Lecture - XV
Network Programming - I

Tevfik Koşar
Louisiana State University
November 9th, 2010

Sockets

- A Socket is comprised of:
  - a 32-bit node address (IP address)
  - a 16-bit port number (like 7, 21, 13242)

- Example: 192.168.31.52:1051
  - The 192.168.31.52 host address is in “IPv4 dotted-quad” format, and is a decimal representation of the hex network address 0xc0a81f34

- First developed at UC-Berkeley in 1983, Berkeley Socket API part of BSD 4.2

Ports

- Ports 0 through 1023 are reserved, privileged ports, defined by TCP and UDP well known port assignments
- Ports 1024 through 49151 are ports registered by the IANA (Internet Assigned Numbers Authority), and represent second tier common ports (socks (1080), WINS (1512), kermit (1649))
- Ports 49152 through 65535 are ephemeral ports, available for temporary client usage

Network Communication

Data Encapsulation

- Application puts data out through a socket
- Each successive layer wraps the received data with its own header:
TCP Header Format

- Source and Destination ports
- Sequence Number tells what byte offset within the overall data stream this segment applies
- Acknowledgement number lets the recipient set what packet in the sequence was received OK.

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
<th>Sequence Number</th>
<th>Acknowledgement Number</th>
<th>Flags</th>
<th>Window Size</th>
<th>Checksum</th>
<th>Urgent Pointer</th>
<th>Options</th>
<th>Header Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>2 bytes</td>
<td>4 bytes</td>
<td>8 bytes</td>
<td>1 byte</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td>4 bytes</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

IP Header Format

- Packets may be broken up, or fragmented, if original data is too large for a single packet (Maximum Transmission Unit is currently 12k bits, or 1500 Bytes)
- Packets have a Time To Live, number of seconds/rounds it can bounce around aimlessly among routers until it’s killed.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Source Port</th>
<th>Destination Port</th>
<th>Identification</th>
<th>Flags</th>
<th>Fragment Offset</th>
<th>Fragment Length</th>
<th>Time To Live</th>
<th>Protocol ID</th>
<th>Options</th>
<th>Header Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td>1 byte</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td>1 byte</td>
<td>1 byte</td>
<td>2 bytes</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

Common Network Applications

- FTP (file transfer protocol)
- SMTP (simple mail transfer protocol)
- telnet (remote login)
- rlogin (simple remote login between UNIX machines)
- World Wide Web (built on http)
- NFS (network filing system – originally for SUNs)
- TFTP (trivial file transfer protocol – used for booting)
- SNMP (simple network management protocol)

Well Known Services & Ports

<table>
<thead>
<tr>
<th>Service</th>
<th>Port no</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>echo</td>
<td>7</td>
<td>UDP/TCP</td>
</tr>
<tr>
<td>discard</td>
<td>9</td>
<td>UDP/TCP</td>
</tr>
<tr>
<td>daytime</td>
<td>13</td>
<td>UDP/TCP</td>
</tr>
<tr>
<td>chargen</td>
<td>19</td>
<td>UDP/TCP</td>
</tr>
<tr>
<td>ftp</td>
<td>21</td>
<td>TCP</td>
</tr>
<tr>
<td>telnet</td>
<td>23</td>
<td>TCP</td>
</tr>
<tr>
<td>smtp</td>
<td>25</td>
<td>TCP</td>
</tr>
<tr>
<td>Daytime</td>
<td>37</td>
<td>UDP/TCP</td>
</tr>
<tr>
<td>tftp</td>
<td>69</td>
<td>UDP</td>
</tr>
<tr>
<td>finger</td>
<td>79</td>
<td>TCP</td>
</tr>
<tr>
<td>http</td>
<td>80</td>
<td>TCP</td>
</tr>
<tr>
<td>login</td>
<td>513</td>
<td>TCP</td>
</tr>
<tr>
<td>who</td>
<td>513</td>
<td>UDP</td>
</tr>
<tr>
<td>Nnserver</td>
<td>6965</td>
<td>TCP</td>
</tr>
</tbody>
</table>

TCP & UDP

TCP
- connection based
- reliable
- byte stream
- used in: FTP, telnet, http, SMTP

UDP
- connectionless
- unreliable
- datagram (packet based)
- used in: NFS, TFTP

An HTTP Request

- <command> <argument> <HTTP version>
- <optional arguments>
- <blank line>

- GET /index.html HTTP/1.0
Server Response

- HTTP version <status code> <status message>
- <aditional information>
- <a blank line>
- <content>

