer
  • Guarantee deliveries of all data
  • No duplicate data would be delivered to application

• Ordered data transfer
  • If A sends data D1 followed by D2 to B, B will also receive D1 before D2

• Data transmission: full-duplex byte stream

• Regulated data flow: flow control and congestion control
UDP: User Data Protocol

- Basic data transmission service
  - Unit of data transfer: datagram (in variable length)
- No reliability guarantee
- No ordered delivery guarantee
- No flow control / congestion control
Network programming

• What is the model for network programming?

• Where are we programming?

• **Which APIs can we use? How to use them?**
Our secret weapon: socket programming APIs

- From Wikipedia: “A network socket is an endpoint of an inter-process communication flow across a computer network”

- A socket is a tuple of <ip:port>

- Socket programming APIs help build the communication tunnel between applications and transport/network service

- We use TCP socket in this project
Socket: port number

- Port numbers are allocated and assigned by the IANA (Internet Assigned Numbers Authority)

- See RFC 1700 or https://www.ietf.org/rfc/rfc1700.txt

<table>
<thead>
<tr>
<th>Port Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-512</td>
<td>standard services (see /etc/services), super-user only</td>
</tr>
<tr>
<td>513-1023</td>
<td>registered and controlled, also used for identity verification, super-user only</td>
</tr>
<tr>
<td>1024-49151</td>
<td>registered services/ephemeral ports</td>
</tr>
<tr>
<td>49152-65535</td>
<td>private/ephemeral ports</td>
</tr>
</tbody>
</table>
TCP socket: basic steps

• Create service
• Establish a TCP connection
• Send and receive data
• Close a TCP connection
TCP socket: service setup
TCP socket: service setup

TCP Client

TCP Server

socket()
TCP socket: service setup

TCP Client

TCP Server

socket()

bind()
TCP socket: service setup

TCP Client

TCP Server

socket()

bind()

listen()
TCP socket: service setup

**TCP Client**

**TCP Server**

1. `socket()`
2. `bind()`
3. `listen()`
4. `accept()`
TCP socket: service setup

TCP Client

TCP Server

- socket()
- bind()
- listen()
- accept()

blocked until connection from client
TCP socket: service setup

TCP Client

TCP Server

- socket()
- bind()
- listen()
- accept()
TCP socket: establish connection

**TCP Client**

socket()

connect()

**TCP Server**

socket()

bind()

listen()

accept()

blocked until connection from client
TCP socket: send and receive data

TCP Client

socket( )

connect( )

write( )

TCP Server

socket( )

bind( )

listen( )

accept( )

blocked until connection from client
TCP socket: send and receive data

TCP Client

socket()
connect()
write()

TCP Server

socket()
bind()
listen()
accept()
read()

Data (request)

Blocked until connection from client
TCP socket: send and receive data

TCP Client
- socket()
- connect()
- write()

TCP Server
- socket()
- bind()
- listen()
- accept()
- process request
- read()
- write()

Blocked until connection from client

Data (request)
TCP socket: send and receive data

TCP Client

socket() → connect() → write() → read()

TCP Server

socket() → bind() → listen() → accept() → read() → process request → write() blocked until connection from client

data (request) → data (reply)
TCP socket: close connection

**TCP Client**
- `socket()`
- `connect()`
- `write()`
- `read()`
- `close()`

**TCP Server**
- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `read()`
- `process request` (unconnected)
- `write()`

Blocked until connection from client

Data flow:
- Data (request) from client to server
- Data (reply) from server to client
TCP socket: close connection

TCP Client

socket()
connect()
write()
read()
close()

TCP Server

socket()
bind()
listen()
accept()
read()
process request
write()
read()
close()

blocked until connection from client
Socket programming API: syscalls

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

  - Create a socket

  - returns the socket descriptor or -1(failure). Also sets errno upon failure

- `domain`: protocol family

  - `PF_INET` for IPv4, `PF_INET6` for IPv6, `PF_UNIX` or `PF_LOCAL` for Unix socket, `PF_ROUTE` for routing

- `type`: communication style

  - `SOCK_STREAM` for TCP (with `PF_INET`)

  - `SOCK_DGRAM` for UDP (with `PF_INET`)

- `protocol`: protocol within family, which is typically set to 0
Socket programming API: syscalls

