Network Programming

- There is one way computers can communicate together
  - By sending network messages to each other
  - All other kinds of communications are built from network messages
- There is one way programs can send/receive network messages
  - Through sockets
  - All other communication paradigms are built from sockets
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
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>Datagram (THE DATA) (up to 12k bits)</th>
</tr>
</thead>
</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>Preamble</th>
<th>Length of data</th>
<th>Fragmentation Information (if it’s too big for an ethernet frame buffer)</th>
<th>Type Of Service</th>
<th>Time To Live</th>
<th>Protocol (TCP, UDP)</th>
<th>Checksum</th>
<th>Source Address (192.32.63.5)</th>
<th>Destination Address (192.32.65.1)</th>
<th>Options</th>
<th>Datagram (THE DATA) (up to 12k bits)</th>
</tr>
</thead>
</table>

2 bytes-4 bytes-8 bytes-1 byte-1 byte-2 bytes-4 bytes-4 bytes-variable-variable
Common Network Applications

- FTP  (file transfer protocol)
- SMTP  (simple mail transfer protocol)
- telnet  (remote logins)
- 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>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>echo</td>
<td>7</td>
<td>UDP/TCP</td>
<td>sends back what it receives</td>
</tr>
<tr>
<td>discard</td>
<td>9</td>
<td>UDP/TCP</td>
<td>throws away input</td>
</tr>
<tr>
<td>daytime</td>
<td>13</td>
<td>UDP/TCP</td>
<td>returns ASCII time</td>
</tr>
<tr>
<td>chargen</td>
<td>19</td>
<td>UDP/TCP</td>
<td>returns characters</td>
</tr>
<tr>
<td>ftp</td>
<td>21</td>
<td>TCP</td>
<td>file transfer</td>
</tr>
<tr>
<td>telnet</td>
<td>23</td>
<td>TCP</td>
<td>remote login</td>
</tr>
<tr>
<td>smtp</td>
<td>25</td>
<td>TCP</td>
<td>email</td>
</tr>
<tr>
<td>daytime</td>
<td>37</td>
<td>UDP/TCP</td>
<td>returns binary time</td>
</tr>
<tr>
<td>tftp</td>
<td>69</td>
<td>UDP</td>
<td>trivial file transfer</td>
</tr>
<tr>
<td>finger</td>
<td>79</td>
<td>TCP</td>
<td>info on users</td>
</tr>
<tr>
<td>http</td>
<td>80</td>
<td>TCP</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>login</td>
<td>513</td>
<td>TCP</td>
<td>remote login</td>
</tr>
<tr>
<td>who</td>
<td>513</td>
<td>UDP</td>
<td>different info on users</td>
</tr>
<tr>
<td>Xserver</td>
<td>6000</td>
<td>TCP</td>
<td>X windows (N.B. &gt;1023)</td>
</tr>
</tbody>
</table>
TCP & UDP

Both
- built on top of IP
- addressed using port numbers
  ⇒ process to process
    (on UNIX platforms)

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>
- <additional information>
- <a blank line>
- <content>

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

<HTML><HEAD><BODY> ....

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

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"> <html lang="en"> <head> <title>CNN.com 44
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 is connection based
  ... establishing it is a complex multistage process
- initially all machines are the same
- no special ‘server’ machines
- the difference is all in the software
TCP Connection: Passive 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: Active Open

- client process usually on a different machine
- performs an ‘active’ open on the port
- port number at the client end is needed
  usually automatic (e.g., 2397)
  but can be chosen
- network message → server machine
  requests connection
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. <151.100.17.25:2397/161.112.192.5:21>
- server port is not consumed
  can stay "passive" open for more connections
- like telephone call desk: one number many lines

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
TCP Client-Server view
- Connection-oriented socket connections

**Server Side Socket Details**

```c
int socket(int domain, int type, int protocol)
sockfd = socket(PF_INET, SOCK_STREAM, 0);

int bind(int sockfd, struct sockaddr *server_addr, socklen_t length)
bind(sockfd, &server, sizeof(server));

int listen(int sockfd, int num_queued_requests)
listen(sockfd, 5);

int accept(int sockfd, struct sockaddr *incoming_address, socklen_t length)
newfd = accept(sockfd, &client, sizeof(client)); /* BLOCKS */

int read(int sockfd, void * buffer, size_t buffer_size)
read(newfd, buffer, sizeof(buf));

int write(int sockfd, void * buffer, size_t buffer_size)
write(newfd, buffer, sizeof(buf));
```
Client Side Socket Details

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define PORTNUM 8824
#define oops(msg)      { perror(msg) ; exit(1) ; }

int socket(int domain, int type, int protocol)
socketf = socket(PF_INET, SOCK_STREAM, 0);

int connect(int sockfd, struct sockaddr *server_address, socklen_t length)
connect(sockfd, &server, sizeof(server));

int write(int sockfd, void * buffer, size_t buffer_size)
write(sockfd, buffer, sizeof(buffer));

int read(int sockfd, void * buffer, size_t buffer_size)
read(sockfd, buffer, sizeof(buffer));
```

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) ; }
```
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 */
    int slen,sock_id,sock_fd; /* line id, file desc */
    FILE *sock_fp; /* use socket as stream */
    char *ctime(); /* convert secs to string */
    long time(), 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" );

    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!\n");
        if ( sock_fd == -1 )
            oops( "accept" ); /* error getting calls */

        sock_fp = fdopen(sock_fd,"w"); /* we'll write to the */
        if ( sock_fp == NULL ) /* socket as a stream */
            oops( "fdopen" ); /* unless we can't */

        thetime = time(NULL); /* get time */
        /* and convert to string */
        fprintf( sock_fp, "*****************************\n");
        fprintf( sock_fp, "** From Server: The current time is: \"");
        fprintf( sock_fp, "%s", ctime(&thetime) );
        fprintf( sock_fp, "*****************************\n");

        fclose( sock_fp ); /* release connection */
        fflush(stdout); /* force output */
    }
}
Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), J. Kim (KAIST), A. Dix (Hiraeth), and J. Schaumann (SIT).