HTTP/1.1 200 OK
Date: Thu, 06 Nov 2008 18:27:13 GMT
Server: Apache

Example

$ telnet www.cnn.com 80
Trying 64.236.90.21...
Escape character is "^]".
GET /index.html HTTP/1.0

HTTP/1.1 200 OK
Date: Thu, 06 Nov 2008 18:27:13 GMT
Server: Apache
Accept-Ranges: bytes
Cache-Control: max-age=60, private
Expires: Thu, 06 Nov 2008 18:28:14 GMT
Content-Type: text/html
Vary: Accept-Encoding,User-Agent
Connection: close

Basics of a Server (Web, FTP ..etc)

1. Listen to a Network port
2. Interpret incoming messages (requests)
3. Serve requests
   a. Read requested files
   b. Send them over network
4. Run consistently in the background (daemon process)

TCP Connection: Initial State

TCP Connection: Passive Open

TCP Connection: Active Open

- server process does a 'passive' open on a port
- it waits for a client to connect
- at this stage there is no Internet network traffic
- tells the TCP layer which process to connect to
TCP Connection: Rendezvous

- server side accepts and TCP connection established
- a bi-directional reliable byte-stream
- connection identified by both host/port numbers
  e.g. 151.10017.25.2397/81.112.182.21
- server port is not consumed
  can stay ‘passive’ open for more connections
- like telephone call desk: one number many lines

TCP Client-Server view
- Connection-oriented socket connections

TCP Connection: more..

- other clients can connect to the same port
- state for connections in the client/server only
- no information needed in the network
  not like old style relay-based exchanges
- server can restrict access to specified host or port
- server can find out connected host/port

Server Side Socket Details

CLIENT

- TCP Client-Server view
- Connection-oriented socket connections

Example: A Time Server

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORTNUM 8824
#define oops(msg) { perror(msg) ; exit(1) ; }

int main(int argc, char **argv)
{
    char *buf;
    int sockfd, n;
    struct sockaddr_in server;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        oops("socket() failed");

    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(PORTNUM);

    if (bind(sockfd, (struct sockaddr *)&server, sizeof(server)) < 0)
        oops("bind() failed");

    listen(sockfd, 5);

    while (1)
    {
        int len = sizeof(server);
        struct sockaddr_in cliaddr;
        socklen_t clilen = len;

        int connfd = accept(sockfd,
                             (struct sockaddr *)&cliaddr,
                             &clilen);
        if (connfd < 0)
            oops("accept() failed");

        n = read(connfd, buf, size_t(buffer_size));
        if (n < 0)
            oops("read() failed");

        write(connfd, buf, n);
        if (n < 0)
            oops("write() failed");

        close(connfd);
    }
    return 0;
}
```

Client Side Socket Details

void main(int ac, char **av)
{
    struct  sockaddr_in saddr; /* build our address here */
    struct  hostent *hp;    /* this is part of our */
    char *hostname[256];    /* address */
    FILE *sock_fp;          /* use socket as stream */
    char *ctime();          /* convert secs to string */
    long thetime;           /* time and the val */

    gethostname( hostname , 256); /* where am I? */
    hp = gethostbyname( hostname ); /* get info about host */
    bzero( &saddr, sizeof(saddr) ); /* zero struct */
    /* fill in hostaddr */
    bcopy( hp->h_addr, &saddr.sin_addr, hp->h_length);
    saddr.sin_family = AF_INET ; /* fill in socket type */
    saddr.sin_port = htons(PORTNUM); /* fill in socket port */
    sock_id = socket( AF_INET, SOCK_STREAM, 0 );    /* get a socket */
    if (sock_id == -1 ) oops( "socket" ); /* error getting calls */
    if (bind(sock_id, &saddr, sizeof(saddr)) != 0 )/* bind it to */
        oops( "bind" );                         /* an address */
    if ( listen(sock_id, 1) != 0 ) oops( "listen" );

    while ( 1 ){
        sock_fd = accept(sock_id, NULL, NULL); /* wait for call */
        printf("** Server: A new client connected!");
        if (sock_fd == -1 )
            oops( "accept" ); /* error getting calls */
        sock_fp = fdopen(sock_fd,"w"); /* we'll write to the */
        if (sock_fp == NULL )
            oops( "fdopen" ); /* unless we can't */
        thetime = time(NULL);           /* get time */
        /* and convert to string */
        fprintf( sock_fp, "**************************************
" );
        fprintf( sock_fp, "** From Server: The current time is: ");
        fprintf( sock_fp, "%s", ctime(&thetime) );
        fprintf( sock_fp, "**************************************
" );
        fclose( sock_fp );              /* release connection */
        fflush(stdout); /* force output */
    }
}