Chapter 7
Client Software
(code details)

Principle: Design small modules, then use them.

Reason: You only have to figure it out once.

Principle: Build modules with parameters.

Reason: You can build fewer of them.

Flexible Connection Maker
(Detailed Comments)

See Sections 7.5-7.7

Include file overview (p. 85):

Include files are found in `/usr/include`

`types.h`: Short hand type names such as `u_long` for `unsigned long`

`socket.h`: Types and calls for sockets

Identifier/number mappings for
socket types (`SOCK_STREAM`)
socket options (`SO_BROADCAST`)
address families (`AF_INET`)
protocol families (`PF_INET`)

Structures for: protocols (`sockproto`) , addresses (`sockaddr`), messages (`msghdr`)

`in.h`: Constants and structures for IP

Identifier/number mappings for
protocol (`IPPROTO_IP`)
standard ports/sockets (`IPPORT_ECHO`, `IPPORT_LOGINSERVER`)

Internet version of structures (`sockaddr_in`, `in_addr`)

Conversion functions (`ntohs`, `htons`, `ntohl`, `htonl`)

`inet.h`: inet-to-ascii function `inet_ntoa`, ascii-to-inet function `inet_addr`

`netdb.h`: Structures and function definitions for accessing the network databases.

Structures:
`hostent`, `servent`, `protoent`

Functions:
`gethostbyname`, `gethostbyaddr`, `get net/serv/proto/rpc by name/addr`

Error message meanings:
`HOST_NOT_FOUND`

Making a Connection

`connectTCP` calls `connectsock` with a protocol of `"tcp"`.

`connectUDP` calls `connectsock` with a protocol of `"udp"`.

```
connectsock(host,service,protocol)
```

Parameters are strings.
`host`: the name of the server machine
`service`: the name of the service
`protocol`: `"udp"` or `"tcp"`

`memset(&sin,0,sizeof(sin));`:
The address structure (`sin`) will be used to make the connection. Initialize it by clearing all bits.

`sin.sin_family=AF_INET;`: address is for an IP connection (not Decnet, Novell ...)

Getting a Port Number

First: guess it was the name of a service (e.g., `"time"`)
Second: guess it was a port number (e.g., `"7154"`)
Third: abort the program

`if (pse=getservbyname(service))`: Try looking up the service in the `/etc/services` file.
if it finds a service it creates a `servent` structure and returns a pointer to that structure. If an entry is not found it returns `NULL`.
C trick: not found = `NULL` = 0 = false.
If a `NULL` is returned we will go to the `else`.

`if` part (valid pointer returned)
`sin.sin_port=pse->s_port`.
Fill in the port number from the structure. It's already in network order.

`else` part (`NULL` returned)
`if((sin.sin_port=htons((ushort)atoi(service)))==0)`

`atoi`: convert a string to a number.
The number is converted to network order and stored in `sin_port`
Trick: if the string wasn't a number `atoi` returns a zero
A zero will cause `errexit` to be called, which will print a message and abort the program.
`errexit("can't get...");`

Getting an Internet Number

First: guess that it was a name like `cheetah.cecs.csulb.edu`.
Second: guess that it was a number like `134.139.248.17`
Third: abort the program

`if (phe=gethostbyname(host))` Try looking up a host name. If it was not a legal a `NULL` will be returned and we will end up at in the `else`

`memcpy(&sin.sin_addr,phe->haddr,phe->h_length);` The name look up worked, copy the address from the `hostent` structure to the `sockaddr_in` structure. Note: the `sockaddr_in` assumes an internet address (4 bytes), but the `hostent` is a `*char`, it is using a "string" to store an arbitrary number of bytes (so it works for non-internet addresses).

Not a hostname (the `else`)

```
if
((sin.sin_addr.s_addr=inet_addr(host))==INADDR_NONE)
```

Assume it's a dotted notation internet address
convert it to an address and store it in the `sockaddr_in` structure.
Trick: if it isn't a dotted notation, `inet_addr` returns `INADDR_NONE` and we do `errexit("can't get...")`

Setting the Protocol Field

`if ((ppe=getprotobyname(transport))==0)` If the protocol doesn't exist if returns `NULL` otherwise it returns a pointer to a protocol.
Since we require tcp or udp this is really a spelling check.

```
if (strcmp(transport,"udp"))
  type = SOCK_DGRAM;
else
  type = SOCK_STREAM;
```

if the protocol string was `"udp"` we are connectionless (`SOCK_DGRAM`), otherwise we are connection-oriented (`SOCK_STREAM`).

At this point the address structure has been correctly set up.

Inside `connectsock`, `transport` is the name of the third parameter (the protocol).

```
s=socket(PF_INET, type, ppe->proto);
```
get an internet socket of the appropriate type and protocol.

```
if (connect(s,(struct sockaddr *)&sin,sizeof(sin)<0)
  errexit(..., strerr(errno));
return s;
```

Try to connect to the correct server by using the address structure we have just filled in.
Note: type conversion `(struct sockaddr *)`
connect is multiprotocol, it looks at the first byte (`AF_INET` for our connect) to know if it is Internet, Decnet, ...

