Notes on Computer Networks

Bob Dickerson

January 2005

Preface

These notes formed the main material for a one semester Computer Science course on networks. The course was last taught in the academic year 2005–6. The course was primarily about the Internet, the TCP/IP protocol family. The rest of the preface is part of the original written for the course (or "elective module" as it was called) and it tries to show how the material in these notes relates to the units that made up the course, and references to sections or chapters in books that provide better, or alternative, explanations.

The notes and books

This is a description of the teaching material, its organisation and how it relates to the units in the Open Systems and Networks elective module.

The main material for the module is provided by these notes. The notes try to cover the range of material that I think is appropriate to this course (module), and they are meant to be at a suitable level, ie. depth of treatment of each topic. This means that there is no *required* textbook.

However the notes are written by me (Bob Dickerson) and therefore it is possible that they are: shallow, incomplete, difficult to understand and perhaps wrong. Even if they are not as bad as that it is still very useful to have alternative explanations for some topics so I am recommending some books as supporting material. Since the books are only meant to supplement or clarify the notes you should really only consult relevant sections or chapters of the books *after* reading the notes; this is because they might have a different emphasis and on individual topics have too much or too little material. Because the use of a textbook is just to reinforce the notes it is not compulsory, if you are brave, lazy, or, in fact, the notes are enough, you can try to manage without extra reading. All the following books are quite good, you can use bits of whichever one you want:

1. Douglas E. Comer. *Computer Networks and Internets with Internet Applications*. Prentice Hall, fourth edition, 2003. Good introduction, mainly TCP/IP, some stuff on data transmission.

2. James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, third edition, 2005. Good introduction, has some deeper treatment, no data transmission stuff.

3. L. L. Peterson and B. S. Davie. *Computer networks, a systems approach*. Morgan Kaufman, third edition, 2003. Good introduction, practical implementation examples, mainly TCP/IP, not much on data transmission stuff.

4. William Stallings. *Computer Networking with Internet Protocols*. Prentice Hall, first edition, 2004. Less conventional introduction, more advanced, has special chapters on congestion and quality of service, no data transmission stuff.

5. A. S. Tanenbaum. *Computer networks*. Prentice-Hall, fourth edition, 2003. Very good introduction, wide coverage, some data communications stuff too.

The units

This is a list of the units, links to the related notes and references to chapters or sections in the books. It is possible to vary the order of presentation of topics. In most books (Comer, Tanenbaum and Peterson) they are presented "bottom-up", starting from the lowest level, or (in Kurose and Rose, and Stallings) "top-down", starting with high-level application protocols. Even as I type this I cannot decide how to do it this time . . . , wait while I decide . . . , OK bottom up but with an overview of some general concepts first.

Another choice is whether to include any material on data transmission, this is about how binary data is actually transmitted by "guided" media (wires or fibre optic) or by "unguided" media (wireless). This course (module) doesn't cover the topic. This is a serious, but deliberate omission. There is not space or time to discuss signal propogation, noise, bandwidth, modulation etc. These topics not required for the assessment but if you feel unhappy reading about sending data over network connections without knowing how the bits are actually transmitted you can find some information in the books:

Comer: chapter 4, 5, 6 and 7 deal with data transmission,

Peterson & Davie: no chapter on data transmission, but some stuff about bandwith and latency in chapter 1,

Kurose & Ross: no chapters on data transmission but one section on "Physical media" in chapter 1,

Tanenbaum: chapter 2, about 90 pages on data transmission, quite good,

Stallings: nothing on data transmission,

1. **Introduction: layers and protocols**
   This unit includes a brief overview of what protocols and layers are, and how a message moves down through the layers acqiring different protocol headers. The unit introduces the concepts of:

   - division of responsibility in networking: *layers* that carry out different functions,
   - equivalent layers on different machines called *peers*,
   - *protocols* that allow peer layers on different machines to communicate,
   - *message encapsulation* the way layers attach their own headers to the messages they are asked to pass on by higher layers.

   There is one chapter in the notes Introduction, layers and protocols (chapter 1). Relevant material in the textbooks:

   Comer: these concepts are explained in chapter 16, "Protocols and Layering",

   Peterson & Davie: no separate chapter but the is a section on "Network architecture" in chapter 1,

   Kurose & Ross: two separate sections on protocols and layers in chapter 1,

   Tanenbaum: some stuff on layers in chapter 1,

   Stallings: idea of protocols and layers in chapter 2.

2. **Data link layer and network topologies**
   The data-link layer is responsible for sending packets (lumps) of data between directly connected machines, ethernet, PPP, and wireless 802.11 are data-link protocols. The issues dealt with are:

   - network topologies,
   - the functions of data-link, simple encoding, framing and error checking,
   - how ethernet operates,
   - ethernet bridges, hubs and switches,
   - some stuff on wireless LANs

   The chapter on data-link and ethernet is Data link layer and network topologies (chapter 2). The chapter on wireless LAN is 802.11 Local Area Wireless Networks (chapter 3).

