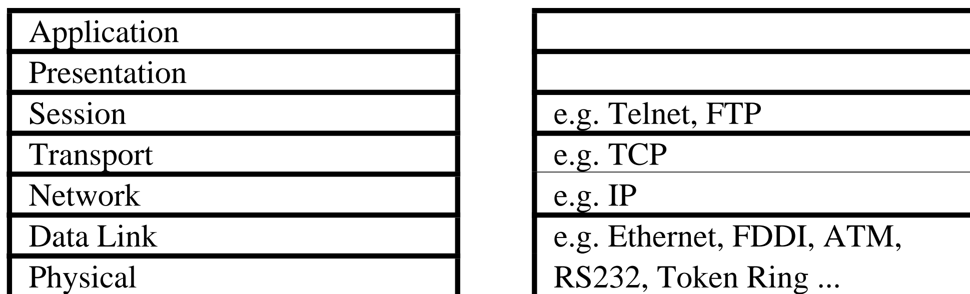
MSc Integrated Electronics
Networks Assignment

Investigation of TCP/IP Sockets and Ports

Gavin Cameron

Introduction

TCP and IP (Transmission Control Protocol / Internet Protocol) are two protocols from the OSI (Open Systems Interconnection) 7 layer model which enable data to be transmitted across a network between applications. The protocols do not concern themselves with how the data is physically transmitted, as that is the job of the two layers below them (see figure 1 for an example of the OSI 7 layer model). Rather, the model has been designed in such a way that the layers are independent of each other. The only bounding factor is the interfaces between layers. Hence, the TCP / IP pair of layers format the data in such a way that they can pass the data onto the Data Link layer and forget about it.



Layer	Example
Application	
Presentation	
Session	e.g. Telnet, FTP
Transport	e.g. TCP
Network	e.g. IP
Data Link	e.g. Ethernet, FDDI, ATM,
Physical	RS232, Token Ring ...

figure 1

The IP layer allows the data packets to be routed through, perhaps, several networks to its destination node with the aid of an IP address. The layer does not know if the destination node actually receives the data, that is part of the job of the TCP layer - to establish and monitor a reliable virtual connection, or stream, between the two nodes.

The TCP layer allows applications on each node to talk transparently to each other with the aid of ports and sockets. A port is analogous to a pigeon hole mail box, where only the mail for any one person is delivered to a hole. With ports, data of a specific type of service is addressed to specific ports, for example Telnet data is directed to port 23, Simple Mail Transfer Protocol (SMTP) is directed to port 25. This feature of TCP can be used as one barrier in a firewall system, where a firewall might allow SMTP into a company, but not allow Telnet in, however, let it out. A socket is the method of providing a unique connection, i.e. there may be several telnet applications running and each one will have it’s own socket number to allow data to get to the correct application.

Programming with TCP / IP

From the programming point of view, it is very simple: the TCP / IP layers take care of themselves, the user just has to communicate with the TCP layer interface. All that is required is an IP address, a port number, the unique socket number (assigned by the Operating System) and the data.

With a TCP / IP connection, there will be a server and a client application. These could be on the same node, or more realistically, on different nodes anywhere in the world. A typical example of this scenario is the World Wide Web (WWW). When "surfing" the internet for hypertext documents, the documents are held on servers all over the globe. These servers will be running some form of http (hyper text transfer protocol) daemon which will be listening for a request on a specific port - usually port 80. When they receive a request for information from a client (often a home PC), they will service the request and transmit the information to the client.

Unix / Linux systems are based upon TCP / IP as their main protocol for communicating with each other. Because of this, the functions for communicating with the TCP interface are all contained within the standard system "C" libraries.

The investigation into TCP / IP ports and sockets will now focus on a scenario based around Unix / Linux. It should be noted that this was chosen because of accessibility, the scenario will work for any platform assuming the code can be written. For example, in a Microsoft Windows environment, the programmer communicates with WINSOCK.DLL. This is a Dynamically Linked Library that "bolts in" to the Windows environment and provides the interface to the TCP layer of the OSI model. This DLL is required as Windows does not natively communicate with TCP / IP.

