LECTURE: NETWORKING FOR VISUALIZATION

Chee Yap

November 6, 2001

1 INTRODUCTION

We are interested in visualization over networks. In this lecture, we provide some basic background to networking. Our main reference is Stevens [1].

Computer networking is a diverse topic without some abstraction. The commonly accepted abstract model for networks is the **OSI open systems interconnection Model** (which we simply call the **OSI model**). OSI stands for *International Standards Organization.* This model defines seven layers, which in increasing levels of abstraction are

1. physical
2. data link
3. network
4. transport
5. session
6. presentation
7. application

A simplified version of the OSI model with only 4 layers is seen in the next table:

- In Column 1, we merge the first two (calling it 'data link') and the last three (calling it 'process') layers of the original OSI model. We also invert the order of the layers.
- Column 2 of this table gives the names of the unit of data sent in the specified layer.
- Column 3 gives a typical application, TFTP ('Trivial File Transfer Protocol') and the names of the protocols used in each layer.
- Column 4 shows how N bytes of data at the process level is augmented with header (and trailer) bytes at each level.

Table 1: Simplified OSI Model

LAYER	DATA	PROTOCOLS	BYTE SIZE
process	messages	TFTP	N
transport	messages	UDP	N+4
network	packets	IP	N+4+8
data link	frames	Ethernet	N+4+8+20

1.1 Layering

Layering reduces complexity – each layer solving a specific set of issues. Each layer offering a "service" to the one above it. Here are some of the services of the layers in the simplified model:

- Data Link Layer: simple error checking, breaking up into frames and reassembling.
- Network Layer: Routing of packets, congestion control.
- Transport Layer: Maintains persistent connections for end-to-end communication.
- Process Layer: Network API.

A process in a particular layer is conceptually communicating with its **peers** in the same layer in other computers. Except for the lowest layer, this communication is actually directed to some process at the level below, which (again) conceptually communicates with its peer in the other computer.

Example In the TFTP example in the above table, the TFTP server and TFTP client are peers. Say the server wants to send N bytes of data to the client. However, what happens is that the TFTP server actually sends the N bytes **down** to the next layer. The UDP layer adds a 4 byte header to this data, and sends it down to the layer below. The IP layer adds another 8 bytes of header and sends it to the Ethernet layer which adds 20 more bytes of header and trailer. On the client side, the information is passed **up** from the Ethernet layer to the IP layer after stripping off the 20 bytes of header and trailer. This continues all the way up to the TFTP client. In short, most communication is between adjacent layers.

Why header and trailers? One use is to store the target address of the data (there will be intermediate targets in addition to the ultimate target). Another is the need for multiplexing and demultiplexing in each layer – remember that there may be several other protocols in each layer.

1.2 Network Terminology

Some general concepts need to be explained:

- Computer networks can be **circuit switched** or **packet switched**. The internet is mainly packet switched, and this is our focus.
- Within packet switched networks, we can still have two analogous forms of communications: **connection-oriented** (virtual circuit) or **connectionless** (datagram).
- TCP/IP is an example of the former. UDP/IP is an example of the latter.
- An everyday example of these two forms of communications are telephones (circuit switched or connection-oriented) and postal service (packet switched or connectionless).

2 Internet Protocol (IP) Suite

- Most widely used protocol suite.
- IP= Internet Protocol (network layer)
- TCP=Transmission Control Protocol
- UDP=User Datagram Protocol
- Users usually do not interact with IP layer but TCP or UDP.
- Developed by ARPA in late 60's and 70's for the ARPANET, for research. In 1984, ARPANET splits off MILNET.
- Some standard applications based on this suite:

FTP	file transfer protocol	TCP
TFTP	Trivial file transfer protocol	UDP
Telnet	Remote login service	TCP
SMTP	Simple Mail Transfer Protocol	TCP

- Other protocol suites: SNA (from IBM), XNS (from Xerox), OSI.

