EE122: Socket Programming

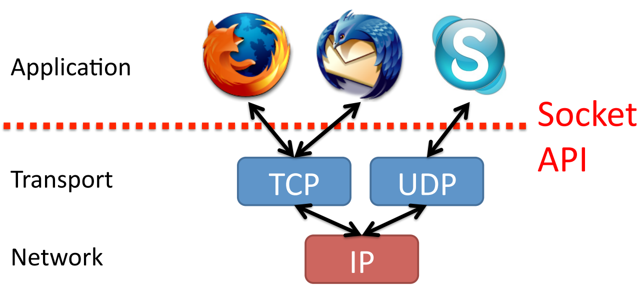
DK Moon
http://inst.eecs.berkeley.edu/~ee122

Socket API?

- Q. What would you expect when learning a new Unix command (*e.g., ls*) ?
  a) Source code **=> Implementation detail**
  b) Program options **=> Interface**
- *Application Programming Interface (API)*
  – Interface to a particular "service"
  – Abstracts away from implementation detail
  – Set of functions, data structures, and constants.
- Socket API
  – Network programming interface

Socket API

- Socket API
  – Network programming interface



BSD Socket API

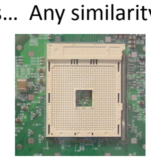
- From your university, UC Berkeley (1980s)
- Most popular network API
- Ported to various OSes, various languages
  – Windows Winsock, BSD, OS X, Linux, Solaris, …
  – Socket modules in Java, Python, Perl, …
- Similar to Unix file I/O API
  – In the form of *file descriptor* (sort of handle).
  – Can share the same `read()/write()/close()` system calls.

Outline

- Socket API motivation, background
- Types of sockets (TCP vs. UDP)
- Elementary API functions
- I/O multiplexing
- Project 1 – tiny World of Warcraft
- Appendix (not covered in the lecture)

Sockets

- Various sockets… Any similarity?


- Endpoint of a connection
  – Identified by IP address and Port number
- Primitive to implement high-level networking interfaces
  – e.g., Remote procedure call (RPC)

Types of Sockets

Stream socket (aka TCP)

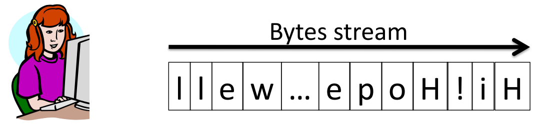
- Connection-oriented
  - Requires connection establishment & termination
- Reliable delivery
  - In-order delivery
  - Retransmission
  - No duplicates
- High variance in latency
  - Cost of the reliable service
- File-like interface (streaming)
- E.g., HTTP, SSH, FTP, …

Datagram socket (aka UDP)

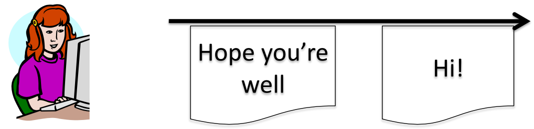
- Connection-less
- "Best-effort" delivery
  - Possible out-of-order delivery
  - No retransmission
  - Possible duplicates
- Low variance in latency
- Packet-like interface
  - Requires packetizing
- E.g., DNS, VoIP, VOD, AOD, …

Types of Sockets (cont'd)

- When sending "Hi!" and "Hope you're well"
- TCP treats them as a single bytes stream

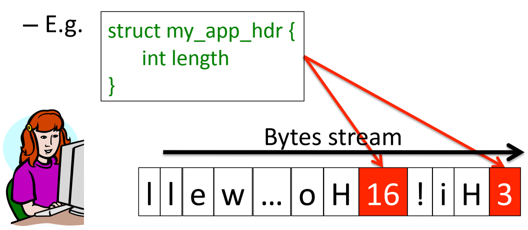


- UDP treats them as separate messages



Types of Sockets (cont'd)

- Thus, TCP needs application-level message boundary.
  - By carrying length in application-level header
  - E.g.

