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)

- Thus, TCP needs application-level message boundary.
  - By carrying length in application-level header
  - E.g.
    ```
    struct my_app_hdr {
      int length;
    }
    ```

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
Error code in Unix programming

```c
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)`
    - `errno` is always looked for
  - `printf` prints “my str: error string”
- `const char *` ` sterror(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.

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

```c
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. (cf, struct sockaddr_in6)
- INADDR_ANY: if server has multiple IP addresses, binds any address.
- `htons` converts host byte order into network byte order.

Endianness

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

```
+-----------------+-----------------+
| Increasing addr |       Big Endian       |
| 0x0A             | 0x0B             |
```

```
+-----------------+-----------------+
| Increasing addr |     Little Endian       |
| 0x0B             | 0x0A             |
```

- 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

```c
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
```

- Hint: `h, n, s, and i` 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.

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

```c
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. (cf, 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()`

```c
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.

```c
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, ...)

### Scenario #1 – TCP client-server

#### Sequence of actions

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket()</td>
<td>socket()</td>
</tr>
<tr>
<td>bind()</td>
<td>bind()</td>
</tr>
<tr>
<td>listen()</td>
<td>listen()</td>
</tr>
<tr>
<td>connect()</td>
<td>accept()</td>
</tr>
<tr>
<td>send() or write()</td>
<td>recv() or read()</td>
</tr>
<tr>
<td>recv() or read()</td>
<td>send() or write()</td>
</tr>
<tr>
<td>close()</td>
<td>close()</td>
</tr>
</tbody>
</table>

### Connection Establishment (client)

```c
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., 12.3.4 => 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)

```c
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
  - QJ Are they in Big endian or Little endian?
Scenario #1 – TCP client-server
• Sequence of actions

<table>
<thead>
<tr>
<th>Time</th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>socket()</td>
<td>socket()</td>
</tr>
<tr>
<td></td>
<td>bind()</td>
<td>bind()</td>
</tr>
<tr>
<td></td>
<td>listen()</td>
<td>listen()</td>
</tr>
<tr>
<td></td>
<td>connect()</td>
<td>accept()</td>
</tr>
<tr>
<td></td>
<td>send() or write()</td>
<td>recv() or read()</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>recv() or read()</td>
<td>send() or write()</td>
</tr>
<tr>
<td></td>
<td>close()</td>
<td>close()</td>
</tr>
</tbody>
</table>

Sending Data: server+client, send()

```c
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()

```c
int 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()

```c
// 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
• What must be changed?

<table>
<thead>
<tr>
<th>Time</th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>socket()</td>
<td>socket()</td>
</tr>
<tr>
<td></td>
<td>bind()</td>
<td>bind()</td>
</tr>
<tr>
<td></td>
<td>listen()</td>
<td>listen()</td>
</tr>
<tr>
<td></td>
<td>connect()</td>
<td>accept()</td>
</tr>
<tr>
<td></td>
<td>send() or write()</td>
<td>recv() or read()</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>recv() or read()</td>
<td>send() or write()</td>
</tr>
<tr>
<td></td>
<td>close()</td>
<td>close()</td>
</tr>
</tbody>
</table>

Scenario #2 – UDP client-server
• We need a different initialization

<table>
<thead>
<tr>
<th>Time</th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>socket()</td>
<td>socket()</td>
</tr>
<tr>
<td></td>
<td>bind()</td>
<td>bind()</td>
</tr>
<tr>
<td></td>
<td>listen()</td>
<td>listen()</td>
</tr>
<tr>
<td></td>
<td>connect()</td>
<td>accept()</td>
</tr>
<tr>
<td></td>
<td>send() or write()</td>
<td>recv() or read()</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>recv() or read()</td>
<td>send() or write()</td>
</tr>
<tr>
<td></td>
<td>close()</td>
<td>close()</td>
</tr>
</tbody>
</table>
Initialization: UDP

```c
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

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

![Scenario #2 – UDP client-server](image)

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

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

![Scenario #2 – UDP client-server](image)
Send Data Over UDP: `sendto`()

```c
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);

int 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.
- As opposed to TCP, UDP packetizes data. So, `sendto` () sends all data or nothing.

Receive Data Over UDP: `recvfrom`()

```c
struct sockaddr_in sin;

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)`
  - `bind()`
  - `listen()`
- Connection
  - `connect()`
  - `accept()`
- Data transfer
  - `send()`
  - `recv()`
- Termination
  - `close()`

**UDP**
- Initialization
  - `socket(AF_INET, SOCK_DGRAM, 0)`
  - `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

```c
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();
    }
  }
}
```

- This approach wastes CPU cycles
I/O multiplexing using `select()`

```c
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
}
```

---

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

---

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?

```c
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()`

```c
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:

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

Tip #4 - Structure Packing

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

<table>
<thead>
<tr>
<th>length</th>
<th>type</th>
<th>source_addr</th>
<th>dest_addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

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

```c
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**

```c
#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)
```

**GCC style**

```c
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`