Connect fills in the socket descriptor (`s`) with information about the connection, including the server you are connected to and which port (protocol) on that server.
or connect returns a negative number (error).
On error: connect fills in the global variable `errno`, `errno` is used to look up a message (`strerror`)

UDP note: connect sets up a default address (not a connection)
TCP note: connect sets up a connection

Summary: `connectsock` returns a socket connected to the right service and server.

The Switch

(Section 7.10, page 89)

All clients use this `switch`.
Goal 1: allow the user to specify the name or number of the server they want to connect to on the command line. If they don't specify a server, assume they want to use the local host.

Goal 2: allow the user to specify the service on the server or the port number on the server they is to be used. if they don't specify a service or port number, assume they want the `daytime` service.

The allowed formats for the command are:

```
a.out
a.out cheetah.cecs.csulb.edu
a.out 134.139.248.18
a.out cheetah.cecs.csulb.edu ftp
```

The last form uses the time client to connect to the ftp service. (Dumb idea if you are not an ftp client!)

Notice: there are one, two are three arguments (counting the command)

How many commandline arguments are there?
1: use the default server and service
2: user specified a server, use that string
3: user specified a server and service, use both strings

```
int main(int argc, char* argv[]) {
  char *host = "localhost";
  char *service = "time";
  switch (argc) {
    case 1:
      break;
    case 3:
      service = argv[2];
    case 2:
      host = argv[1];
      break;
    default: fprintf(stderr,"goofed\n");
  }
  ServiceProcedure(host,service);
  exit(0);
};
```

First, setup the defaults; then the switch overrides them.
Note: Case 3 falls through to case 2.
Note: the error is printed to standard error.
(This method is flexible, i.e., good software engineering.)

UDP

Each write is atomic, a single message.

If it goes in one write,
it will be received in one read.

If the read buffer isn't big enough, any excess is discarded (lost).

Delivery is not guaranteed.
If delivery fails, no one is notified,
i.e., the data is lost.

A UDP connect, sets up a default address for `write` and `send`.

A UDP connect will accept replies only from the default address.

UDP Time

(section 7.15) Time service returns an unsigned integer representing the time.

`s=connectUDP(host,service);` Not a connect. Associates the socket with a default address.
The socket can be used to send messages to any address
write sends a message to the default address
read will only read replies from the default address.

`write(s,MSG,strlen(MSG));` send a dummy message indicating we want the time

`read(s,&now,sizeof(now));`
Up do 4 bytes will be read.
You need the destination address (`&now`. Remember, the name of an array is an address.
A long integer comes back.
UDP reads a complete message, i.e., reads one write
If there are more than 4 bytes in the reply, the remainder are discarded.

`now=ntohl(now);` By convention every number sent across the network is sent in network standard order. To do math and print the time, we need host order.
Always use network standard order when sending data.

`now-=UNIXEPOCH;` The time protocol specifies seconds since 1 January 1900, the time routines use seconds since 1 January 1970; To use the time routines on this machine, we convert from the 1900 format to the 1970 format.

`printf("%s",ctime(&now));` The system routine `ctime` converts the time to a string. That string is printed.

Notes:

1) To be consistant and the code should be put into a procedure.

2) write specifies how much to write.
The number of bytes actually written is returned.
If it is not equal to the number of bytes you asked to be written; either a system limit has been exceeded, or the OS has a problem.

3) read specifies the size of the buffer. Up to that much may be read.
The number of bytes actually read is returned.

UDP Echo

writes a line, then reads the line.

`char buf[LINELEN+1];`
You need a real buffer, with real space, a `char*` will not do.

`s=connectUDP(host,service);`
Sets default for write and focus for reply.

`while (fgets(buf,sizeof(buf),stdin))`
What the user types goes into `buf`.
If the user types more than `sizeof(buf)` the first `sizeof(buf)` letters are returned (immediately).
Returns `NULL` on end of file from keyboard. The user gets this by typing control-D. A `NULL` will exit the loop.

`buf[LINELEN]='\0';`
The buffer will have a string terminator unless the `fgets` returned because the user typed more characters than the buffer will hold. In this case a string delimiter (`'\0'`) will be placed in the last position of the buffer, overwritting the last letter in the buffer.

If the string is shorter than the buffer, it will have two delimiters, but only the first delimiter (the one set by `fgets`) matters.

`nchars=strlen(buf);`
We don't need this, we could continue to use the `strlen` call.

`(void) write(s,buf,nchars);` We write the meaningful part of the buffer (and not the whole thing). Notice the `'\0'` didn't get written, it would be better if it did, we'd get a delimiter back in the echo. Notice the cast, bad, should not cast and should check
`if (write(...)<0)`
for an error

```
if (read(s,buf,nchars)<0)
  errrexit("socket read...);
```

Read the echo backinto the buffer. The buffer should be big enough, because it should be the message that was sent from the buffer Good, check for an error here. If we needed to use the characters read, we could:

```
if ((nchars=read(s,buf,nchars))<0)
  errrexit("socket read...);
```

`fputs(buf,stdout);`
This might work, but the delimiter was forgotten.
Could use
`fwrite(buf,1,nchars,stdout);`
or could add Note (again): UDP does not form a connection.