3 Reliable Transmission

TCP/IP is said to be **reliable** but UDP/IP is not. Reliability usually means that the protocol takes care of three properties:

1. Error Control
2. Flow Control
3. Sequence Control

Error Control: There are two kinds of transmission errors:

- corrupted data

- lost data

These can be addressed by two corresponding methods:

Acknowledgement: To address corrupted data, the receiver can send back acknowledgement (whether the data was good or corrupted).

Timeout: To address lost data, the sender can retransmit data after a timeout on not getting acknowledgement.

Now, there are several new kinds of errors:

- Message OK, Acknowledgement OK.
- Message OK, Acknowledgement fail.
- Message fail, Acknowledgement fail.

This forces the receiver to perform the following checks:

- Check if received data is OK.
- Check if data is duplicated.

Flow Control: The receiver may easily be swamped with messages for a variety of reasons. If the sender ensures that this does not happen ("pacing"), then we have flow control.

Sequence Control: Received order may differ from sending order. **Fragmentation** can happen at each layer. In the above example, if N+4+8 bytes is larger than the MTU of the IP layer, it may have to break it up into smaller packets, and these are reassembled in the receiver IP layer. This clearly require some sequence control.

In visualization applications, it is clear that sequence control is important.

4 Realtime Issues

TCP processes data as a **byte stream** while UDP views them as **records**.

In byte streams, there is **buffering** of data, both in transmission and receiving.

We can **flush** (or, **push** in TCP terminology) the buffer. There is a tradeoff between throughput and realtime responses in using flush mechanism: this is because every message has a fixed overhead.

There is also the notion of **out-of-band data** (or, **urgent data** in TCP terminology). For example, when a user types control-C or DELETE in an interactive application.

In TCP, there is a special out-of-band data buffer, both in the sending and receiving protocols. This buffer takes precedence over the usual data.

We need some way of informing the process that some out-of-band data arrives – this is done using SIGNALs in UNIX.

NOTE: there is no bound on the size of out-of-band buffers, but some implementation do not guarantee more than 1 byte of such data!

UDP does not provide buffers or out-of-band communication.

5 Addresses

Internet addresses has three levels:

- Network Address (Net ID)
- Host on Network Address (Host ID)
- Process in Host Address (port or process)

The Net ID and Host ID is combined into a 32-bit value. The port or process is a 16-bit integer.

Well-known ports. In TCP/IP, the port numbers 1-255 is reserved for "well-known ports" (corresponding to standard services).

SERVICE	TCP PORT	UDP PORT
echo	7	7
daytime	13	13
netstat	15	
ftp-data	20	
ftp	21	
telnet	23	
smtp	25	
time	37	37
domain	53	53
finger	79	
http	80	
pop-2	109	
pop	101	
sunrpc	111	111
uucp-path	117	
nntp	119	
talk		517

In some systems, ports 1-1023 are reserved.

6 Connections

A connection is an instance of communication link between two processes. Each connection is specified by a unique 5-tuple, called a **association**:

1. Protocol (e.g., tcp, udp)

2. Local Address (Net ID + Host ID)
3. Local Process (Port Number)
4. Foreign Address (Net ID + Host ID)
5. Foreign Process (Port Number)

For instance,

$$\{ \text{tcp}, 128.10.9.3, 1500, 128.10.0.7, 21 \}$$

One half of an association is called a **socket**. An association is thus a 3-tuple (the protocol is duplicated in a socket pair).

7 Client-Server Model

The server basically sleeps, waiting to be contacted by clients. There are 2 kinds of servers:

Iterative: for short jobs. E.g., serve time of day.

Concurrent: for long services. It typically uses the UNIX **fork** and perhaps **exec** system calls.

SERVER PROGRAM:

1. Open communication channel and inform local host that it is waiting at some (well-known) port.
2. Waits for clients.
3. Serves clients (perhaps after forking)
4. Goes back to waiting.

CLIENT PROGRAM:

1. Open communication channel and connects to a server port.
2. Sends request to server.
3. Receives response.
4. Close channel.

8 Berkeley Sockets

REFERENCE: [Stevens, Chapter 6].