- `int bind(int sockfd, struct sockaddr* myaddr, int addrlen);`
  - Bind a socket to a local IP address and port number
  - returns 0 on success, -1 and sets errno on failure
  - `sockfd`: socket file descriptor returned by `socket()`
  - `myaddr`: includes IP address and port number
    - **NOTE**: sockaddr and sockaddr_in are of same size, use sockaddr_in and convert it to socketaddr
  - `sin_family`: protocol family, e.g. AF_INET
  - `sin_port`: port number assigned by caller
  - `sin_addr`: IP address
    - `sin_zero`: used for keeping same size as sockaddr
  - `addrlen`: sizeof(struct sockaddr_in)

```c
struct sockaddr {
    short    sa_family;
    char sa_data[14];
};
```

```c
struct sockaddr_in {
    short    sin_family;
    ushort   sin_port;
    struct   in_addr sin_addr;
    unsigned char        sin_zero[8];
};
```
Socket programming API: syscalls

- `int listen(int sockfd, int backlog);`
  - Put socket into passive state (wait for connections rather than initiating a connection)
  - returns 0 on success, -1 and sets errno on failure
  - **sockfd**: socket file descriptor returned by `socket()`
  - **backlog**: the maximum number of connections this program can serve simultaneously
Socket programming API: syscalls

- int accept(int sockfd, struct sockaddr* client_addr, int* addrlen);

  - Accept a new connection

  - Return client’s socket file descriptor or -1. Also sets errno on failure

  - sockfd: socket file descriptor for server, returned by socket()

  - client_addr: IP address and port number of a client (returned from call)

  - addrlen: length of address structure = pointer to int set to sizeof(struct sockaddr_in)

  - NOTE: client_addr and addrlen are result arguments

    - i.e. The program passes empty client_addr and addrlen into the function, and the kernel will fill in these arguments with client’s information (why do we need them?)
Socket programming API: syscalls

- **int connect (int sockfd, struct sockaddr* server_addr, int addrlen);**
  - Connects to another socket (server)
  - Return 0 on success, -1 and sets errno on failure
  - **sockfd**: socket file descriptor (returned from socket)
  - **server_addr**: IP address and port number of the server
    - server’s IP address and port number should be known in advance
  - **addrlen**: sizeof(struct sockaddr_in)
Socket programming API: syscalls

- int write(int sockfd, char* buf, size_t nbytes);
  - Write data to a TCP stream
  - Return the number of sent bytes or -1 on failures

- sockfd: socket file descriptor from socket()

- buf: data buffer

- nbytes: the number of bytes that caller wants to send
Socket programming API: syscalls

- `int read(int sockfd, char* buf, size_t nbytes);`
  - Read data from TCP stream
  - Return the number of bytes read or -1 on failures
  - Return 0 if socket is closed

- `sockfd`: socket file descriptor returned from `socket()`

- `buf`: data buffer

- `nbytes`: the number of bytes that caller can read (usually set as buffer size)
Socket programming API: syscalls

- `int close(int sockfd);`
  - close a socket
  - return 0 on success, or -1 on failure
  - After close, sockfd is no longer valid
Caveat: byte ordering matters

- Little Endian: least significant byte of word is stored in the lowest address
- Big Endian: most significant byte of word is stored in the lowest address
- Hosts may use different orderings, so we need byte ordering conversion
- **Network Byte Order = Big Endian**

![Byte Order Diagram]

<table>
<thead>
<tr>
<th>Least significant Byte</th>
<th>Most significant Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xBB</td>
<td>0xAA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Most significant Byte</th>
<th>Least significant Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xAA</td>
<td>0xBB</td>
</tr>
</tbody>
</table>

0x1000 0x1001

*Little-Endian*

*Big-Endian (Network-Byte-Order)*

*Memory address*
Caveat: byte ordering matters

- Byte ordering functions: used for converting byte ordering

- Example:

  ```c
  int m, n;
  short int s, t;
  
m = ntohl (n)  net-to-host long (32-bit) translation
s = ntohs (t)  net-to-host short (16-bit) translation
n = htonl (m)  host-to-net long (32-bit) translation
t = htons (s)  host-to-net short (16-bit) translation
  ```

- Rule: for every int or short int

- Call htonl( ) or htons( ) before sending data

- Call ntohl( ) or ntohs( ) before reading received data
Address util functions

• All binary values are network byte ordered

• **struct hostent** *gethostbyname*(const char* hostname);
  • Translate host name (e.g. “localhost”) to IP address (with DNS working)