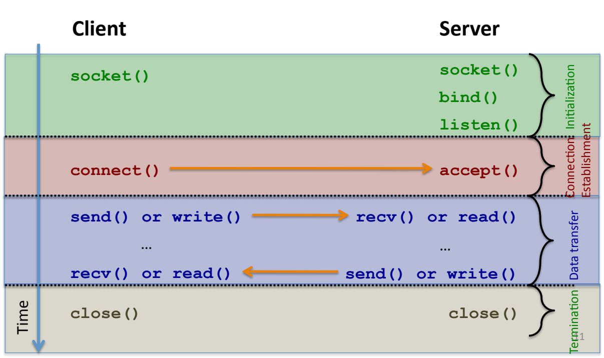


Outline

- Socket API motivation, background
- Types of sockets (TCP vs. UDP)
- Elementary API functions
- I/O multiplexing
- Project 1 – tiny World of Warcraft

Scenario #1 – TCP client-server

- Sequence of actions



Initialization: server + client, `socket()`

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
  perror("socket() failed");
  abort();
}
```

- **`socket()`: returns a socket descriptor**
- **`AF_INET`: IPv4 address family.** (also OK with PF_INET)
  - C.f. IPv6 => AF_INET6
- **`SOCK_STREAM`: streaming socket type**
  - C.f. SOCK_DGRAM
- **`perror()`**: prints out an error message

Error code in Unix programming

```
extern int  errno;   // by #include <errno.h>
```

- Many Unix system calls and library functions set `errno` on errors
- Macros for error codes ('E' + error name)
  - `EINTR, EWOULDBLOCK, EINVAL, …`
  - "`man` *func_name*" shows possible error code for the function name
- Functions to convert error code into human readable msgs
  - `void perror(const char *my_str)`
    - Always looks for **`errno`**
    - prints out "my str: error code string"
  - `const char *strerror(int err_code)`
    - You must provide an error code
    - returns a string for the `err_code`

Initialization: server, `bind()`

- Server needs to bind a particular port number.

```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(server_port);

if (bind(sock, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
   perror("bind failed");
   abort();
}
```

- **`bind()`**: binds a socket with a particular port number.
  - Kernel remembers which process has bound which port(s).
  - Only one process can bind a particular port number at a time.
- **struct sockaddr_in**: Ipv4 socket address structure. (c.f., struct sockaddr_in6)
- **INADDR_ANY**: If server has multiple IP addresses, binds any address.
- **htons()**: converts host byte order into network byte order.

Endianess

- Q) You have a 16-bit number: 0x0A0B. How is it stored in memory?

Increasing address →

0x0A	0x0B	**Big Endian**
0x0B	0x0A	**Little Endian**

Increasing address →

- Host byte order is not uniform
  - Some machines are Big endian, others are Little endian
- Communicating between machines with different host byte orders is problematic
  - Transferred $256 (0x0100), but received $1 (0x0001)

Endianness (cont'd)

- Network byte order: Big endian
  - To avoid the endian problem
- We must use network byte order when sending 16bit, 32bit , 64bit numbers.
- Utility functions for easy conversion

```
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);
```

- Hint: **h**, **n**, **s**, and **l** stand for host byte order, network byte order, short(16bit), and long(32bit), respectively

Initialization: server, `bind()`

- Server needs to bind a particular port number.

