The Application Layer

Michael Brockway

February 25, 2018

Introduction

This lecture concerns the services the *transport* layer (3) of the TCP/IP protocol stack provides for networking *applications* in layer 4.

We shall be concentrating on the *transport control protocol (TCP)* and the *user datagram protocol (UDP)*.

Points to remember from the last lecture that

- *TCP* is a connection-oriented service which sets up a *virtual circuit* between end-points and provides *reliable* data transport between the end-points;
- *UDP* is *connectionless*, light-weight, and there-by, comparatively high-performance; and data transport between end-points is in the form of *datagrams*, independent and individually addressed packets of data.
- The end-points are represented in software by constructs called *sockets*.

Programming TCP connections

```
#include <sys/socket.h>
```

- `int socket(...)`; creates a new socket
- `int gethostname(char *name, int namelen)`; translates a host name to an ip address
- `int bind(int s, struct sockaddr *name, int namelen)`; binds a socket to a specific address (and port)
- `int connect(int s, struct sockaddr *addr, int *addrlen)`; used by client to request a socket connection to a remote address (and port)
- `void listen (socket_id s, int backlog )`; causes server to start listening for requests for a connection
- `int accept(int s, struct sockaddr *addr, int *addrlen)`; used by server to accept a connection request from a client
- `int read(int d, char *buf, int nbytes)`;
- `int write(int d, char *buf, int nb)`; d = socket id; reads from/writes to a socket
- `int close(int d)`;

TCP server logic

```
Fix the port number;
Create a socket for the server;
Start listening on the socket for requests to connect;
Repeat
  Wait for a request for a connection.
  Accept function returns id of a new socket which will manage t
    connection; also gets the name of the client host.
  Spin off a new thread to serve the client.
```

Serving the client

- A subroutine running in a new thread
- Uses socket returned by Accept function
- Uses socket I/O functions to send data to / receive data from client according to protocol
- When finished, closes the socket.

TCP client logic

```
Specify server name/addr, port num to which we wish to connect;
Create a socket;
Bind this socket to host name and port number;
Request a connection to the server host / port;
if (return value indicates connection successful)
 use socket I/O functions to send data to/receive data;
Close the socket when finished.
```

- The port number is same at both ends: identifies the virtual circuit between the client and server.

TCP server and client examples

Download http://computing.northumbria.ac.uk/staff/cgmb3/teaching/sockets/4JavaCExs/3_CltSvrDataApp.zip

Unzip it in two hosts connected by a network.

- The server has a list of registered users with passwords, and a scoreboard: a list of player names and their scores.
- Any client may ask for and receive a complete scoreboard listing, or lines matching a submitted name.
- A registered user may log in using their password and may then edit the scoreboard: update, insert, delete lines.

TCP server and client examples

The protocol

- `All>` - ask the server for a complete listing
- `Rpt#Name>` - ask the server for score(s) matching Name
- `Login#User#Pswd>` - log in a user
- `Term>` - terminate current session (server still running)

Once logged in:

- `Upd#Name#newScore>` - update score matching Name to newScore (a numeric string)
- `Ins#Name#newScore>` - insert score (Name must not match an existing name)
- `Del#Name>` - delete score matching Name
- `Down>` - down the server

TCP server and client examples

- There are two clients: one with a GUI, written in Java, and one written in C, with a command-line user interface. The C client uses the 'simplified interface' tcpFunctions.c,h.
- The java server is multithreaded - each client accepted is served by a new thread which runs until the client terminates.
- The server goes on accepting new clients until a client sends a 'down' command. After this one more connection is accepted but the server goes to EOJ after that. Thus, ...
- Shutdown procedure: A logged-in client should send a "down" command, then terminate; another client will then be able to connect and terminate; the server will then finish.

TCP server and client examples

Building the example: see make file - Java builds use javac, C builds use gcc.

- `$ make JServer`
- `$ make JClient`
- `$ make CServer`
- `$ make CClient`
- `$ make clean` (delete binary and intermediate files)

Run either server on a host on the same network as the client host. Quote a *port number* on the command-line. `passwords.txt` and `scoreboard.txt` need to be in the same directory as the server executible.

Run either client on a host on the same network as the server host. Quote the IP address of the server, and the port number, on the command line. The IP address should be quotated.

Programming UDP

Again use a *socket* but no connection is established. Instead, a version of the send function is used which incorporates a destination address parameter:

```
int sendto(
  int sockID,           //socket ID
  char * msg,           //data payload (array of bytes)
  unsigned int msgLen, //length of array
  int flags,            //options; eg, non-blocking, timeout
  struct sockaddr * destAddress,
  unsigned int addressLen)
```

`destAddress` points to a record containing destination address information, including the IP address and port number.

The return value is a result code: the number of bytes actually sent or (if negative) an error code.

Programming UDP

The corresponding receive function is incorporates a source address parameter:

```
int recvfrom(
  int sockID,           //socket ID
  char * msg,           //array for data to go into
  unsigned int msgLen, //length of array
  int flags,            //options; eg, non-blocking, timeout
  struct sockaddr * sourceAddress,
  unsigned int * addressLen)
```

The return value in each case is the number of bytes received.

