Networks

Practical Investigation of TCP/IP Ports and Sockets

Gavin Cameron

MSc/PGD Networks and Data Communication
May 9, 1999

TABLE OF CONTENTS

TABLE OF CONTENTS 2
LIST OF FIGURES AND TABLES 3
ABSTRACT 4
INTRODUCTION 5
TCP / IP IN THE OSI MODEL 6
TRANSMISSION CONTROL PROTOCOL (TCP) 7
FORMAT OF A TCP PACKET 7
PORTS AND SOCKETS 9
OPENING A TCP CONNECTION 10
FLOW CONTROL 10
CLOSING A TCP STREAM 11
TCP LINK STATES 11
PROGRAMMING WITH TCP / IP 13
SCENARIO 13
SERVER APPLICATION 14
CLIENT APPLICATION 16
EXECUTION 16
CONCLUSION 18
APPENDIX A - SERVER APPLICATION CODE 19
APPENDIX B - CLIENT APPLICATION CODE 21

LIST OF FIGURES AND TABLES

Figure 1 - Seven layer OSI Model. 6
Figure 2 - TCP Header Format 7
Table 1 - TCP Flag bit definitions. 8
Figure 3 - Opening a TCP Stream. 10
Figure 4 - Closing a TCP Stream 11
Figure 5 - TCP Link States 12

ABSTRACT

This report contains the procedure and results of the laboratory exercise investigating TCP ports and sockets.

The report focuses on the structure of a TCP segment (or packet) and explains how the protocol makes for a reliable two way communication "stream."

The report them implements a practical scenario of TCP/IP ports and sockets using Unix / Linux as the platforms and "C" as the language.

The scenario is fully implemented and test results included within the report.

INTRODUCTION

Transmission Control Protocol (TCP) is a means for building a reliable communications stream on top of the unreliable packet Internet Protocol (IP). TCP is the protocol that supports nearly all Internet applications. The combination of TCP and IP is referred to as TCP/IP and many people imagine, incorrectly, that TCP/IP is a single protocol.

The basic method of operation involves

- wrapping higher level application data in segments
- wrapping the segments into IP datagrams
- associating port numbers with particular applications
- associating a sequence number with every byte in the data stream
- exchanging special segments to start up and close down a data flow between two hosts
- using acknowledgments and time-outs to ensure the integrity of the data flow

TCP / IP IN THE OSI MODEL

TCP and IP fit into the OSI (Open Systems Interconnection) 7 layer model as shown below. The model has been designed in such a way that the layers are independent of each other.

Figure 1 - Seven layer OSI Model

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

The Physical layer provides the physical connection between two hosts which may be an electrical connection, a fibre optic connection, a radio link etc.

The Data Link layer provides the communications protocol between the two hosts, for example voltage levels, timing information, carrier information, parity, Manchester encoding, non-return to zero etc.

These two layers (Physical and Data Link) are generally contained within the one piece of hardware, for example, an Ethernet card.

The IP layer allows the data packets to be routed through, perhaps, several hosts (or networks) to its destination host with the aid of an IP address. This may require the data to be routed through switches, routers, hosts or any other piece of hardware that physically connects two sub-nets together.

The TCP layer allows a reliable structure to the data that is transferred between hosts, i.e. it takes care of block numbering, sequencing and handshaking. This is all achieved with the aid of TCP addresses, referred to as ports and sockets.