This is a user API from BSD Unix (ca. 1982). An alternative is the **Transport Layer Interface** (TLI) from System V (ca. 1986), and modeled after the OSI model [Stevens, Chapter 7]. To access these libraries, you compile your program as follows:

Here is how the server-client would be implemented using Berkeley sockets. There are two versions, depending on whether the communication is connection-oriented or not:

CONNECTION-BASED SERVER:

A. socket()

B. bind()

C. listen() (cf. step b in client)

D. accept() (cf. step b in client)

E. read() (cf. step c in client)

F. write() (cf. step d in client)

CONNECTION-BASED CLIENT:

a. socket()

b. connect() (cf. steps C and D in server)

c. write() (cf. step E in server)

d. read() (cf. step F in server)

CONNECTIONLESS SERVER:

A. socket()

B. bind()

C. recvfrom() (cf. step c in client)

D. sendto() (cf. step d in client)

CONNECTIONLESS CLIENT:

a. socket()

b. bind()

c. sendto() (cf. step C in server)

d. recvfrom() (cf. step D in server)

Note that only the connection-based client skip the bind() step. Below we try to explain this.

9 Berkeley Socket System Calls

SOCKET This is the first action for all processes.
–It returns a **socket ID** (a small integer).
–It specifies the protocol (first item in an association).
–A socket ID is treated like a file descriptor in UNIX.

BIND(sock-ID,...) This needs the socket ID from the SOCKET call.
–One argument to BIND is the local socket (=address and port) information for an association. The user can specify this local socket or specify a "NULL socket". In the latter case, the system will automatically assign a local socket and fill in this information.
–All Servers use this to register their well-known address.
–Connectionless client use this to get an address for receiving data from server.
–Connection-oriented client do not need to bind, but call connect instead.

CONNECT(sock-ID,...) Using the above socket ID, a connection-oriented client tries to contact server.
–This operation returns with a connection (if successful).
–The caller specifies the foreight address and port.
–The local address and port are automatically assigned by the transport layer.
–Connectionless clients can also use this call,

LISTEN(sock-ID,...) A connection-oriented server waits for clients.
–This specifies how many connection requests can be queued (while waiting for accept).
–Then server waits to accept.

ACCEPT(sock-ID,...) This is paired up with the LISTEN call.
–It blocks until there is a connection request.
–It takes the first connection request from the queue.
–Then it creates a new socket ID with same property as the just one used up, and fill up the 5-tuple of the association. This new socket ID is returned.
–In the concurrent server, the server process would then fork, and the child process would use the new socket ID to communicate with client.

SEND, SENDTO, RECV, RECVFROM These are similar to read/write system calls, except we read from or write into sockets.

CLOSE This closes a socket.

Here is a code fragment for a concurrent connection-based server:

```
\tt
  for (;;) {
```

```
newsockID = accept(sockID,...);
if (newsockID<0)
err_sys("accept error");
if (fork() == 0) {  // child process
close(sockID);
serve(newsockID); // serve client
exit(0);
}
// else parent process continues
close(newsockfd);
  } // loop again
```

> Why did the connection-oriented client skip the bind() step? Well, it does not have to skip the bind step, but then it still has to call connect(). That is, it could do socket() bind() connect() However, since the client has no particular interest in any local port, it is easier to skip the bind() step, and let the protocol assign the local port. This way, you avoid one system call. But there is a deeper reason in the case of a concurrent server: such a server forks and let the child server process serve the client (see above). The child client will use a new socket with a different port number – and this need to be communicated to the client in order for the association to be complete. However, the client does not know this BEFORE it connects.

On UNIX systems, you can obtain further information about these commands using the man command (in section 3N of the manual pages). For example:

10 A Simple Client-Server Example and Exercises

We illustrate the TCP/IP protocol under Berkeley sockets. The following C programs (under unix) includes a simple server that waits at port 1234. The server is iterative. The client reads a number N (unsigned integer or float) from the user, and sends it to the server. The server sends back 2N which the client prints. This is repeated until the user types "quit".

The client program is in the Appendix A.

The server program is in the Appendix B.

Thanks to Ting-jen Yen for providing this example.