Note is the use of pointers to provide *in/out* parameters: the memory pointed at by (non-null) `sourceAddress` and `addressLen` are filled in by the inpcoming message with source address data and its length.

Programming UDP

The sourceAddress and destAddress are 'struct' records incorporating addressing information, including the IP address and the port number.

- The IP address directs a datagram to the correct host computer and
- the port number directs it to the correct process running on the host.

UDP examples

Download http://computing.northumbria.ac.uk/staff/cgmb3/teaching/sockets/3JavaExamples/UDP_Java_Echo.zip

Unzip it in two hosts connected by a network.

This is a java implementation of a simple echo server and a client.

- Both have a simple grapical user interface.
- The client accepts a short message into a text box; ENTER dispatches it to the server.
- The Server unpacks it from the UDP datagram, displays information about the received packet in a scrolling JTextArea and simply sends a replay to the client coontaining a copy of the original message.
- The client displays information about the packet received in reply in a scrolling JTextArea.

UDP examples

This example illustrate the rather neat Java encapsulation of UDP using classes

- java.net.DatagramSocket
- java.net.DatagramPacket

Simply run them in two networked hosts. You need to quote the IP address of the server on the command-line you use to start the client. The server knows the IP address of its client because it arrives in the packet.

Try running a server with multiple clients.

UDP examples

Download http://computing.northumbria.ac.uk/staff/cgmb3/teaching/sockets/4JavaCExs/5_DisplayServer.zip

Unzip it in two hosts connected by a network.

- `DisplayServerUDP.java` is a server that uses connectionless UDP to display a graph of a time series of numbers that it receives as a series of UDP datagrams.
- `DisplayClientUDP.java` and `DisplayClientUDP.c` are example clients. They send a burst of 200 random-walking numbers. Hit ENTER for another burst, x to finish.

This zip also contains a TCP client and server fr the sake of comparison.

- `DisplayServer.java` is a server that provides the same service: displays a graph of a time series of numbers that it receives over a TCP connection.
- `DisplayClient.java` sends a burst of 200 random-walking numbers. Hit ENTER for another burst, x or X to finish.

UDP (and TCP) examples

The TCP server and client are built with javac:

```
$ javac DisplayServer.java
$ javac DisplayClient.java
```

The UDP C client uses the 'simplified interface' tcpFunctions.c,h which have to be included in the build. To build the UDP examples use make (see make file - Java builds use javac, C builds use gcc):

```
$ make Server
$ make JClient
$ make CClient
```

$ `make clean` (delete binary and intermediate files)

An EMBED client

Download

http://hesabu.net/kf5011/lectures/kf5011-L07-1-DisplayClient.zip

```
#include <mbed.h>
#include <EthernetInterface.h>
#include <USBSerial.h>
#include <rtos.h>
#include <mbed_events.h>
#include <C12832.h>

EthernetInterface eth;
UDPSocket udp;
unsigned char buffer[8];

char servAddr[] = "192.168.1.253";
SocketAddress server(servAddr, 5000);

union {
  double xx;
  unsigned char bts[8];
} u;
```

An EMBED client

```
/*** Switch to big-endian! Reverse byte order in byte array
  * in: b; out: r
  **/
void revBytes(unsigned char *b, unsigned char *r, int lh) {
  int i=0;
  for (i = 0; i < lh; i++)
    r[lh-1-i] = b[i];
}

AnalogIn left(A0), right(A1);
C12832 lcd(D11, D13, D12, D7, D10); // Using Arduino pin notation

DigitalOut red(LED_RED,1); /* initial state 1 led is off */
DigitalOut green(LED_GREEN,1);
DigitalOut blue(LED_BLUE,1);
bool flashing = true;
```

An EMBED client

```
/** update RGB LED: RGB = bottom 3 bits of s */
void flash(void) {
  static unsigned char s = 0; //initilised on first call only
  if (flashing) {
    s++;
  } else {
    s = 0;
  }
  red.write(1 - (s&1));
  green.write(1 - ((s>>1)&1));
  blue.write(1 - ((s>>2)&1));
  wait(0.5);
}

void toggleFlash(void) {
  flashing = !flashing;
}
```

An EMBED client

```
int main() {
  printf("connecting \n");
  eth.connect();
  const char *ip = eth.get_ip_address();
  printf("IP address is: %s\n", ip ? ip : "No IP");

  udp.open(&eth);
  printf("sending to %s\n", servAddr);

  while (true) {
    u.xx = ((left.read() + right.read()*5)-3)*100;
    revBytes(u.bts, buffer, 8);
    nsapi_size_or_error_t r = udp.sendto(server, buffer, sizeof(buffer));
    if (r < 0)
      printf("sendto returned code %d\n",r);
    //  printf("sent %d: %f\n", r, u.xx);
    lcd.locate(0,10);
    lcd.printf("data = %f", u.xx);
    flash();
    wait(0.5);
  }
}
```

Further reading

Kenneth L. Calvert and Michael J. Donahoo (The Morgan Kaufmann Practical Guides Series)

- TCP/IP Sockets in C, Second Edition: Practical Guide for Programmers (2009)
- TCP/IP Sockets in Java, Second Edition: Practical Guide for Programmers (2008)
- Pocket Guide to TCP/IP Socket Programming in C (2000)