   Relevant material in the textbooks:

   Comer: this topic is covered in Comer's book in chapters 7, 8 and 9. Then chapter 10 deals with physical connecting ethernets, chapter 11 with bridges, chapter 12 and 13 are about longer distance networks and are less relevant.

   Peterson & Davie: direct data-link networks are dealt with in chapter 2 this is relevant to the module, chapter 3 is about more complicated networks like ATM, this goes beyond what is required for the module,

   Kurose & Ross: chapter 5 is "The link layer and local area networks", chapter 6 is about wireless and mobile networks and contains more material than is dealt with in the module,

   Tanenbaum: the treatment of data link is split into chapter 3 called "The data link layer", and chapter 4 called "The medium access control sublayer" which actually contains most of the material about ethernet and wireless. These chapters contain more material than is needed by the module so be guided by the coverage of the notes,

   Stallings: data link is covered in Part 6, the first chapter is 13 on "Wide area networks" which is not really necessary for this module (too "wide"?), chapter 14 "Data link control" about issues in data link is more useful, and chapter 15 on "Local area networks" is relevant too.

3. **Network layer**
   Climbing up one level above data link layer is the network (or internet) layer. This layer conveys a packet across different networks to any addressable destination. This is split into two units, the first about IP, and the second about routing; it is only split to allow more time to cover it. The topics are:

   - IP addressing,
   - packet format,
   - packet forwarding
   - addressing on a LAN (ARP).

   This is covered in the first part of the Network layer chapter 4.

   Relevant material in the textbooks:

   Comer: this topic is covered in chapters 18, 19 and 20. There is additional material about IP fragmentation in chapter 21, interesting but not essential for this module. Chapter 22 is about the new version of IP called IPv6.

   Peterson & Davie: in chapter 4 on "Internetworking" section 1,

   Kurose & Ross: it is in chapter 4, but it is hard to disentangle routing from other aspects of IP. Perhaps read sections 4.1, 4.2 and 4.4 first,

   Tanenbaum: in chapter 5. There is a lot more material than is needed for this module, so maybe just look at sections 5.5 and 5.6,

   Stallings: chapter 8, sections 8.1 and 8.2 are most relevant

4. **Routing**
   This is still at the network layer, it is about how systems discover which connections to use for forwarding packets—routing. Instead of examining the details of real protocols this looks at two algorithms used for discovering routes. I hope to add some additional notes about the real problems of routing on the backbone of the Internet. The topics are:

   - static link-state, or Dijkstra's shortest routes algorithm,
   - dynamic distance vector routing,
   - something about Internet routing (I hope).

   This is covered in the second part of the Network layer chapter in section4.7

   Comer: this is covered in two places, he covers the general routing algorithms in chapter 13, and then deals with IP Internet routing in chapter 27. There is very little about backbone routing,

   Peterson & Davie: more of chapter 4, sections 4.2 and 4.3,

   Kurose & Ross: chapter 4, sections 4.3, 4.5 and 4.6,

   Tanenbaum: chapter 5, section 5.2,

   Stallings: chapter 11 and the chapter 12 section 12.1.

5. **Transport layer**
   This layer is responsible for providing reliable, data-streams, from program to program. It builds this out of the out-of-order unreliable computer to computer datagrams sent by the network layer. Topics:

   - end to end messages using *port* addresses,
   - providing streams from packets,
   - reliability and retransmission,
   - congestion and flow control,

   The chapter in my notes is Transport layer chapter6

   Comer: chapter 25,

   Peterson & Davie: chapter 5, sections 5.1 and 5.2, the later stuff on RPC in 5.3 is not necessary. Chapter 6 is also about transport layer problems but is more than is needed, however 6.3 on TCP congestion control is interesting,

   Kurose & Ross: chapter 3, sections 3.1 to 3.5,

Tanenbaum: chapter 6, sections 6.1 and 6.5,

Stallings: chapter 6, sections 6.1, 6.4 and 6.5

6. **Network programming**
   This describes the basic facilities used by nearly all network applications. These can be used in Java, C++ or any other language. It introduces:

   - the (almost) universal BSD socket interface used by all network applications
   - the asymmetry of client and server programs,
   - the Java classes that provide sockets and how to use them
   - the concept of a *concurrent server*,
   - *threads* in Java and how they can be used to create a concurrent server.

   The chapter in the notes is Network programming (chapter 7).

   Comer: in chapters 28, 29 and 30, but he only provides program examples in C++ not in Java.

   Peterson & Davie: a bit in section 1.3 (in C),

   Kurose & Ross: section 2.7, it does have some Java stuff,

   Tanenbaum: a bit in subsection 6.1.4,

   Stallings: in section 4.4

   other perhaps the simplest way to get extra information about network programming in Java is to look at Sun's Java tutorial and guide:

   http://java.sun.com/docs/books/tutorial/networking/index.html

7. **The application layer: HTTP**
   This says something about one application level protocol (ie. one that runs above and uses the *socket API*). The application is the Web, the core of which is a very simple protocol called HTTP. The chapter in the notes says a bit about:

   - the operation of the HTTPprotocol,
   - the common format of the files (pages) which is currently HTML, and
   - a bit about server-side functionality provided by CGI programs or PHP.