Chen Li has provided a Java-based version of this client-server program at http://cs.nyu.edu/~chenli/doc/ta/client_server_Java/

EXERCISE: Construct a similar client-server program using UDP/IP.

EXERCISE: Perform measurements to determine the packet loss rate in your UDP/IP protocol.

EXERCISE: (a) Modify the Doubling client-server into one that can send and receive packages of data, and for each packet attach a time sent and received.
(b) Suppose we want to send 100 MB of data to some remote host. Determine the optimal packet size S so as to maximize the throughput T. The experiment must be repeated over a variety of network conditions (time of day). Is S a function of the network conditions (a parameter you may measure). (We will help arrange the remote setup – alternatively access NYU remotely from your modem.)
(c) In visualization, we also want to ensure the most predictable flow of data. What the the optimal S and T, subject making this transmission delay as consistent as possible?

EXERCISE: Modify Homework 1 (rotating color cubes) into a client-server framework. That is, we want a client that displays the rotating cube, but the rendering is performed on the server side. The server sends the rendered buffer to the client who displays it. Upon receipt of an image, the client makes another request. The client requests to the server are one of the following:
(1) Rotate Outer Cube (x,y,x',y') (see HW1 for these parameters).
(2) Rotate Inner Cube (x,y,x',y').
(3) Update Image. This is in the absence of other user commands.
(4) Radio Button Command (spin, shading).
(5) Quit.

EXERCISE: Modify the previous program as follows: we now insert a wavelet transform step into the server side. If I_0 is the image, let I_i be the i-th level resolution of I_0. The client specifies a level k and this is the initial image that he receives. If the image I_0 is not updated between requests, then subsequent requests only has to send the difference to improve the image from I_k to I_{k-1}, I_{k-2}, etc. Of course, if I_0 is modified, the next image is goes back to level k. So, for spinning cubes, we only get low level images. But for stationary cubes, the images gets progressively better.

EXERCISE: Extend the previous exercise to allow foveation.