```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(server_port);

if (bind(sock, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
   perror("bind failed");
   abort();
}
```

- **`bind()`**: binds a socket with a particular port number.
  - Kernel remembers which process has bound which port(s).
  - Only one process can bind a particular port number at a time.
- **struct sockaddr_in**: Ipv4 socket address structure. (c.f., struct sockaddr_in6)
- **INADDR_ANY**: If server has multiple IP addresses, binds any address.
- **htons()**: converts host byte order into network byte order.

Reusing the same port

- After TCP connection closes, waits for 2MSL, which is twice maximum segment lifetime (from 1 to 4 mins, implementation dependent). Why?
- Segment refers to maximum size of packet
- Port number cannot be reused before 2MSL
- But server port numbers are fixed => Must be reused
- Solution: Put this code before `bind()`

```
int optval = 1;
if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
  &optval, sizeof(optval)) < 0) {
  perror("reuse failed");
  abort();
}
```

- **setsockopt()**: changes socket, protocol options.
  - e.g., buffer size, timeout value, …

Initialization: server, listen()

- Socket is active, by default
- We need to make it passive to get connections.

```
if (listen(sock, back_log) < 0) {
  perror("listen failed");
  abort();
}
```

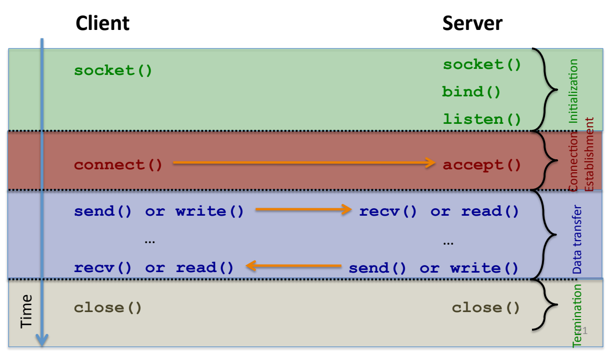
- **listen()**: converts an active socket to passive
- **back_log**: connection-waiting queue size. (e.g., 32)
  - Busy server may need a large value (e.g., 1024, …)

Initialization Summary

- Client
  - socket()
- Server
  - socket()
  - setsockopt(sock, SOL_SOCKET, SO_REUSEADDR)
  - bind()
  - listen()
- Pitfalls
  - The order of the functions matter
  - Do not forget to use htons() to handle port number

Scenario #1 – TCP client-server

- Sequence of actions



Connection Establishment (client)

```
struct sockaddr_in sin;
memset(&sin, 0 ,sizeof(sin));

sin.sin_family   = AF_INET;
sin.sin_addr.s_addr = inet_addr("128.32.132.214");
sin.sin_port = htons(80);

if (connect(sock, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
   perror("connection failed");
   abort();
}
```

- **Connect()**: waits until connection establishes/fails
- **inet_addr()**: converts an IP address string into a 32bit address number (network byte order).

Host name, IP address, Port number

- Host name
  - Human readable name (e.g., www.eecs.berkeley.edu)
  - Variable length
  - Could have multiple IP addresses
- IP version 4 address
  - Usually represented as dotted numbers for human readability
    - E.g., 128.32.132.214
  - 32 bits in network byte order
    - E.g., 1.2.3.4 => 0x04030201
- Port number
  - Identifies a service (or application) on a host
    - E.g., TCP Port 80 => web service, UDP Port 53 => name service (DNS)
  - 16 bit unsigned number (0~65535)

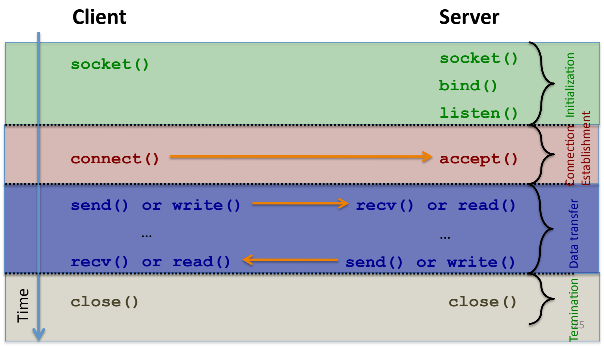
Connection Establishment (server)