Scenario

A powerful database (aka a PC running linux) is being used as a service for finding out information on any number of subjects. This will be running a TCP server daemon which will listen on a special port for questions from client PCs. The client will then pass to the server the question. Upon completion of the database search, the server will return the answer to the client.

The two applications will have different structures to them. An outline of the sequence of events for both is shown below:

Server - socket - bind - listen - (accept - (read - write) - close)
Client - socket - connect - (write - read) - close

Server Application

For simplicity of this investigation (this is a protocol investigation, not a software exercise), the application will not be a daemon (an application that runs in the background), but will run in the foreground.

The first TCP related action is to define a socket. This is achieved with the *socket* command:

```
int socket(int domain, int type, int protocol);
```

where:

domain is

AF_UNIX	(UNIX internal protocols)
AF_INET	**(ARPA Internet protocols)**
AF_ISO	(ISO protocols)
AF_NS	(Xerox Network Systems protocols)
AF_IMPLINK	(IMP "host at IMP" link layer)

type is

SOCK_STREAM	**sequenced, reliable, two-way connection**
SOCK_DGRAM	connectionless, unreliable messages
SOCK_RAW	access to internal network protocols and interfaces
SOCK_SEQPACKET	presently implemented only for AF_NS
SOCK_RDM	planned, but not yet implemented

protocol is

IP	internet protocol, pseudo protocol number
ICMP	internet control message protocol
IGMP	internet group multicast protocol
GGP	gateway-gateway protocol
TCP	**transmission control protocol**
PUP	PARC universal packet protocol
UDP	user datagram protocol
IDP	Undocumented?
RAW	RAW IP interface

As can be seen from the available options, this command can be very powerful for communicating with a number of different protocols which use the Internet Protocol.

This command declares a socket, the number of which is returned by the function. In C, this is treated as a file descriptor, as are all Input / Output functions in C. At this point, it cannot do anything as it has not been allocated to a port / IP address. This is achieved with the *bind* command:

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

This assigns the socket file descriptor *sockfd* to the address *my_addr* (which contains IP address and port address) which is of length *addrlen*.

The socket now has to be assigned as a server with the *listen* command:

```
int listen(int s, int backlog);
```

This tells the application to listen to socket *s* for a request, but in case several requests come in before the application has time to service them, have a log of *backlog* entries to service.

After a request is received, the server will acknowledge it with the *accept* command:

```
int accept(int s, struct sockaddr *addr, int *addrlen);
```

Which will accept socket *s* which was a request from node *addr* whos address is *addrlen* long. A stream has now been established between the server and client applications. The function returns a file descriptor of the socket created for the incoming stream - this is the socket that communication between the server and client will be carried out on.

The server will then expect data from the client. It shall do this with the *read* command:

```
ssize_t read(int fd, void *buf, size_t count);
```

This command attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*. The actual number of bytes read is returned.

The server will then perform its database search based on the question from the packet(s) received, and transmit the answer back to the client with the *write* command:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Which writes the buffer *buf* of size *count* to the file descriptor *fd*.

There could be several transactions between the two nodes until finally the server will terminate the stream with a *close* command:

```
int close(int fd);
```

Which closes the file descriptor *fd*. Both file descriptors must be closed before the application exits.

Client Application

The client application follows a very similar structure. Firstly a socket is defined with the *socket* command:

```
int socket(int domain, int type, int protocol);
```

However, the next step in the clients application is to try to establish a connection with the server. It does this with the *connect* command:

```
int connect(int sockfd, struct sockaddr *serv_addr, int  addrlen );
```

This tries to connect the socket with file descriptor *sockfd* to the server *serv_addr* whos address length is *addrlen*. The function returns a zero on a successful connection, but on a failure, returns the reason to the OS, for example, "connection timed out" or "connection refused."