   The chapter in the notes is WWW, HTTP, HTML, CGI and PHP (chapter 8).

   Comer: in chapters 35, 36 and 37, chapter 35 deals with HTTP, chapter 36 deals with server-side functionality like CGI, and chapter 37 covers client-side functionality like Javascript, it is not so important for this module,

   Peterson & Davie: in subsection 9.2.2,

   Kurose & Ross: in section 2.2,

   Tanenbaum: in section 7.3,

   Stallings: section 4.1.

8. **Application layer: DNS etc.**
   This is another application level protocol like HTTP discussed earlier, although it is not an ordinary application, this is the protocol that enables names (eg. herts.ac.uk) to be used on the Internet. Also other application protocols might be introduced for example those supporting email (but these extra notes don't yet exist). The chapter in my notes is The domain name service, DNS, chapter9

   Comer: DNS in chapter 31, mail in 32,

   Peterson & Davie: DNS in section 9.1, mail in subsection 9.2.1,

   Kurose & Ross: DNS section 2.5, mail in section 2.4,

   Tanenbaum: sections 7.1 and 7.2,

   Stallings: section 4.2 for DNS and section 3.3 for mail.

9. **Application layer: P2P etc.**
   This unit considers the characteristics of peer-to-peer networking and how it differs from the client-server architecture. It also looks at an example of a file-sharing peer-to-peer system, Gnutella. Once again, if I finish the notes there will some other protocols considered, for example messaging systems and or real-time protocols. In my notes the chapter is Application layer: P2P (chapter 10)

   Comer: nothing about peer-to-peer protocols but chapter 33 is about a real time problem: Voice over IP

   Peterson & Davie: yes

   Kurose & Ross: section 2.6,

   Tanenbaum: 2 pages in chapter 1,

   Stallings: I can't find anything.

10. **Security**
    The problems of how systems connected can be attacked and how traffic can be intercepted of spied on. The notes say a little about cryptography and how it can be used to provide greater security. In the notes this is in Security (chapter 11).

    Comer: chapter 40,

    Peterson & Davie: chapter 8,

    Kurose & Ross: chapter 8,

    Tanenbaum: chapter 8 is very good but too much. He covers all the relevant topics but provides too much about each,

    Stallings: chapter 16.

Contents

Chapter 1

Introduction: networks, layers and protocols

1.1 Networking

Networking supports communication between two or more programs running on physically distant machines. For example all the following require network support:

- a WWW browser client using a WWW server,
- mail from a user agent program to a remote mail box,
- remote access to a data-base,
- a remote shared file server system,
- downloading an MP3 music file.

1.2 Protocols

To request any service or exchange any information between 2 programs there must be an agreed set of commands and data formats, this is a *protocol*. So, for example, the commands and data sent between a World Wide Web browser and a remote server are a protocol. The browser (probably) uses the GET command follow by the name of the required file (page), this protocol is recognised and understood by the web server program which responds appropriately. Similarly the format of packets sent between Ethernet cards and their drivers are a protocol. The programs exchanging messages are called *peers*.

1.3 Networking layers

Two very important concepts in understanding networking are *protocols* and *service layers*. Figure 1.1 is a simplified view of the layers of network service in TCP/IP.

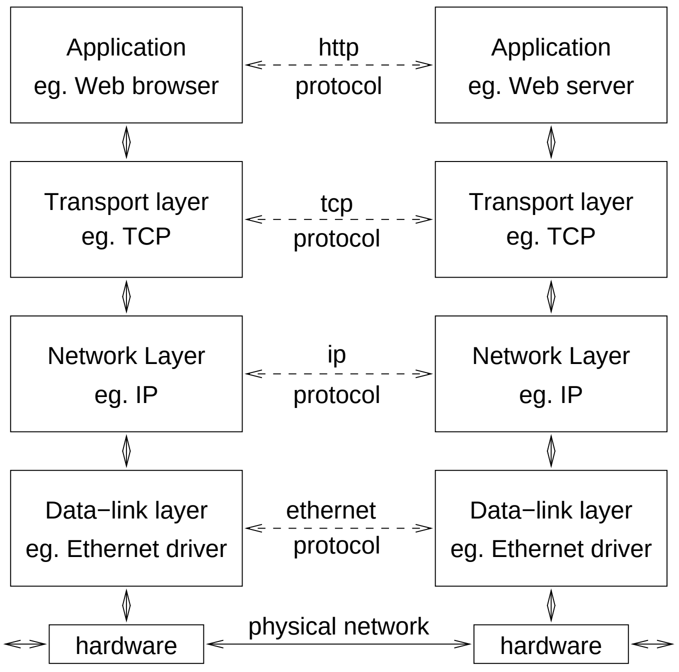


Figure 1.1: Layers and protocols

1.3.1 The functions of the layers

Each layer in the simple model provides facilities and carries out certain tasks:

Hardware Bits of wire that can carry bits?