APPENDIX A: A Simple Client

```
/*

  Author : Ting-jen Yen (yentj@cs.nyu.edu)
  file name : client.c
  related files: server.c

  Description : This is the client side program
                of a simple TCP client-server test program.
                It simply reads a number (unsigned, integer
or float) from the user input, sends it to the
```

```
server (see server.c file), and then
                display everything returned from the server.
                This is repeated until the user types "quit",
at which point both sides will close the connection.

  Usage: Simply type "client" to access the server at
jinai.cs.nyu.edu, at default port 1234.
The general format is
% client [<server_addr>] [-p <port number>]

  How to compile :

  On Solaris 2 :

% gcc -o client client.c -lsocket -lnsl

  On SGI machines or Linux machines:

% gcc -o client client.c


*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>

int             con_sock;
int             srv_port = 1234;
char host[128];

/*
  Received a signal. Close the connection and quit.
*/
void signal_handler()
{
```

```
close(con_sock);

exit(0);
}

/*
  establish_connection() will do what its name
  suggests: makes a connection to the server.
  It will use the global variables host and srv_port
  as input.  Then it will update the value of con_sock,
  which is the file descriptor of the socket
  used for communicating with the server.
*/
establish_connection()
{
struct sockaddr_in name;
struct hostent *hptr;
int flag;

/*  create a socket  */
con_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

if (con_sock <= 0) {
perror("Open socket failed");
exit(1);
}

name.sin_family = AF_INET;
name.sin_port = htons(srv_port);

/* getting the host IP information */
if ((hptr = gethostbyname(host)) == NULL) {
  perror("gethostbyname");
  exit(1);
}

bcopy(hptr->h_addr, &name.sin_addr.s_addr, hptr->h_length);

printf("Trying to contact %s...\n", host);

/* making connection */
flag = connect(con_sock, (struct sockaddr *) &name, sizeof(name));

if (flag < 0)
{
  perror("No connection");
```

```
  close(con_sock);
  exit(1);
}
printf("Connected\n");
}

extern int errno;

/*
  make_trans() will get user input, put them to the
server, and then display the return message from
the server.
*/
make_trans()
{
  char buff[1024], out[1024];
  int lg, flag;

  flag = 1;

  do {
/* getting one line from standard input */
    gets(buff);

    sprintf(out, "%s\r\n", buff);
/* write everything to the network. */
    lg = write(con_sock, out, strlen(out));

/* If having any trouble with the socket, it means that
   the connection is closed somehow, quit the program. */
    if (lg == -1 && errno == EBADF)
      goto error;

/* If the user input is "quit", then end the loop. */
    if (strncmp(buff, "quit", 4) == 0)
    {
      flag = 0;
      break;
    }
    do {
/* get response from the server. If therer is error
   and it is EBADF, it means the connection might
   be closed, otherwise, try to read it again until
   we read it successfully.   */
    lg = read(con_sock, buff, 1024);
    if (lg == -1 && errno == EBADF)
```

```
      goto error;
    } while (lg <= 0);
    if (buff[lg - 1] != '\n' || buff[lg - 2] != '\r') {
       fprintf(stderr, "Error: Wrong format\r\n");
    }

/* Display the response to standard output */
    buff[lg - 1] = '\0';
    buff[lg - 2] = '\n';
    printf("%s", buff);
  } while (flag);

/* close the connection if */
  error:
  close(con_sock);
}


usage()
{
fprintf(stderr, "usage: client [-p port]\n");
exit(1);
}

main(argc, argv)
int argc;
char **argv;
{
int             lg;


/* shift the parameters */
argc--;
argv++;

/* set initial value of the server. */
strcpy(host, "jinai.cs.nyu.edu");
while (argc > 0) {
if (argv[0][0] != '-')
{
/* setting new server name */
  strcpy(host, argv[0]);
  argv ++;
  argc --;
  continue;
}
```

```
switch (argv[0][1]) {
/* setting port number */
case 'p':
srv_port = atoi(argv[1]);
argv += 2;
argc -= 2;
break;
default:
usage();
}
}

/* Intercept some signals so we can quit the program
   more gracefully when interrupted by such signals. */
signal(SIGTERM, signal_handler);
signal(SIGQUIT, signal_handler);
signal(SIGINT, signal_handler);

establish_connection();
make_trans();
}
```