A conversation can then take place between the client and server with the *write* and *read* commands:

```
ssize_t write(int fd, const void *buf, size_t count);

ssize_t read(int fd, void *buf, size_t count);
```

Again, the socket must be closed with the *close* command:

```
int close(int fd);
```

Execution

The server application must be started up first, or when the client tries to connect it shall get a "connection refused" error.

The two application were written and compiled on a linux PC. Below is the text dumps from both applications:

Server

```
Gav’s Networks assignment - server
server: Got a connection from 127.0.0.1
server: requested question: "What is a banana?"
```

Client

```
Gav’s Networks assignment - client
Enter IP address of server: 127.0.0.1
client: answer to the question "What is a banana?" is
 "A banana is a fruit."
```

Note: IP address 127.0.0.1 is a special address on Unix systems. It is the Local Loopback address for communication with itself.

If the server is not running when the client tries to initiate a connection, the following dump will be produced:

```
Gav’s Networks assignment - client
Enter IP address of server: 127.0.0.1
client: connect(): Connection refused
```

As the applications were written in C which is a generic language, the server application was FTP’d over to a Sun Microsystem and re-compiled there. The applications were then re-run (server on a Sun and client on a Linux PC) and dumps are shown below:

Server

```
Gav’s Networks assignment - server
server: Got a connection from 137.221.203.182
server: requested question: "What is a banana?"
```

Client

```
Gav’s Networks assignment - client
Enter IP address of server: 137.221.203.50
client: answer to the question "What is a banana?" is
 "A banana is a fruit."
```

As these applications use some of the unix libraries, they cannot be ported directly over to, for example, a Microsoft Windows environment. However, the commands used are very similar in structure to commands in Visual C++ for communicating with the WINSOCK.DLL.

Conclusion

As was the intended idea with the 7 layer OSI model, communicating with other applications is extremely easy. The programmer just has to interface with the TCP layer, and the rules of the layers take care of the rest of the functions involved. The protocol is independent of the system that it is running on which is why machines of different types can integrate so easily.

Server Application Code

```
/* Gav's "All knowing" server
   This server emulates the scenario of a powerful server with
   huge databases. Any client can ask the server a question and
   will then get the answer sent back in a TCP/IP packet.

   Gavin Cameron, 18th Jan 1999 */

#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/utsname.h>
#include <netinet/in.h>
#include <string.h>

#define PORT   1111                  /* Currently unused port */

main()
{
  int my_file_descriptor,            /* servers TCP file descriptor */
      client_file_descriptor,        /* clients TCP file descriptor */
      client_address_size;           /* size of clients TCP/IP address */
  struct sockaddr_in my_address,     /* servers TCP/IP address */
                     client_address; /* clients TCP/IP address */
  unsigned char *address_holder;     /* clients IP address */
  char buffer[256],                  /* Receive TCP buffer */
       answer[50];                   /* "answer" to the question */

  printf("Gav's Networks assignment - server\n");

  strcpy(answer, "A banana is a fruit.\0");

  /* define a socket and check that it was created ok, if not display OS
     error message on stderr (stdout) */
   my_file_descriptor = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (my_file_descriptor < 0)
  {
    perror("server: socket()");
    return;
  }

  /* Clear servers IP address field */
  memset((void*)&my_address, 0, sizeof(my_address));

  /* Set servers IP address field */
  my_address.sin_family = AF_INET;
  my_address.sin_port = htons(PORT);
  my_address.sin_addr.s_addr = htonl(INADDR_ANY);

  /* "name" the socket created, ie. give it the servers IP address */
  if (bind(my_file_descriptor, (struct sockaddr*)&my_address, sizeof(my_address)) < 0)
  {
    perror("server: bind()");
    return;
  }

  /* Listen for a request on the socket */
  if (listen(my_file_descriptor, 5) < 0)
  {
    perror("server: listen()");
    return;
  }

  /* Clear clients address field */
  memset((void*)&client_address, 0, sizeof(client_address));

  client_address.sin_family = AF_INET;
  client_address_size = sizeof(client_address);

  /* Accept the connection on the socket */
  client_file_descriptor = accept(my_file_descriptor, (struct sockaddr*)&client_address,
&client_address_size);
  if (client_file_descriptor < 0)
  {
    perror("server: accept()");
    return;
  }

  /* Point to client IP address */
  address_holder = (unsigned char*)&client_address.sin_addr.s_addr;

  printf("server: Got a connection from %d.%d.%d.%d\n", address_holder[0], address_holder[1],
address_holder[2], address_holder[3]);

  /* Read the "question" from the client */
  if (read(client_file_descriptor, buffer, 255) < 0)
  {
    perror("server: read()");
```

```
    return;
  }

  printf("server: requested question: \"%s\"\n", buffer);

  /* send the "answer" to the client */
  if (write(client_file_descriptor, answer, sizeof(answer)) < 0)
  {
    perror("server: write()");
    return;
  }

  /* close the socket connections */
  close(client_file_descriptor);
  close(my_file_descriptor);

  return;
}
```