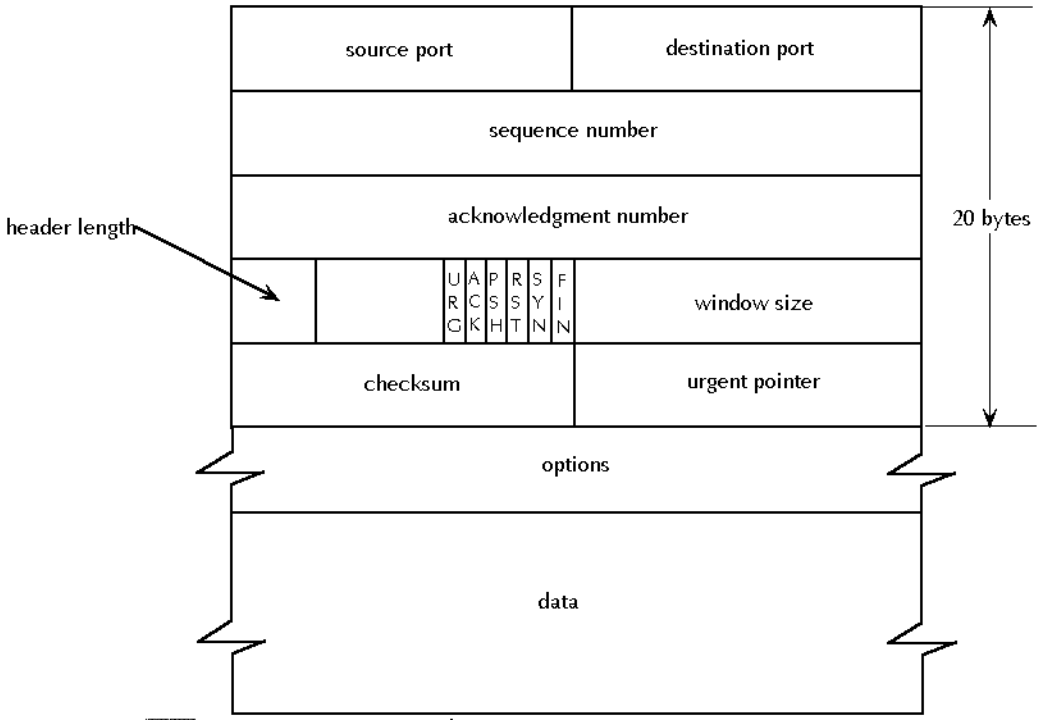
The TCP/IP protocols do not concern themselves with what data they transfer as it is the job of to the upper layers to format the data. Similarly, they do not concern themselves with how the data is physically transmitted, as that is the job of the two layers below them. The only bounding factor is the interfaces between layers. Hence, the TCP layer receives data, bundles it with its own information, passes it onto the IP layer which bundles it again (with *it's* own information) and then passes the data onto the Data Link layer which, with the aid of the Physical layer transmits the data (bundled with the Data Link layer information).

TRANSMISSION CONTROL PROTOCOL (TCP)

FORMAT OF A TCP PACKET

TCP segments are constructed from 32 bit words and include a 20 byte (5 word) header. The basic layout is shown below.

Figure 2 - TCP Header Format



- source port number

  The source (and destination) port numbers are used for de-multiplexing the data stream to applications. It is entirely possible for there to be multiple simultaneous TCP data streams between two hosts. A TCP data stream is uniquely identified by a group of four numbers. These are the two hosts addresses (in the IP information) and the two port numbers.
  The source port number is the one to be used as destination in any reply to the segment.

- destination port number

  This is the "target" port number on the remote system.

- sequence number

  This 32 bit number identifies the first byte of the data in the segment.

- acknowledgment number

This 32 bit number is the byte number of the next byte that the sender expects to receive from the remote host. The remote host can infer that all bytes up to this number minus one have been safely received and the remote host's local copies can be discarded.

- header length

  This 4-bit field specifies the header length in 32 bit words. Clearly the maximum value is 15 words (60 bytes) allowing for 10 (40 bytes) of options.

- flag bits

  This group of 6 bits identify various special states in the protocol. Several of the bits may be set simultaneously. The definitions of the bits are as follows:

Table 1 - TCP Flag bit definitions

URG	Indicates that the Urgent pointer is valid, i.e. there is urgent data.
ACK	The acknowledgment number is valid. This will usually be set.
PSH	The data should be passed to the application as soon as possible. This will typically involve flushing buffers.
RST	Reset the connection. This involves marking the sequence numbers as invalid.
SYN	The synchronize bit is used to establish initial agreement on the sequence numbers.
FIN	The sender has finished sending data. This fact will, normally, be passed on to the application as close.

- window size

  This is the amount of space that the receiver has available for the storage of unacknowledged data. The units are bytes unless the window scale factor option is used. The maximum value is 65535.

- checksum

  This covers both the header and the data.

- urgent pointer

  This is part of TCP's mechanism for sending urgent data that will overtake the normal data stream. If the URG flag bit is set this field indicates the position within the data of the last byte of the urgent data. There is no way of indicating where the urgent data starts.

- options

  There are a number of options defined in various RFCs. The most useful is the Maximum Segment Size (MSS) specification facility.

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[1] system, where a firewall might allow SMTP into a company, but not allow Telnet in, however, let it out.

As mentioned earlier a TCP stream is defined by the source and destination IP address and port number, this is for connections between hosts. Internal to the host, this combination of numbers is assigned a unique number called a socket and it is the socket that any applications wishing to communicate with other hosts talk to.

When a stream is first initiated, the initiator determines a free port to use as well as which port to talk to on the remote host, for example in an FTP session, the initiator may choose port 1025 to listen on and port 21 to talk to on the remote host (which is the default port for FTP). The listening port is generally determined by the operating system - the user applications have no control over this. When the remote host is sending a packet to the initiator, the same port numbers will be used but their positions in the header will change (source / destination).