```
struct sockaddr_in client_sin;
int addr_len = sizeof(client_sin);
int client_sock = accept(listening_sock,
                         (struct sockaddr *) &client_sin,
                         &addr_len);
if (client_sock < 0) {
   perror("accept failed");
   abort();
}
```

- **accept()**: returns a new socket descriptor for a client connection in the connection-waiting queue.
  - This socket descriptor is to communicate with the client
  - The passive socket (listening_sock) is not to communicate with a client
- **client_sin**: contains client IP address and port number
  - Q) Are they in Big endian or Litten endian?

Scenario #1 – TCP client-server

- Sequence of actions



Sending Data: server+client, send()

```
char *data_addr = "hello, world";
int data_len = 12;

int sent_bytes = send(sock, data_addr, data_len, 0);
if (sent_bytes < 0) {
  perror("send failed");
}
```

- **send()**: sends data, returns the number of sent bytes
  - Also OK with write(), writev()
- **data_addr**: address of data to send
- **data_len**: size of the data

- With blocking sockets (default), send() blocks until it sends all the data.
- With non-blocking sockets, **sent_bytes** may not equal to **data_len**
  - If kernel does not have enough space, it accepts only partial data
  - You must retry for the unsent data

Receiving Data: server+client, recv()

```
char buffer[4096];
int expected_data_len = sizeof(buffer);

int read_bytes = recv(sock, buffer, expected_data_len, 0);
if (read_bytes == 0) {  // connection is closed
  …
} else if (read_bytes < 0) { // error
  perror("recv failed");
} else {    // OK. But no guarantee read_bytes == expected_data_len
  …
}
```

- **recv()**: reads bytes from the socket and returns the number of read bytes.
  - Also OK with **read()** and **readv()**

- **read_bytes** may not equal to **expected_data_len**
  - If no data is available, it blocks
  - If only partial data is available, read_bytes < expected_data_len
  - On socket close, expected_data_len equals to 0 (not error!)
  - If you get only partial data, you should retry for the remaining portion.

Termination: server+client, close()

```
// after use the socket
close(sock);
```

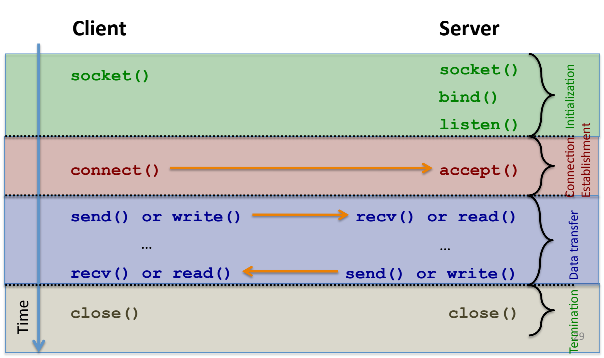
- **close()**: closes the socket descriptor

- We cannot open files/sockets more than 1024*
  - We must release the resource after use

* Super user can overcome this constraint, but regular user cannot.

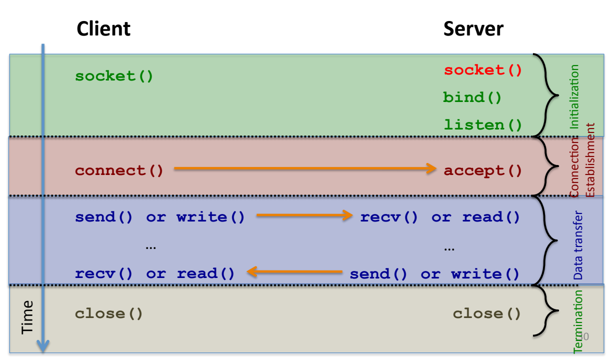
Scenario #2 – UDP client-server

- Q) What must be changed?



Scenario #2 – UDP client-server

- A) We need a different initialization



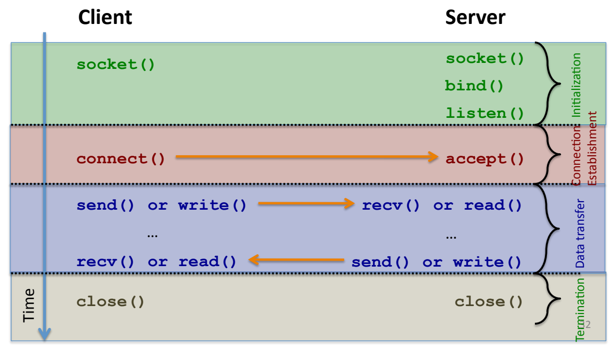
Initialization: UDP

