Socket Programming

Ming-Hung Chen
2010/03/24

Overview

• Background
  • What is Socket?
  • Socket API (Application Programming Interface)
  • TCP (Transmission Control Protocol)
  • UDP (User Datagram Protocol)

• Socket Programming
  • Network Addressing
  • Byte Ordering
  • TCP Server and Client
  • UDP Server and Client

What is Socket?
From the point of view of the networking

• Service access point of TCP/IP protocol stack
  • between Application layer and Transport layer

• A file descriptor that lets an application read/write data from/to the network

• Once configured the application can
  • Send data to the socket
  • Receive data from the socket

What is File Descriptor?

• E.g. cout writes data to stdout (#1)
  cin read data from stdin (#0)

Socket API

Server and Client exchange messages over the network through a common Socket API.

TCP
- Reliable – guarantee delivery
- Byte stream – in-order delivery
- Connection-oriented – single socket per connection
- Setup connection followed by data transfer

Telephone Call
- Guaranteed delivery
- In-order delivery
- Connection-oriented
- Setup connection followed by conversation

User Datagram Protocol (UDP)

Example TCP applications
Web, Email, Telnet

Example UDP applications
Multimedia, voice over IP

TCP vs. UDP
- Transmission Control Protocol (TCP)
  - One-to-one
  - Connection-oriented
  - Reliable
  - In-order delivery
  - Transmission after Connect

- User Datagram Protocol (UDP)
  - One-to-one or one-to many
  - Connectionless
  - Unreliable
  - Unordered delivery
  - Transmission with destination address
Overview

- Background
  - What is Socket?
  - Socket API (Application Programming Interface)
  - TCP (Transmission Control Protocol)
  - UDP (User Datagram Protocol)
- Socket Programming
  - Network Addressing
  - Byte Ordering
  - TCP Server and Client
  - UDP Server and Client

Concept of Port Numbers

- Port numbers are used to identify "entities" on a host
- Port numbers can be
  - Well-known (port 0-1023)
  - Dynamic or private (port 1024-65535)
- Servers/daemons usually use well-known ports
  - Any client can identify the server/service
  - HTTP = 80, FTP = 21, Telnet = 23, ...
  - /etc/service defines well-known ports
- Clients usually use dynamic ports
  - Assigned by the kernel at run time

Network Addressing Analogy

Telephone Call
- Professors at CMU
  - 412-268-8000 ext. 123
  - 412-268-8000 ext. 554
- Extension
- Telephone No
- Central Number
- Exchange
- Area Code
- 15-441 Students

Network Programming
- Applications/Servers
  - Web Port 80
  - Mail Port 25
- Port No.
- IP Address
  - Network No.
  - Host Number
- Clients

Names and Addresses

- Each attachment point on Internet is given unique address
  - Based on location within network – like phone numbers
- Humans prefer to deal with names not addresses
  - DNS provides mapping of name to address
  - Name based on administrative ownership of host
Address Translating Functions

<table>
<thead>
<tr>
<th>Domain Name</th>
<th>DNS Server</th>
<th>gethostbyname()</th>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(hostent.h_addr.s_addr -&gt; sockaddr_in.sin_addr.s_addr)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Service Name</th>
<th>/etc/service</th>
<th>getservbyname()</th>
<th>Port 0~65535</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(servent.s_port -&gt; sockaddr_in.s_port)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IP Address</th>
<th>inet_addr()</th>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(string)</td>
<td></td>
<td>(u_long)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IP Address</th>
<th>inet_ntoa()</th>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u_long)</td>
<td></td>
<td>(string)</td>
</tr>
</tbody>
</table>

Translating Names to Addresses

- Gethostbyname provides interface to DNS
- Additional useful calls
  - Gethostbyaddr – returns hostent given sockaddr_in
  - Getservbyname
    - Used to get service description (typically port number)
    - Returns servent based on name

```c
#include <netdb.h>

struct hostent *hp; /*ptr to host info for remote*/
struct sockaddr_in peeraddr;
char *name = "www.cs.cmu.edu";
peeraddr.sin_family = AF_INET;
if((hp = gethostbyname(name))!=NULL)
{
    peeraddr.sin_addr.s_addr = ((struct in_addr*)(hp->h_addr))->s_addr;
    printf("Translate %s => %s\n", name, inet_ntoa(peeraddr.sin_addr));
}
```

