Reference

Server & Client

TCP/UDP

IP

Ethernet Adapter

Socket API

TCP/UDP

IP

Ethernet Adapter

ports

Clients

Server

user space

hardware

user space

kernel space
Server & Client
What is Socket?

- Service access point of TCP/IP protocol stacks
- Socket is an interface between Application layer and Transport layer
What is Socket?

- A socket is a file descriptor that lets an application read/write data from/to the network.
- Once configured, the application can:
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network by some other host)
TCP Client & Server

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

TCP Server
- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `read()`
- `write()`
- `read()`
- `close()`

**Steps:**
- **Connection Establishment**: `socket()` -> `bind()` -> `listen()` -> `accept()`
- **Data Request**: `write()` -> `read()`
- **Data Reply**: `write()`
- **End-of-File Notification**: `read()` -> `close()`
int s = socket(domain, type, protocol);

- **s**: socket descriptor
- **domain**: integer, communication domain
  - e.g., AF_INET (IPv4 protocol) - typically used
- **type**: communication type
  - **SOCK_STREAM**: reliable, 2-way, connection-based service (TCP)
  - **SOCK_DGRAM**: unreliable, connectionless (UDP)
- **protocol**: specifies protocol (see file /etc/protocols for a list of options) - usually set to 0
socket()

- Ex.

```c
int s;      /* socket descriptor */

if((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    exit(1);
}
```
int status = bind(sockid, &addrport, size);

- **status**: error status, = -1 if bind failed
- **sockid**: integer, socket descriptor
- **addrport**: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY – chooses a local address)
- **size**: the size (in bytes) of the addrport structure
bind()

Ex.

```c
int s;  /* socket descriptor */
struct sockaddr_in srv;  /* used by bind() */

/* create the socket */

srv.sin_family = AF_INET;  /* use the Internet addr family */
srv.sin_port = htons(8888);  /* bind socket ‘fd’ to port 8888*/

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(s, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
    perror("bind"); exit(1);
}
```
listen()

- int status = listen(sock, queuelen);
  - status: 0 if listening, -1 if error
  - sock: integer, socket descriptor
  - queuelen: integer, # of active participants that can “wait” for a connection
  - listen is **non-blocking**: returns immediately
listen()

Ex.

```c
int s;       /* socket descriptor */
struct sockaddr_in srv; /* used by bind() */

/* 1) create the socket */
/* 2) bind the socket to a port */

if(listen(s, 5) < 0) {
    perror("listen");
    exit(1);
}
```
int s = accept(sock, &name, &namelen);

- **s**: integer, the new socket (used for data-transfer)
- **sock**: integer, the orig. socket (being listened on)
- **name**: struct sockaddr, address of the active participant
- **namelen**: sizeof(name): value/result parameter
  - must be set appropriately before call
  - adjusted by OS upon return
- **accept is blocking**: waits for connection before returning
**Ex.**

```c
int s; /* socket descriptor */
struct sockaddr_in srv; /* used by bind() */
struct sockaddr_in cli; /* used by accept() */
int newfd; /* returned by accept() */
int cli_len = sizeof(cli); /* used by accept() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */

newfd = accept(s, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
    perror("accept"); exit(1);
}
```
connect()  

- int status = connect(sock, &name, namelen);
  - status: 0 if successful connect, -1 otherwise
  - sock: integer, socket to be used in connection
  - name: struct sockaddr: address of passive participant
  - namelen: integer, sizeof(name)

- connect is **blocking**
Ex.

```c
int s; /* socket descriptor */
struct sockaddr_in srv; /* used by connect() */

/* create the socket */

/* connect: use the Internet address family */
srv.sin_family = AF_INET;

/* connect: socket ‘s’ to port 8888 */
srv.sin_port = htons(8888);

/* connect: connect to IP Address “140.112.1.1” */
srv.sin_addr.s_addr = inet_addr("140.112.1.1");

if(connect(s, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
    perror("connect"); exit(1);
}
```
Ex.

```
int s;           /* socket descriptor */
char buf[512];   /* used by read() */
int nbytes;      /* used by read() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
/* 4) accept the incoming connection */

if((nbytes = read(s, buf, sizeof(buf))) < 0) {
    perror("read"); exit(1);
}
```

- **read** blocks waiting for data from the client but does not guarantee that `sizeof(buf)` is read
**write()**

**Ex.**

```c
int s; /* socket descriptor */
char buf[512]; /* used by write() */
int nbytes; /* used by write() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
/* 4) accept the incoming connection */

if((nbytes = write(s, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```
Address and port are stored as integers

- Problem:
- different machines / OS’s use different word orderings
  - little-endian: lower bytes first
  - big-endian: higher bytes first
- these machines may communicate with one another over the network
Byte Ordering Solution

- htons(): “Host to Network Short”
- htonl(): “Host to Network Long”
- ntohs(): “Network to Host Short”
- ntohl(): “Network to Host Long”
Other Useful Function

- void bzero(void *dest, size_t nbytes);
- void bcopy(const void *src, void *dest, size_t nbytes);
- int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
- void *memset(void *dest, int c, size_t nbytes);
- void *memcpy(void *dest, const void *src, size_t nbytes);
- int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
Other Useful Function

- `int gethostname(char *name, int len)`
- `struct hostent * gethostbyaddr(char *addr, int len, int type)`
- `in_addr_t inet_addr(const char *cp)`
- `char * inet_ntoa(const struct in_addr in)`

Make sure to `#include` the header files, check it yourself !!!
HW1: Echo Server & Client

- Your job is to implement an Echo Server which will send back whatever it receives.
- You also have to write a program which will send message to the Echo Server.

![Diagram of Client and Server interactions]

- Establish TCP Connection
- "Hello World"
- "Hello World"
- Terminate Connection
HW1: Echo Server & Client

- **Echo Server execution format**

  ```
  EchoServer server_port
  ```

  - While someone connects to the server, server should output some information

  ```
  Client_IP[140.112.30.1]
  Client_Port[54321]
  Message_Received[Hello World]
  Message_Sendback[Hello World]
  ```

- **Client execution format**

  ```
  EchoClient server_ip server_port message
  ```
HW1: Echo Server & Client

- Your program must be gcc 3.3.1 compatible
- Name your program in this way
  - b91902xxx_s.cc for Echo Server
  - b91902xxx_c.cc for Client
- Make your program as a tarball
  - `tar zcvf b91902xxx_hw1.tar.gz b91902xxx_s.cc b91902xxx_c.cc`
- Email the tarball to
  - network_hw@voip.csie.ntu.edu.tw
    - Subject:[network hw1]b91902xxx name
- Due after two weeks (3/30 PM 2:20)