```
int sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock < 0) {
  perror("socket failed");
  abort();
}
```

- UDP uses SOCK_DGRAM instead of SOCK_STREAM

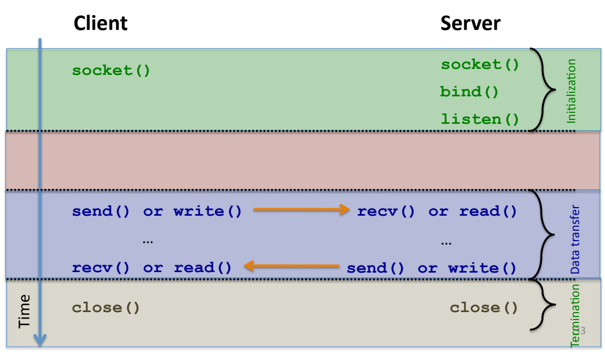
Scenario #2 – UDP client-server

- Q) What else must be changed?



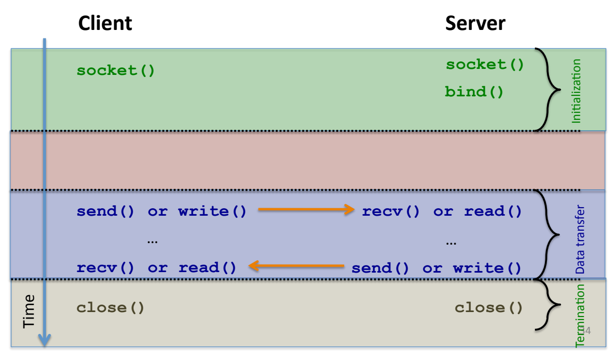
Scenario #2 – UDP client-server

- A) UDP is **connection-less**. We remove all connection related steps.



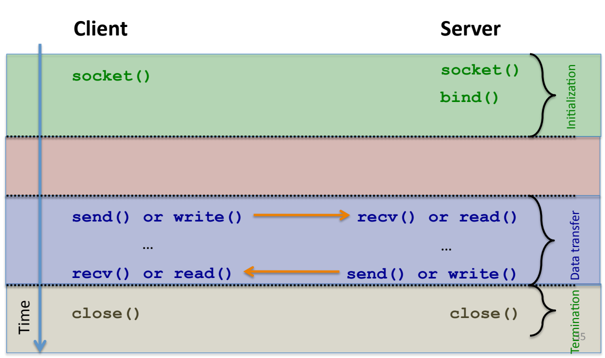
Scenario #2 – UDP client-server

- A) `listen()` is also related to connection. Remove it.



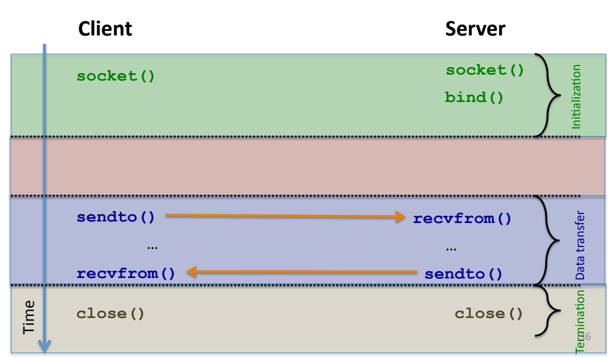
Scenario #2 – UDP client-server

- Q) Now it's unclear where to send packets and from where I receive! Can we solve this?



Scenario #2 – UDP client-server

- A) Give <address,port> information when sending a packet. That is, use **`sendto()`** and **`recvfrom()`** instead of `send()` and `recv()`



Send Data Over UDP: `sendto()`