Data-link This layer is responsible for delivering packets for the network layer to other physically connected machines. It is responsible for error checking and driving the devices. Ethernet is a data-link layer protocol, it can only send packets to machines that are physically attached to the same wire.

Network This "spans" different physical networks, it is a protocol that makes minimal assumptions so it can work on any and all data-link networks. Its job is to get packets from a machine on one physical network to a machine on another—the *inter-network* protocol IP. Its main job is finding and maintaining routes to the remote systems.

Transport This layer turns IP packets into a "stream" of characters between different *processes* on different machines. This layer provides a "reliable" service, if any IP datagrams are lost this layer must recognise this and re-transmit them. The layer guarantees delivery of all the data (for TCP anyway) in the correct sequence by using sequence numbers. This layer provides an interface to the application and supports streams of data (TCP) or arbitrary length single messages (UDP) to selected services on selected systems. The interface it provides is called the *socket* interface.

Application These are either user programs or standard utilities like: ftp, telnet, WWW browsers, network file store, or mail programs, each provides its own application oriented protocol. All of them use the transport layer service.

Usually all the layers upto and including the transport layer are in the kernel of the operating system and the applications are programs. So the interface between is usually a set of system calls.

1.3.2 Why have layers?

One reason for having separate layers is that it makes the system simpler to use by defining clear interfaces for application or protocol developers.

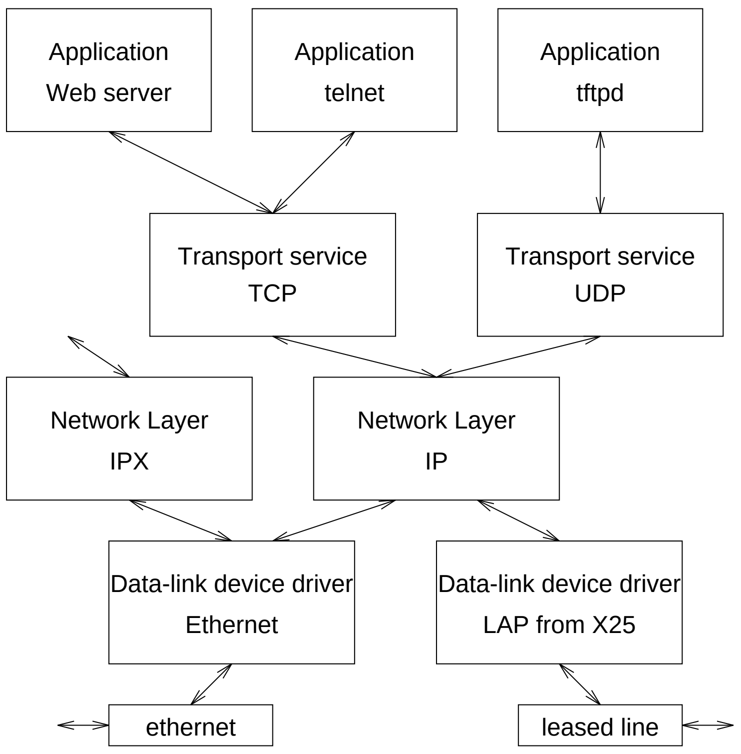


Figure 1.2: Layers with alternative protocols

Another reason is that the separation simplifies the use of alternative layers and protocols so that if the network level determines that one site is connected via a leased line it can pass a message packet to the appropriate driver, whereas a message to a different site will be passed to a different data-link level protocol driver, this is shown in figure 1.2. It also works in reverse: network (IP) packets contain a field in their header identifying whivh transport level protocol they use and this is used to determine which level to pass the packet up to (either TCP or UDP).

1.3.3 Relationship between protocols and layers

If a browser communicates with a web server they exchange messages (using the HTTP protocol), the messages are simple character strings:

- In order for the browser to send the HTTP message it must request that the layer below it (the transport layer) opens a connection to the server on the remote machine.
- The transport layer has to communicate with its *peer*, (the transport layer software on the remote machine) to establish the connection to the web server. Peers at the transport layer use the TCP protocol.
- In order for the transport layer to send its TCP messages it breaks them into “packets” and requests that the network layer below it sends these packets to the remote machine which will pass them up to the peer transport layer.
- The network layer uses the IPv4 (and soon IPv6) protocol. It also uses a routing protocol to work out which machine to send to in order to get the remote end. and it must ask the datalink layer . . .

This is very similar to using the Post Office to convey letters.

- You write to your friend, the letter is your message (what you say in the letter and how they respond is your “protocol”), you put it in an envelope, put the address on the front and pass it down to the next “layer”—the postal service,
- The local postal service sorts the letters and puts them in bags for different destinations, these are labelled. The bags are then given to an airline or a railway that uses the labels to deliver them to the remote postal service,
- The remote postal service unpacks the bags and delivers the letters.

Notice that it is necessary to have a “protocol” that is understood by the lower layer (TCP, or postal service bag labels) in order for messages from a higher level to be delivered. Notice also that the layer below knows nothing about the higher level protocol (whether it is HTTP, or the contents of your letter).