Dealing with IP Addresses

- IP Addresses are commonly written as strings ("128.2.35.50"), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```c
struct sockaddr_in srv;

srv.sin_addr.s_addr = inet_addr("128.2.35.50");
if(srv.sin_addr.s_addr == (in_addr_t) -1) {
    fprintf(stderr, "inet_addr failed!\n"); exit(1);
}
```

Converting a numerical address to a string:

```c
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
if(t == 0) {
    fprintf(stderr, "inet_ntoa failed!\n"); exit(1);
}
```

Internet Addressing Data Structure

```c
#include <netinet/in.h>

/* Internet address structure */
struct in_addr {
    u_long s_addr; /* 32-bit IPv4 address */
}; /* network byte ordered */

/* Socket address, Internet style. */
struct sockaddr_in {
    u_char sin_family; /* Address Family */
    u_short sin_port; /* UDP or TCP Port# */
    struct in_addr sin_addr; /* Internet Address */
    char sin_zero[8]; /* unused */
};
```

- sin_family = AF_INET selects Internet address family
Byte Ordering

union {
    u_int32_t addr; /* 4 bytes address */
    char c[4];
} un;
/* 128.2.194.95 */
un.addr = 0x8002c25f;
/ * c[0] = ? */

- Big Endian
  - Sun Solaris, PowerPC, ...
- Little Endian
  - i386, alpha, ...
- Network byte order = Big Endian

Byte Ordering Functions

- Converts between host byte order and network byte order
  - ‘h’ = host byte order
  - ‘n’ = network byte order
  - ‘l’ = long (4 bytes), converts IP addresses
  - ‘s’ = short (2 bytes), converts port numbers

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Recall: What is a Socket?

- A socket is a file descriptor that lets an application read/write data from/to the network

```c
int fd;    /* socket descriptor */
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) 
  perror("socket");
  exit(1);
```

- `socket` returns an integer (socket descriptor)
  - `fd < 0` indicates that an error occurred
  - socket descriptors are similar to file descriptors

- AF_INET: associates a socket with the Internet protocol family
- SOCK_STREAM: selects the TCP protocol
- SOCK_DGRAM: selects the UDP protocol

TCP Server

- For example:
  - web server

- What does a web server need to do so that a web client can connect to it?
Socket I/O: socket()

- Since web traffic uses TCP, the web server must create a socket of type SOCK_STREAM

```c
int fd; /* socket descriptor */
if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    exit(1);
}
```

- `socket` returns an integer (socket descriptor)
  - `fd < 0` indicates that an error occurred
- `AF_INET` associates a socket with the Internet protocol family
- `SOCK_STREAM` selects the TCP protocol

Socket I/O: bind()

- A `socket` can be bound to a `port`

```c
int fd; /* 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(80); /* bind socket 'fd' to port 80*/
/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(fd, (struct sockaddr*)&srv, sizeof(srv)) < 0) {
    perror("bind"); exit(1);
}
```

- Still not quite ready to communicate with a client...

Socket I/O: listen()

- `listen` indicates that the server will accept a connection

```c
int fd; /* socket descriptor */
struct sockaddr_in srv; /* used by bind() */
/* 1) create the socket */
/* 2) bind the socket to a port */
if(listen(fd, 5) < 0) {
    perror("listen");
    exit(1);
}
```

- Still not quite ready to communicate with a client...

• `accept` blocks waiting for a connection

```c
int fd; /* 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(fd, (struct sockaddr*)&cli, &cli_len);
if(newfd < 0) {
    perror("accept");
    exit(1);
}
```

- `accept` returns a new socket (`newfd`) with the same properties as the original socket (`fd`)
  - `newfd < 0` indicates that an error occurred
Socket I/O: accept() continued...

```
struct sockaddr_in cli; /* used by accept() */
int newfd; /* returned by accept() */
int cli_len = sizeof(cli); /* used by accept() */

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

- How does the server know which client it is?
  - cli.sin_addr.s_addr contains the client’s IP address
  - cli.sin_port contains the client’s port number
- Now the server can exchange data with the client by using `read` and `write` on the descriptor `newfd`.
- Why does accept need to return a new descriptor?

TCP Client

- For example: web client

```
2 Web Clients
TCP
IP
Ethernet Adapter
```

- How does a web client connect to a web server?