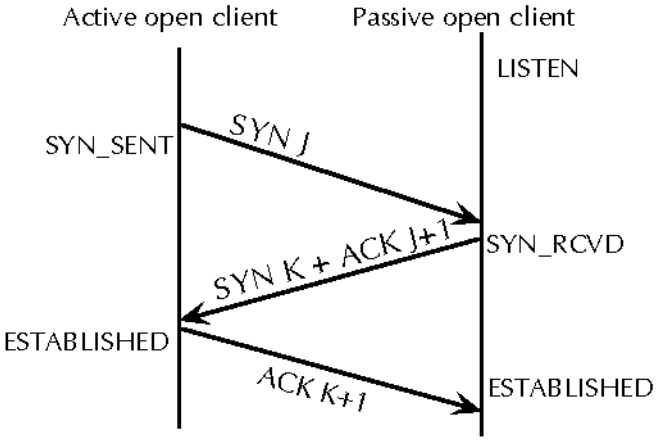
[1] A firewall is a security feature of a companies network infrastructure. It is the only point where the internal intranet is connected to the glabal Internet. The flow of traffic through the firewall can be controlled in both directions in an effort to prevent hackers gaining access to confidential material.

OPENING A TCP CONNECTION

A TCP connection is opened by a three-way handshake to establish a common view of the sequence numbers. A connection will be initiated by an active client, the other end of the connection is described as the passive client, although in terms of the client/server software model this is likely to be a server. The passive client should be in a state known as LISTEN which simply means that it is expecting an incoming connection request.

The three way exchange involves the active client sending a SYN (see table 1) segment with the sequence number set to an arbitrary value (J). The passive client responds with a SYN segment with the acknowledgment number set to J+1 and the sequence number set to a further arbitrary value (K). The active client responds to the SYN segment by sending an ACK segment with the acknowledgment number set to K+1. This is illustrated below.

Figure 3 - Opening a TCP Stream



The "arbitrary" initial sequence number is required (by RFC 793) to increment approximately every 4 µs, this avoids delayed segments from a previous connection getting mixed up with a new connection. The initial sequence number will wrap in about 4½ hours. Once a connection is established the sequence numbers can wrap much more quickly depending on traffic and line speed.

Once the stream has been established, the data can flow as and when required to do so.

FLOW CONTROL

Flow control is associated with the current byte sequence numbers at each end of the data flow. Whenever a segment is sent it includes the sequence number of the last byte sent. A segment will also include the sequence number of the next byte that the sending host expects to receive, this is called the acknowledgment number (ACK). A host receiving a segment can assume that the remote host has safely received all bytes up to and including byte ACK-1, local copies may now be discarded.

The difference between the number of the last byte sent and the acknowledgment number is known as the window. The maximum size of the window is advertised by a host as part of every TCP segment the host sends, a host can quench the flow of data from a remote host by advertising a window size of zero. Once a zero window size advertisement has been received a host can no longer send data. A host may not, under any circumstances, send data with byte sequence numbers greater than the sum of the remote acknowledgment number and the remote window. Under normal

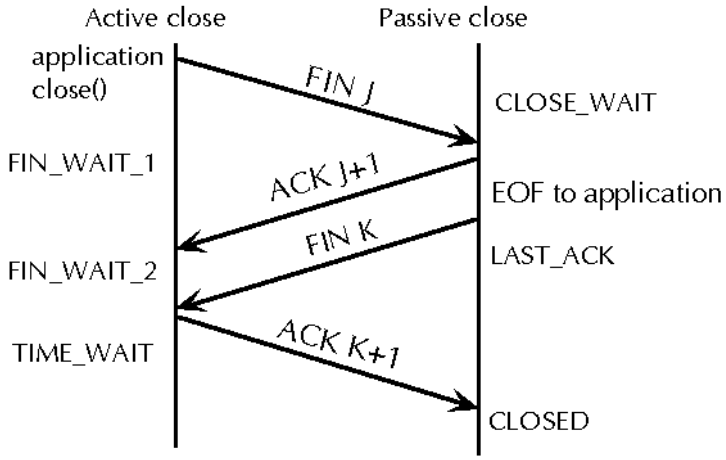
circumstances the remote window can be thought of as a buffer where out-of-sequence segments are held temporarily awaiting the filling in of gaps in the sequence when delayed data turns up.

Window size is not negotiated. It is up to the sender not to over-run the receiver's buffers. A small sender buffer constraint will only mean that the sender cannot take full advantage of the receiver's buffering capabilities.

CLOSING A TCP STREAM

The orderly close down of a TCP connection requires the four way exchange illustrated in the diagram below.