A PACKET’S JOURNEY

Figure 1.3 shows the path of a packet through the network software layers when a client application sends a message to its *peer* (the corresponding server) application. First the application calls on the transport layer on its machine to convey the message to the right program at the destination, the transport layer will use the network layer to send the packet to the correct host, the network layer, once it has found the *next hop* on the journey to the destination, will call the appropriate data link driver to send the packet.

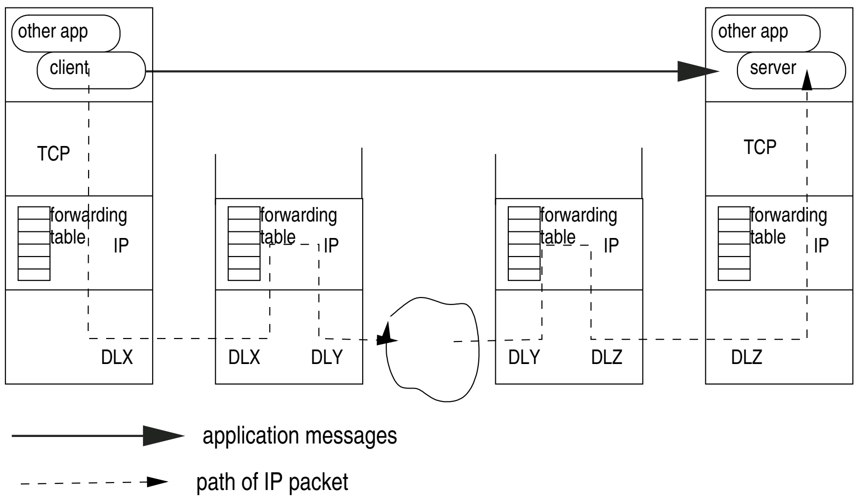


Figure 1.3: Packet encapsulation

When the packet arrives at the next machine the data-link layer passes the packet to the network layer, it examines the packet's destination address, it finds the *next hop* and uses the appropriate data-link driver. This continues until the packet arrives at the destination, then the network layer software will examine the destination address and find that it is its own machine so, instead of forwarding it, it passes the packet up to the transport layer software. The transport layer looks at the transport message and determines which application to give the message to.

1.4 Message encapsulation

As data are passed down from an application level through the transport level, the network layer to the data-link layer they are *encapsulated*, this is shown in figure 1.4. In order to transmit the characters the transport layer puts a header on to communicate with its *peer* module at the remote end. In this header will be the port number. The transport module passes the data plus header to the network module which puts on its header containing the remote system address. Finally when this is passed to the data-link code another header is added.