Socket I/O: connect()

```
int fd; /* 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 ‘fd’ to port 80 */
srv.sin_port = htons(80);

/* connect: connect to IP Address "128.2.35.50" */
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

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

- `connect` allows a client to connect to a server...

```
int fd; /* 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(newfd, buf, sizeof(buf))) < 0) {
    perror("read"); exit(1);
}
```

- `read` can be used with a socket
- `read` blocks waiting for data from the client but does not guarantee that sizeof(buf) is read
Socket I/O: write()

- **write** can be used with a socket

```c
int fd; /* socket descriptor */
struct sockaddr_in srv; /* used by connect() */
char buf[512]; /* used by write() */
int nbytes; /* used by write() */

/* 1) create the socket */
/* 2) connect() to the server */

/* Example: A client could “write” a request to a server */
if((nbytes = write(fd, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```

UDP Server Example

- For example: NTP daemon

```
NTP daemon

Port 123

UDP

IP

Ethernet Adapter
```

Review: TCP Client-Server Interaction

Socket I/O: socket()

- The UDP server must create a **datagram** socket...

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

if((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("socket");
    exit(1);
}
```

- **socket** returns an integer (**socket descriptor**) if fd < 0 indicates that an error occurred

- AF_INET: associates a socket with the Internet protocol family
- **SOCK_DGRAM**: selects the UDP protocol
Socket I/O: bind()

- A socket can be bound to a port

```c
int fd; /* socket descriptor */
struct sockaddr_in srv; /* used by bind() */
/* create the socket */
/* bind: use the Internet address family */
srv.sin_family = AF_INET;
/* bind: socket 'fd' to port 80*/
srv.sin_port = htons(80);
/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
    perror("bind"); exit(1);
}
```

- Now the UDP server is ready to accept packets...

Socket I/O: recvfrom() continued...

```c
nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
    (struct sockaddr*) cli, &cli_len);
```

- The actions performed by recvfrom
  - returns the number of bytes read (nbytes)
  - copies nbytes of data into buf
  - returns the address of the client (cli)
  - returns the length of cli (cli_len)
  - don't worry about flags

Socket I/O: recvfrom()

- read does not provide the client's address to the UDP server

```c
int fd; /* socket descriptor */
struct sockaddr_in srv; /* used by bind() */
struct sockaddr_in cli; /* used by recvfrom() */
char buf[512]; /* used by recvfrom() */
int cli_len = sizeof(cli); /* used by recvfrom() */
int nbytes; /* used by recvfrom() */
/* 1) create the socket */
/* 2) bind to the socket */
nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
    (struct sockaddr*) &cli, &cli_len);
if(nbytes < 0) {
    perror("recvfrom"); exit(1);
}
```

UDP Client Example

- How does a UDP client communicate with a UDP server?
Socket I/O: sendto()

- **write** is not allowed
- Notice that the UDP client does not **bind** a port number
  - A port number is **dynamically assigned** when the first **sendto** is called

```c
int fd; /* socket descriptor */
struct sockaddr_in srv; /* used by sendto() */
/* 1) create the socket */

/* sendto: send data to IP Address "128.2.35.50" port 80 */
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("128.2.35.50");
nbytes = sendto(fd, buf, sizeof(buf), 0, /* flags */,
(struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0) {
    perror("sendto"); exit(1);
}
```

Review: UDP Client-Server Interaction

---

UDP Server: Servicing Two Ports

```c
int s1; /* socket descriptor 1 */
int s2; /* socket descriptor 2 */

/* 1) create socket s1 */
/* 2) create socket s2 */
/* 3) bind s1 to port 2000 */
/* 4) bind s2 to port 3000 */
while(1) {
    recvfrom(s1, buf, sizeof(buf), ...);
    /* process buf */
    recvfrom(s2, buf, sizeof(buf), ...);
    /* process buf */
}
```

---

The UDP Server

- How can the **UDP server** service multiple ports simultaneously?
Socket I/O: select()