APPENDIX B: A Simple Server

```
/*

  Author : Ting-jen Yen (yentj@cs.nyu.edu)
  file name : server.c
  associated file: client.c

  Description : This is a simple iterative
server that listens at port 1234.
It will accept a number (unsigned, integer or
float) from the client, and returns twice
the number to the client. You can
can access this server by running the client
program (see file client.c), or just using
"telnet" to connect to the server.

  Usage: Simply type "server" in the host machine to serve
at default port 1234.  Otherwise type
"server -p <port number>".

  How to compile :
```

```
  On Solaris 2 :

% gcc -o server server.c -lsocket

  On SGI machies :

% gcc -o server server.c


*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>


int             server_sock, con_sock;
int             srv_port = 1234;
struct sockaddr_in source;

/* Close all sockets before the forced quit caused by
   signal */
void
signal_handler()
{
close(con_sock);
close(server_sock);

exit(0);
}


/* When the state of a child process is changed, (for example,
   it is terminated, or ended by call exit(),) the server
   should call wait() to get the state. This will prevent
   the child process from becoming a zombie process when it has
```

```
   finished. */
void child_end()
{
  wait(NULL);
}

void
init_servconnection()
{
struct sockaddr_in server;

/* Creating socket */
server_sock = socket(AF_INET, SOCK_STREAM, 0);
if (server_sock < 0) {
perror("socket");
exit(1);
}
server.sin_family = AF_INET;
server.sin_port = htons(srv_port);
server.sin_addr.s_addr = INADDR_ANY;

/* bind the specified port to this socket */
if (bind(server_sock, (struct sockaddr *) & server, sizeof(server))
 < 0)
{
perror("bind socket");
exit(1);
}
/* listen for connection */
if (listen(server_sock, 5) < 0) {
perror("listen");
exit(1);
}
}

attenteconnection()
{
int             lg, pid;

lg = sizeof(struct sockaddr_in);

/* repeat until accept a successful connect */
do
  con_sock = accept(server_sock, (struct sockaddr *) & source, &lg);
while (con_sock <= 0);
```

```
/* Fork a child process to serve the client */
if ((pid = fork()) != 0)
{
/* parent process can close the connect and quit */
  close(con_sock);
}
else
{
/* The child process will handle the client, and the quit when
   finished. */
handle_req();
exit(1);
}
}

usage()
{
fprintf(stderr, "usage homesrvd [-p port]\n");
exit(1);
}

extern int errno;

/* handle_req() will get input from the client, parse
   it and write the result back to the client. In this
   case, it will accept a number, either integer or
   float-point value, and return twice the number.
   It will repeat until the input is string "quit".
*/
handle_req()
{
char            buff[8192];
int             lg, flag, srv1, is_float, result_i;
char           *filename, *c, *p;
struct stat     statres;
char            req[1024];
double result_f;

        flag = 1;
while (flag) {

/* Try to repeat until read something from the client,
   unless the connection is broken. */
  do {
  lg = read(con_sock, req, 1024);
  if (lg == -1 && errno == EBADF)
```

```
    goto error;
  } while (lg <= 0);

/* Assume the input will be integer first. */
  is_float = 0;

  if (req[lg - 1] != '\n' || req[lg - 2] != '\r') {
sprintf(req, "Error: Unknown input\r\n");
write(con_sock, req, strlen(req));
continue;
  }

/* Get rid of the trailing new line character and
   carriage return character. */
  req[lg - 2] = '\0';
/* Get the first part of the line only. (Separated by
   space.) */
  c = strtok(req, " ");
/* Empty line */
  if (c == NULL) {
sprintf(req, "Error: Unknown input\r\n");
write(con_sock, req, strlen(req));
continue;
  }
/* quit */
  if (strncmp(c, "quit", 4) == 0)
  {
flag = 0;
break;
  }
/* Check every character to make sure the input is a number,
   determining it is an integer or not on the way. */
  for (p = c; *p /* != NULL */ ; p ++)
  {
    if (!isdigit(*p))
    {
      if (*p != '.' || is_float)
      {
/* If the character is not a digit, and not a decimal point,
   then it is not a number. If it is a decimal point, but
   there is already another one, then it is not a legal number.
*/
        sprintf(req, "Error: Unknown input\r\n");
        write(con_sock, req, strlen(req));
is_float = -1;
        break;
```

```
      }
      else
/* If we have one decimal point, then the number will not
   be an integer. */
        is_float = 1;
    }
  }
  if (is_float == 1)
  {
/* A float-point number */
    result_f = atof(c) * 2;
    sprintf(req, "%f\r\n", result_f);
    write(con_sock, req, strlen(req));
  }
  else if (is_float != -1)
  {
/* An integer */
    result_i = atoi(c) * 2;
    sprintf(req, "%d\r\n", result_i);
    write(con_sock, req, strlen(req));
  }
}

/* Some connection error happened, or quit the loop
   because receiving "quit"  from the client. Will
   close connection and exit this function. */
error:
close(con_sock);
}

main(int argc,char **argv)
{
int             lg;

/* Shift the parameter */
argc--;
argv++;
while (argc > 0) {
if (argv[0][0] != '-')
usage();
switch (argv[0][1]) {
case 'p':
/* Set port number of the server */
srv_port = atoi(argv[1]);
argv += 2;
argc -= 2;
```

```
break;
default:
usage();
}
}

/* Intercept CHILD signal. This signal occurs when a child process
   changes its status. In this program, it usually means when
   a child process calls exit() to end itself. */
sigset(SIGCHLD, child_end);

/* Intercept some signals which will end this process so
   we can close connection before quit it. */
signal(SIGTERM, signal_handler);
signal(SIGQUIT, signal_handler);
signal(SIGINT, signal_handler);

/* Setup the socket and the port */
init_servconnection();

/* Wait for connection */
do {
attenteconnection();
} while (1);

}
```

References

[1] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.