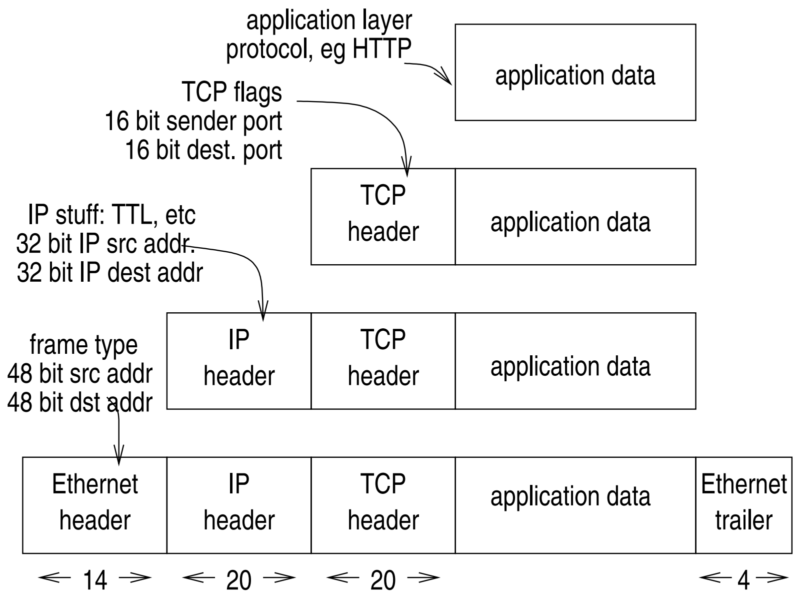


Figure 1.4: Packet encapsulation

1.4.1 Using ethereal to examine packets

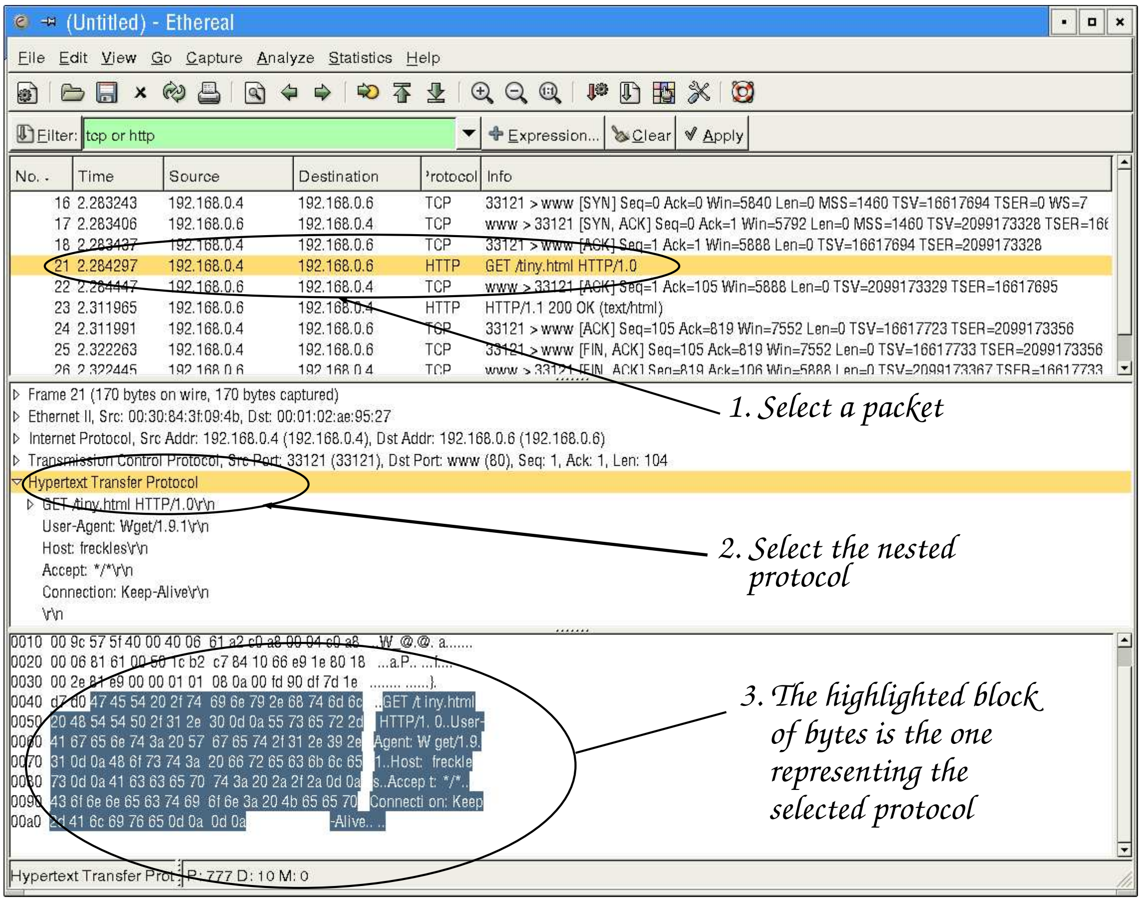
There is a program called ethereal that can "capture" (which means: "take copies of", not "remove") all the raw data data-link packets from a network interface. Since all the higher level protocols are encapsulated in, and carried by, the datalink packet and ethereal can decode all the protocols, it is therefore possible to examine any or all the protocols.

The following pictures (figs 1.5 and 1.6) of ethereal have a lot of detail but most should be ignored, the only concept being examined is packet encapsulation: one message, wrapped inside another.

In figure 1.5 the top window shows a list of packets that were captured, one packet has been selected, it is circled. More details of the selected packet are displayed in the middle window, Remember that each "layer" of networking software has its own task and must communicate with the equivalent layer at the recipient, so it attaches its own header. The middle window shows a decoding of each layer's header, each can be "opened" (using the arrowhead at the left) to get more details, here the application layer protocol, HTTP, has been opened.

In the bottom window there is a hexadecimal dump of the whole raw packet including all protocol headers and data. When one of the protocols is selected in the middle window the corresponding section of the hex dump is highlighted, in the first picture the HTTP protocol is selected so the final (most nested) part is highlighted. But in the second ethereal picture the IP protocol is selected in the middle window and so, in the bottom window, only 20 bytes (the IP packet header length) are hightlighted.

The second picture in figure 1.6 shows the selection of the IP header in the middle window and the highlighting of a different section of the hexadecimal dump in the bottom window.


(Untitled) - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter: tcp or http
Expression... Clear Apply
1. Select a packet
2. Select the nested protocol
3. The highlighted block of bytes is the one representing the selected protocol