• **struct hostent** *gethostbyaddr*(const char* addr, size_t len, int family);
  • Translate IP address to host name

• **char** *inet_ntoa*(struct in_addr inaddr);
  • Translate IP address to ASCII dotted-decimal notation (e.g. “192.168.0.1”)

• **int gethostname**(char* name, size_t namelen);
  • Read local host’s name
Address util functions

• `in_addr_t inet_addr (const char* strptr);`
  
  • Translate dotted-decimal notation to IP address (network byte order)

    ```c
    struct sockaddr_in ina;
    ina.sin_addr.s_addr = inet_addr("10.12.110.57");
    ```

• `int inet_aton (const char* strptr, struct in_addr *inaddr);`

  • Translate dotted-decimal notation to IP address

    ```c
    struct sockaddr_in my_addr;
    my_addr.sin_family = AF_INET;      // host byte order
    my_addr.sin_port = htons(MYPORT);  // short, network byte order
    inet_aton("10.12.110.57", &(my_addr.sin_addr));
    memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
    ```
### FYI: struct hostent

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *h_name</td>
<td>The real canonical host name.</td>
</tr>
<tr>
<td>char **h_aliases</td>
<td>A list of aliases that can be accessed with arrays—the last element is NULL.</td>
</tr>
<tr>
<td>int h_addrtype</td>
<td>The result's address type, which really should be AF_INET for our purposes.</td>
</tr>
<tr>
<td>int length</td>
<td>The length of the addresses in bytes, which is 4 for IP (version 4) addresses.</td>
</tr>
<tr>
<td>char **h_addr_list</td>
<td>A list of IP addresses for this host. Although this is a char**, it's really an array of struct in_addr*s in disguise. The last element is NULL.</td>
</tr>
<tr>
<td>h_addr</td>
<td>A commonly defined alias for h_addr_list[0]. If you just want any old IP address for this host (they can have more than one) just use this field.</td>
</tr>
</tbody>
</table>
How to write a server: headers

/* PLEASE include these headers */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#define PORT 5000 /* Avoid reserved ports */
#define BACKLOG 10 /* pending connections queue size */
How to write a server: body (I)

```c
int main()
{
    int sockfd, new_fd; /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr; /* my address */
    struct sockaddr_in their_addr; /* connector addr */
    int sin_size;

    /* create a socket */
    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }
}
```
How to write a server: body (II)

// ...
/* bind the socket */
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT); /* short, network byte order */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
/* INADDR_ANY allows clients to connect to any one of the host’s IP address */

if (bind(sockfd, (struct sockaddr *) &my_addr,
        sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
}

// ...
    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

while(1) { /* main accept() loop */
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr*)
                        &their_addr, &sin_size)) == -1) {
        perror("accept");
        continue;
    }
    printf("server: got connection from %s\n", 
            inet_ntoa(their_addr.sin_addr));
    close(new_fd);
}
}
How to write a client?

```c
/* include all the headers */
int main() {
    int sockfd, new_fd; /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr; /* my address */
    struct sockaddr_in their_addr; /* connector addr */
    struct hostent* he;
    int sin_size;

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        perror ("socket");
        exit (1);
    }

    their_addr.sin_family = AF_INET; /* interp'd by host */
    their_addr.sin_port = htons (PORT);
    their_addr.sin_addr = *((struct in_addr*) he->h_addr);

    if (connect (sockfd, (struct sockaddr*) &their_addr,
                 sizeof (struct sockaddr)) == -1) {
        perror ("connect");
        exit (1);
    }
    return 0;
}
```
Summary: what we have learned today

• What is the model for network programming?
  • **Client-Server model**
  
• Where are we programming?
  • **TCP and UDP in a nutshell**

• Which APIs can we use? How to use them?
  • **Socket programming**
Further Reading


• Beej’s Guide to Network Programming (http://beej.us/guide/bgnet)

• Socket Programming from Dartmouth, http://www.cs.dartmouth.edu/~campbell/cs60/socketprogramming.html

• C/C++ reference: http://en.cppreference.com
See you next time!

• TA: Zengwen Yuan
• Email: zyuan [at] cs.ucla.edu
• OH: BH2432 Mon 4:30–6:30p
• Website:
  • http://web.cs.ucla.edu/~zyuan/teaching/cs118.html
• Turn in your signed Academic Integrity Agreement if you haven’t done so!
• Credit: previous TA Yuanjie Li for the materials