Client Application Code

```
/* Gav's "questioning" client
   This client emulates the scenario of asking a powerful server
   a question by sending a TCP request. The server will answer the
   question.

   Gavin Cameron 18th Jan 1999 */

#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/utsname.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>

#define PORT   1111                   /* Currently unused port */

main()
{
  int my_file_descriptor,             /* clients TCP file descriptor */
      counter = 0,                    /* counter for use in a loop */
      IPp1, IPp2, IPp3, IPp4;         /* 4 parts of an IP address */
  struct sockaddr_in server_address;  /* servers TCP/IP address */
  unsigned char *address_holder;      /* servers IP address */
  char buffer[256],                   /* receive TCP buffer */
       server_IP_address[256],        /* string to hold servers IP address */
       request[50];                   /* "question" to ask the server */

  printf("Gav's Networks assignment - client\n");

  strcpy(request, "What is a banana?\0");

  /* Ask user for severs address */
  printf("Enter IP address of server: ");
  scanf("%s", server_IP_address);

  /* Scan IP address, changing '.'s to ' 's for sscanf'ing */
  while (server_IP_address[counter] != 0)
  {
    if (server_IP_address[counter] == '.')
      server_IP_address[counter] = ' ';
    counter ++;
  }

  /* Read 4 IP parts into variables */
  sscanf(server_IP_address, "%d %d %d %d", &IPp1, &IPp2, &IPp3, &IPp4);

  /* Define a socket and check that it was created ok, if not display OS
     error message on stderr (stdout) */
  my_file_descriptor = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (my_file_descriptor < 0)
  {
    perror("client: socket()");
    return;
  }

  /* Set up servers IP address field */
  memset((void*)&server_address, 0, sizeof(server_address));
  server_address.sin_family = AF_INET;
  server_address.sin_port = htons(PORT);
  address_holder = (unsigned char*)&server_address.sin_addr.s_addr;
  address_holder[0] = IPp1;
  address_holder[1] = IPp2;
  address_holder[2] = IPp3;
  address_holder[3] = IPp4;

  /* request a connection on the socket */
  if (connect(my_file_descriptor, (struct sockaddr*)&server_address, sizeof(server_address)) < 0)
```

```
  {
    perror("client: connect()");
    return;
  }

  /* ask the server a question */
  if (write(my_file_descriptor, request, sizeof(request)) < 0)
  {
    perror("client: write()");
    return;
  }

  /* read the answer from the server */
  if (read(my_file_descriptor, buffer, 255) < 0)
  {
    perror("client: read()");
    return;
  }

  printf("client: answer to the question \"%s\" is \n \"%s\"\n", request, buffer);

  /* close the socket connection */
  close(my_file_descriptor);

  return;
}
```