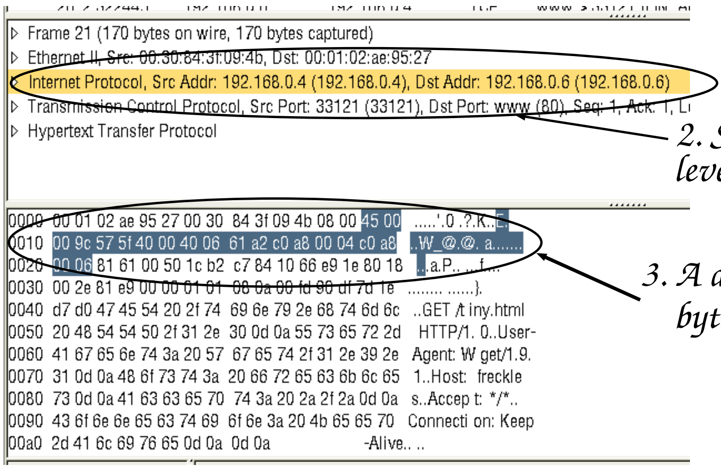
```
No. -  Time      Source        Destination   Protocol Info
    16 2.283243  192.168.0.4   192.168.0.6   TCP      33121 > www [SYN] Seq=0 Ack=0 Win=5840 Len=0 MSS=1460 TSV=16617694 TSER=0 WS=7
    17 2.283406  192.168.0.6   192.168.0.4   TCP      www > 33121 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=2099173328 TSER=16
    18 2.283437  192.168.0.4   192.168.0.6   TCP      33121 > www [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSV=16617694 TSER=2099173328
    21 2.284297  192.168.0.4   192.168.0.6   HTTP     GET /tiny.html HTTP/1.0
    22 2.284447  192.168.0.6   192.168.0.4   TCP      www > 33121 [ACK] Seq=1 Ack=105 Win=5888 Len=0 TSV=2099173329 TSER=16617695
    23 2.311965  192.168.0.6   192.168.0.4   HTTP     HTTP/1.1 200 OK (text/html)
    24 2.311991  192.168.0.4   192.168.0.6   TCP      33121 > www [ACK] Seq=105 Ack=819 Win=7552 Len=0 TSV=16617723 TSER=2099173356
    25 2.322263  192.168.0.4   192.168.0.6   TCP      33121 > www [FIN, ACK] Seq=105 Ack=819 Win=7552 Len=0 TSV=16617733 TSER=2099173356
    26 2.322445  192.168.0.6   192.168.0.4   TCP      www > 33121 [FIN, ACK] Seq=819 Ack=106 Win=5888 Len=0 TSV=2099173367 TSER=16617733

▷ Frame 21 (170 bytes on wire, 170 bytes captured)
▷ Ethernet II, Src: 00:30:84:3f:09:4b, Dst: 00:01:02:ae:95:27
▷ Internet Protocol, Src Addr: 192.168.0.4 (192.168.0.4), Dst Addr: 192.168.0.6 (192.168.0.6)
▷ Transmission Control Protocol, Src Port: 33121 (33121), Dst Port: www (80), Seq: 1, Ack: 1, Len: 104
▽ Hypertext Transfer Protocol
  ▷ GET /tiny.html HTTP/1.0\r\n
    User-Agent: Wget/1.9.1\r\n
    Host: freckles\r\n
    Accept: */*\r\n
    Connection: Keep-Alive\r\n
    \r\n

0010  00 9c 57 5f 40 00 40 06  61 a2 c0 a8 00 04 c0 a8   ..W_@.@. a.......
0020  00 06 81 61 00 50 1c b2  c7 84 10 66 e9 1e 80 18   ...a.P.. ...f....
0030  00 2e 81 e9 00 00 01 01  08 0a 00 fd 90 df 7d 1e   ........ ......}.
0040  d7 d0 47 45 54 20 2f 74  69 6e 79 2e 68 74 6d 6c   ..GET /t iny.html
0050  20 48 54 54 50 2f 31 2e  30 0d 0a 55 73 65 72 2d    HTTP/1. 0..User-
0060  41 67 65 6e 74 3a 20 57  67 65 74 2f 31 2e 39 2e   Agent: W get/1.9.
0070  31 0d 0a 48 6f 73 74 3a  20 66 72 65 63 6b 6c 65   1..Host:  freckle
0080  73 0d 0a 41 63 63 65 70  74 3a 20 2a 2f 2a 0d 0a   s..Accep t: */*..
0090  43 6f 6e 6e 65 63 74 69  6f 6e 3a 20 4b 65 65 70   Connecti on: Keep
00a0  2d 41 6c 69 76 65 0d 0a  0d 0a                     -Alive.. ..

Hypertext Transfer Prot  P: 777 D: 10 M: 0
```