TCP Clients

1) A TCP connection is formed. The server knows there is a client because an accept happens. In UDP the client had to send a message for the server to know there was a client.

2) TCP delivery is reliable. If something goes wrong TCP calls return negative numbers. For example, we will never freeze at the read, read will return -1 if the server isn't there.

3) TCP is a stream. Message boundaries are not guaranteed. Even when the server sends the message in one chunk, it may arrive in several. This is particularly true of larger sends. Even if the server sends the message in two pieces, it may arrive in one. Reading has to be done more carefully than with UDP.

TCPDaytime

`char *service = "daytime";`
This will be the default service. We don't recommend overriding.

`char buf[LINELEN+1];`
The daytime server returns a string. Real storage room to store that string

The main program switch is the same so we concentrate on the `TCPdaytime` function.

`TCPdaytime(const char *host, const char* service)`
This passes in the host and service. The service is expected to be `"daytime"`.

`s=connectTCP(host,service);`
A connection is made. After this call `s` is now a descriptor which can be used to talk to the server. The server knows there is a connection request (accept). Since this is the daytime service, a connection is all that is necessary to cause the server to send the client the daytime.

Important: the answer may arrive in parts.
The parts may not correspond to record, string, or integer boundaries (TCP *does not preserve boundaries*)

Also: type matching is the responsibility of the protocol (daytime) and not the socket/read.

`while ((n=read(s,buf,LINELEN))>0)`
Read from the socket. `n` is number of bytes actually read,

0 is EOF, negative means read error.
`buf`: is where it gets stored.
`LINELEN`: is maximum number of bytes to read.
Notice we go into the loop only if the read worked.
We received a piece of the reply, so we loop until EOF.

`buf[n]='\0';`
As each piece of the message arrives we add a string terminator. That is so the print will work. Remember this could be a piece of the string.

`(void)fputs(buf,stdout);`
print the string to the screen.

Note: The Daytime server sends one string in one write, it may arrive at the client in several reads.

TCP Echo
(Section 7.17)

Although the input stream does not require different handling, an alternate method of handling is used by this example.

In the previous example, each piece that was received was printed when it was received because it didn't matter if a half-line came in. When you receive half an integer, you have to save it until you receive the other half.

Alternate tactic: assemble the entire echo, then print. The number of bytes to be read is equal to the number that were written. (This is an echo.)

Again. the main program is the same except for (1) the default service is `"echo"` and (2) the call is to the `TCPecho` function.

`char buf[LINLEN+1];`
The user's input will go here, then will be written to the echo server. After that, the reply from the echo server will be assembled in this buffer. I recommend different buffers, because it makes it easier to distinguish what the user has input from what has been echo'ed.

`int s, n;`
`s` will be used for the socket, `n` will be used to temporarily store the number of characters returned by the last read.

`int outchars, inchars;`
The number of characters written (which is the number of characters we expect back; and the number of characters read from the echo server. We will have received the entire echo when `inchars` becomes equal to `outchars`.

```
s=connectTCP(host,service);
while (fgets(buf,sizeof(buf),stdin)){
  buf[LINELEN]='\0';
  outchars=strlen(buf);
  (void)write(s,buf,outchars);
```

The same as in the UDP echo except for the variable name. A line is read from the keyboard. A terminator is added that matters only if the line is longer than the buffer. The line is written to the server.

Since we the message may get broken up in transmission it must be carefully reassembled when it is echoed back.

```
for (inchars=0; inchars<outchars; inchars+=n){
  n=read(s,&buf[inchars],outchars-inchars);
  if (n<0) errexit(...);
}
```

`inchars`: keeps track of how many bytes we have read.
`inchars<outchars`: we stop when we have gotten back the same number of bytes that we sent to the server.
`n=`: Save is the number of we got from the read.
`inchars+=n`: increment the total bytes read by the number we got on the last read.
`&buf[inchars]`: is the end of the previous input. It is where the new input should start. Read needs the address of where the input needs to start.
`outchars-inchars`: this is the number of bytes not yet echoed, this is the largest read that could possibly happen.
`if (n<0)`: if the read fails (server died), exit

`fputs(buf,stdout)`: Now the line has been assembled print it. It should be the same as the line that was typed.

Note: `buf[outchars]='\0';` would ensure string termination, especially if we use two buffers.

TCP recap

1) Assemble the entire answer from multiple reads.

2) Process the answer.

Assembling an integer

```
int answer;
for (inchars=0; inchars<4; inchars+=n){
  n=read(s,&buf[inchars],4-inchars);
  if (n<=0) errexit(...);
}
memcpy(&answer, buf, 4);
answer = ntohl(answer);
```

Read the 4 bytes into a buffer (may take several reads).

Copy the 4 bytes from the buffer into an integer.

Convert the integer to host order.

`(n<=0)`, we've got to get 4 bytes, if we get a disconnect or EOF first something is wrong.