Figure 4 - Closing a TCP Stream



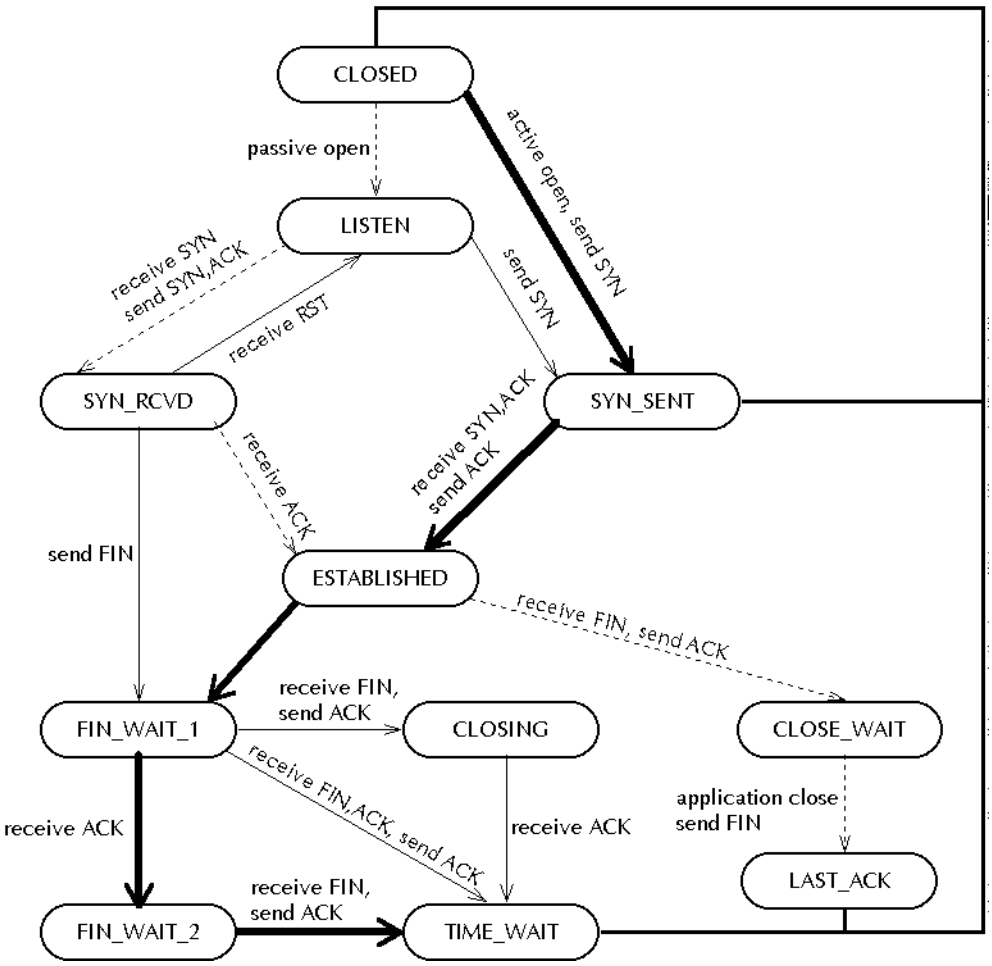
At the active end the application initiates the closure sequence, possibly by a close() system call on a socket. At the passive end receipt of the FIN segment causes the software to pass an "end-of-file" indication to the server software.

It should be noted that the exchange is really two independent exchanges and it is possible to close the connection in one direction but not the other. This is known as a half close and if this is encountered, a time-out (usually of 120 seconds) will close the stream at both ends due to a lack of acknowledgment.

TCP LINK STATES

The behavior of a TCP connection can be shown using a state transition diagram such as that shown below.

Figure 5 - TCP Link States



PROGRAMMING WITH TCP / IP

From a 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 (normally) be a server and a client application. These could be on the same host, or more realistically, on different hosts 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[2], but will run in the foreground as a normal application.

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*.

---

[2] A deamon is a small utility / application that is run as a low priority background task. They are event driven and usually sit in an idle state waiting for an event to trigger them, for example, an http request.

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 host *addr* who's 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 hosts 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* who's 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 are 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 (one of the TCP/IP internet applications) 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 (although each type of machine will have its implementation of the protocol) which is why machines of different types can integrate so easily.

TCP/IP's use is completely transparent to the end user which is what makes a good environment - the user calls up a web page, and it's there, for example.

As can bee seen from the first part of the report, TCP is a very reliable and robust protocol which ensures the integrity of any data transmissions.

APPENDIX A - 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, i.e.. 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;
}
```

APPENDIX B - 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;
}
```