```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(sin));

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = inet_addr("128.32.132.214");
sin.sin_port = htons(1234);

sent_bytes = sendto(sock, data, data_len, 0,
                    (struct sockaddr *) &sin, sizeof(sin));
if (sent_bytes < 0) {
   perror("sendto failed");
   abort();
}
```

- `sendto()`: sends a packet to a specific destination address and port
  - c.f., in TCP, we do this destination setting when calling `connect()`
- As opposed to TCP, UDP packetizes data. So, `sendto()` sends all data or nothing.

Receive Data Over UDP: `recvfrom()`

```
struct sockaddr_in sin;
int sin_len;
char buffer[4096];

int read_bytes = recvfrom(sock, buffer, sizeof(buffer), 0,
                          (struct sockaddr *) &sin, &sin_len);

if (read_bytes < 0) {
   perror("recvfrom failed");
   abort();
}
```

- `recvfrom()`: reads bytes from the socket and sets the source information
- Reading 0 bytes does not mean "connection closed" unlike TCP.
  - Recall UDP does not have a notion of "connection".

API functions Summary

TCP

- Initialization
  - socket(AF_INET, SOCK_STREAM, 0)
  - setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, ...)
  - bind()
  - listen()
- Conneciton
  - connect()
  - accept()
- Data transfer
  - send()
  - recv()
- Termination
  - close()

UDP

- Initialization
  - socket(AF_INET, SOCK_DGRAM, 0)
  - setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, ...)
  - bind()
- No connection
- Data transfer
  - sendto()
  - recvfrom()
- Termination
  - close()

Outline

- Socket API motivation, background
- Types of sockets (TCP vs. UDP)
- Elementary API functions
- I/O multiplexing
- Project 1 – tiny World of Warcraft

How to handle multiple inputs?

- Data sources
  - Standard input (e.g., keyboard)
  - Multiple sockets
- Problem: asynchronous data arrival
  - Program does not know when it will arrive.
- If no data available, `recv()` blocks.
- If blocked on one source, cannot handle other sources
  - Suppose what if a web server cannot handle multiple connections
- Solutions
  - Polling using non-blocking socket ➔ Inefficient
  - I/O multiplexing using select() ➔ simple
  - Multithreading ➔ more complex. Not covered today

Polling using non-blocking socket

- This approach wastes CPU cycles

```
int opt = fcntl(sock, F_GETFL);
if (opt < 0) {
    perror("fcntl failed");
    abort();
}
if (fcntl(sock, F_SETFL, opt | O_NONBLOCK) < 0) {
    perror("fcntl failed");
    abort();
}
while (1) {
    int read_bytes = recv(sock, buffer, sizeof(buffer), 0);
    if (read_bytes < 0) {
        if (errno == EWOULDBLOCK) {
            // OK. Simply no data
        } else {
            perror("recv failed");
            abort();
        }
}
```

Gets the socket's option

Updates the socket's option with non blocking option

When no data, we see EWOULDBLOCK error code.

I/O multiplexing using `select()`

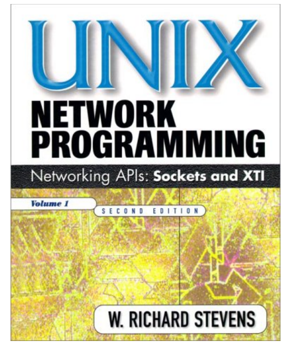
```
fd_set read_set;
struct timeval timeout