```c
int select(int maxfds, fd_set *readfds, fd_set *writefds,
          fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *fds);   /* clear the bit for fd in fds */
FD_ISSET(int fd, fd_set *fds); /* is the bit for fd in fds? */
FD_SET(int fd, fd_set *fds);   /* turn on the bit for fd in fds */
FD_ZERO(fd_set *fds);          /* clear all bits in fds */
```

- **maxfds**: number of descriptors to be tested
  - descriptors (0, 1, ... maxfds-1) will be tested
- **readfds**: a set of fds we want to check if data is available
  - returns a set of fds ready to read
  - if input argument is NULL, not interested in that condition
- **writefds**: returns a set of fds ready to write
- **exceptfds**: returns a set of fds with exception conditions

**Socket I/O: select()**

```c
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};
```

- **timeout**
  - if NULL, wait forever and return only when one of the descriptors is ready for I/O
  - otherwise, wait up to a fixed amount of time specified by timeout
    - if we don’t want to wait at all, create a timeout structure with timer value equal to 0

- Refer to the man page for more information

More Details About a Web Server

**select** allows synchronous I/O multiplexing

```c
int s1, s2; /* socket descriptors */
fd_set readfds; /* used by select() */
fd_set writefds; /* used by select() */
fd_set exceptfds; /* used by select() */
struct timeval *timeout = NULL;

/* create and bind s1 and s2 */
while(1) {
    FD_ZERO(&readfds); /* initialize the fd set */
    FD_SET(s1, &readfds); /* add s1 to the fd set */
    FD_SET(s2, &readfds); /* add s2 to the fd set */

    if(select(s1+1, &readfds, 0, 0, timeout) < 0) {
        perror("select");
        exit(1);
    }
    if(FD_ISSET(s1, &readfds)) {
        recvfrom(s1, buf, sizeof(buf), ...);
        /* process buf */
    }
    /* do the same for s2 */
}
```

How can a web server manage multiple connections simultaneously?

- Port 80
- Port 8001
- TCP
- IP
- Ethernet Adapter
Socket I/O: select()

```c
int fd, next = 0; /* original socket */
int newfd[10]; /* new socket descriptors */
while(1) {
    fd_set readfds;
    FD_ZERO(&readfds); FD_SET(fd, &readfds);
    /* Now use FD_SET to initialize other newfd’s */
    select(maxfd+1, &readfds, 0, 0, 0);
    if(FD_ISSET(fd, &readfds)) {
        newfd[next++] = accept(fd, ...);
    }
    /* do the following for each descriptor newfd[n] */
    if(FD_ISSET(newfd[n], &readfds)) {
        read(newfd[n], buf, sizeof(buf));
        /* process data */
    }
}
```

• Now the web server can support multiple connections...

Other Solutions for Concurrent Client/Server?

• What we talked above is the single-process concurrent solution, and we still have other choices

  • Multi-thread
    • See pthread family manual, e.g. pthread_create().etc
    • Pthread mutex helps in data synchronizing
      • E.g. pthread_mutex_lock(), pthread_mutex_init().etc

  • Multi-process
    • See fork(), vfork(). If you don’t know which one to use, use fork()
    • Share memory for communication between processes
      • E.g. shmget(), shmat(), shmdt().etc
    • Semaphore to helps in data synchronizing
      • See semaphore in your OS textbook
      • E.g. semget(), semct1(), semop().etc

E.g. Concurrent Server using fork()

```
        pid_t pid;
        int listenfd, connfd;
        listenfd = socket(...); /* fill in sockaddr_in() with server’s well known port */
        bind(listenfd, ...);
        listen(listenfd, LISTENQ);
        while(1) {
            connfd = accept(listenfd, ...); /* probably blocks*/
            if( (pid = fork()) == 0) {
                close(listenfd);  /* child closes listening socket*/
                doit(connfd);     /* process the request */
                close(connfd);    /* done with this client */
                exit(0);          /* child terminates */
            }
            close(connfd); /* parent closes connected socket */
        }
```

E.g. Concurrent Server using fork()
Reference

• Books
  • Advanced Programming in the UNIX(R) Environment (2nd Edition) (Addison-Wesley Professional Computing Series) (Hardcover)
    • W. Richard Stevens, Stephen A. Rago
  • UNIX Network Programming, Volume 2: Interprocess Communications (2nd Edition) (Hardcover)
    • W. Richard Stevens
  • Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version
    • Douglas E. Comer, David L. Stevens, and Michael Evangelista

• man
  • man 2 (socket | bind | listen | read | write | connect | …)
  • man XXXXX

• Google