FD_ZERO(&read_set);
FD_SET(sock1, &read_set);
FD_SET(sock2, &read_set);
timeout.tv_sec = 0;
timeout.tv_usec = 5000;

if (select(MAX(sock1, sock2) + 1, &read_set, NULL,
           NULL, &time_out) < 0) {
   perror("select failed");
   abort();
}

if (FD_ISSET(sock1, &read_set)) {
   // sock1 has data
}
if (FD_ISSET(sock2, &read_set)) {
   // sock2 has data
}
```

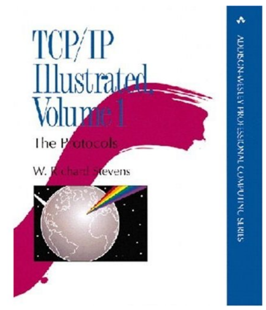
Initializes arguments for select()

Pass NULL instead of &timeout if you want to wait indefinitely

Checks I/O events.

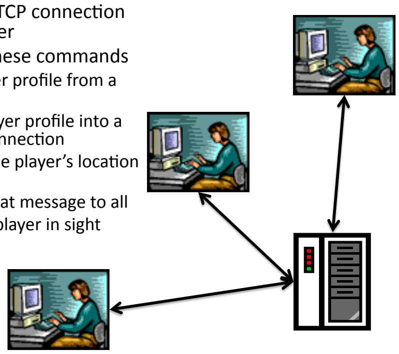
Bibles – both by W. Richard Stevens





Project 1 – *tiny* World of Warcraft

- Game client forms TCP connection with the game server
- It should support these commands
  - Login: loads player profile from a file
  - Logout: saves player profile into a file, closes the connection
  - Move: updates the player's location in the game
  - Speak: sends a chat message to all
  - Attack: attacks a player in sight



Project 1 – *tiny* World of Warcraft

- Divided into 2 parts:
  - Part 1: develop a game client
    - Message formats and commands will be given
    - Can test your client on provided reference server
  - Part 2: develop a game server
    - It should work with your client

Appendix – Programming Tips

- Will not be covered during the lecture
- Please refer to these tips if you're interested

Tip #1

- How to check the host byte order of my machine?

```
union {
  uint16_t number;
  uint8_t  bytes[2];
} test;
test.number = 0x0A0B;
printf("%02x%02x\n", test.bytes[0],
                     test.bytes[1]);
```

Tip #2

- How to get IP address from host name
  - Use **gethostbyname()**

```
struct sockaddr_in sin;
struct hostent *host;
host = gethostbyname(“www.berkeley.edu”);
sin.sin_addr.s_addr
  = *(unsigned *) host->h_addr_list[0];
```

Tip #3

- By default, Unix terminates the process with **SIGPIPE** if you write to a TCP socket which has been closed by the other side. You can disable it by:

```
signal(SIGPIPE, SIG_IGN);
```

Tip #4 - Structure Packing

- We have the following application-level packet header format (the numbers denote field size in bytes)

length	type	source addr	dest addr
2	1	4	4

- So, we define the header as struct like this:

```
struct my_pkt_hdr {
   unsigned short length;
   unsigned char type;
   unsigned int source_addr;
   unsigned int dest_addr;
};
```

- Q) Result of sizeof(struct my_pkt_hdr)?

Tip #4 - Structure Packing (cont’d)

- Compiler will try to be 4-byte aligned (on 32bit machines)
- To avoid the previous case, we must pack struct

Windows programming style

```
#pragma pack(push, 1)
struct my_pkt_hdr {
     unsigned short length;
     unsigned char type;
     unsigned int source_addr;
     unsigned int dest_addr;
};
#pragma pack(pop)
```

OR

GCC style

```
struct my_pkt_hdr {
     unsigned short length;
     unsigned char type;
     unsigned int source_addr;
     unsigned int dest_addr;
} __attribute__((packed));
```

Using man pages

- Best source to study system calls and library functions
  - Tells which header files should be included
  - Describes how each function works
  - Tells what the return value means and what error number can happen
  - E.g., `man connect`