Figure 1.5: Ethereal windows


2. Select a different level of nested protocol
3. A different block of bytes is highlighted

```
▷ Frame 21 (170 bytes on wire, 170 bytes captured)
▷ Ethernet II, Src: 00:30:84:3f:09:4b, Dst: 00:01:02:ae:95:27
▷ Internet Protocol, Src Addr: 192.168.0.4 (192.168.0.4), Dst Addr: 192.168.0.6 (192.168.0.6)
▷ Transmission Control Protocol, Src Port: 33121 (33121), Dst Port: www (80), Seq: 1, Ack: 1, L
▷ Hypertext Transfer Protocol

0000  00 01 02 ae 95 27 00 30  84 3f 09 4b 08 00 45 00   .....'.0 .?.K..E.
0010  00 9c 57 5f 40 00 40 06  61 a2 c0 a8 00 04 c0 a8   ..W_@.@. a.......
0020  00 06 81 61 00 50 1c b2  c7 84 10 66 e9 1e 80 18   ...a.P.. ...f....
0030  00 2e 81 e9 00 00 01 01  08 0a 00 fd 90 df 7d 1e   ........ ......}.
0040  d7 d0 47 45 54 20 2f 74  69 6e 79 2e 68 74 6d 6c   ..GET /t iny.html
0050  20 48 54 54 50 2f 31 2e  30 0d 0a 55 73 65 72 2d    HTTP/1. 0..User-
0060  41 67 65 6e 74 3a 20 57  67 65 74 2f 31 2e 39 2e   Agent: W get/1.9.
0070  31 0d 0a 48 6f 73 74 3a  20 66 72 65 63 6b 6c 65   1..Host:  freckle
0080  73 0d 0a 41 63 63 65 70  74 3a 20 2a 2f 2a 0d 0a   s..Accep t: */*..
0090  43 6f 6e 6e 65 63 74 69  6f 6e 3a 20 4b 65 65 70   Connecti on: Keep
00a0  2d 41 6c 69 76 65 0d 0a  0d 0a                     -Alive.. ..
```

Figure 1.6: Highlighting a different header

1.5 The OSI and TCP/IP layers

There is another (less used) view of layers called the ISO Open Systems Interconnection:

7–5	application or user-process
4	transport
3	network
2–1	data-link & hardware

7	application
6	presentation
5	session
4	transport
3	network
2	data-link
1	hardware

The TCP/IP can be seen as a simplification of the OSI levels:

- The service level, 7–5 merged as the process or application layer. They provide FTP, Telnet, NFS, X11 and other higher level protocols.
- The transport layer, (the OSI layer 4) the link between different processes on different systems, the bit provided by TCP.
- The network layer (OSI layer 3), that links systems across one or more networks, it provides *internet* working. The IP bit.
- The data-link layer, (OSI layers 2 & 1). It is a network, for example Ethernet with its hardware and low-level protocols for moving data between 2 directly connected systems.

1.6 Networks and internets

Networks might be campus networks, company networks, national or local. But in TCP/IP terms a network is most easily though of as a collection of hosts joined directly together at the data-link level. So those systems directly connected to a common Ethernet constitute a network, or some PCs connected via a token ring are a network. Therefore the Hatfield campus has more than one network, even though it is sometimes referred to as one and treated as such for network administrative reasons. A group of interconnected networks is called an *internet*; the most famous and largest internet, that grew from ARPA-net, is called *the* Internet. The Hatfield internet is in turn connected to the UK Universities national network Janet and, in turn, to the Internet.

Chapter 2

The data-link layer

2.1 Functions of data-link layer

The *data-link* layer, in networking software, is reponsible for transferring data from one machine to another directly connected machine. In other words, the networking layer above will pass it packets of data and the name of a network interface and it must transmit the data. This layer must know how to drive the hardware. In different systems the responsibilities might vary but could include:

- encoding
- sending, receiving and *framing* data (all protocols),
- error checking using CRC (*cyclic redundancy checks*),
- error recovery: acknowledgement and re-transmission (in HDLC but not Ethernet).

In many types of network there is a big variation between how much is done by hardware and how much by software, for example an ethernet card will include lots of the functions, but software must do most of the work of driving a dial-up modem line. These notes will examine the logical problems (not electrical issues) whether the functions are in a software of hardware device driver.

2.2 Topologies

The data-link level software in a computer must send data along different physical networks that its computer is connected to. The *topology* of a network is its basic architecture, how components are logically connected. The simplest and oldest (and still widely used) is the *point-to-point*. A system can be build from an arbitrary number of dedicated machine to machine links.

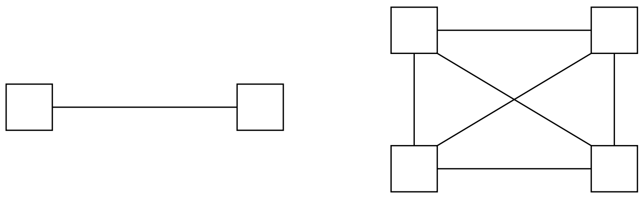


Figure 2.1: Point to point connection

Point-to-point connections like simple serial or parallel lines that join a device on one machine to a device on another, these are commonly used to connect to wide area networks, for example BT leased lines or simple dial-up telephone links. The technology and speed can vary from simple serial lines like RS232 at 9.6 Kbps. to fibre optic cables at 2.5 Gbps. A protocol used on dialup lines PPP. A protocol used for long distance backbone connections is SONET.

- some long distance links, dial-up modems, joining 2 parallel ports (laplink), institutional network to an exchange (our off-site link),
- simple, no addressing needed, if a machine sends on one link it only has one destination,
- Advantages: robust: one lost link only affects that link, no contention: can have all machines communicating at the same time, flexible: different technologies can used for different links,
- BUT scales very badly, there are an exponential number of required connections.

The *star* network, all machines are connected through a dedicated switch:

These are typically used for local area nets and work at about 150 Mbps or more. Actually they may provide the data-link layer but they share some of the characteristics of the network layer.

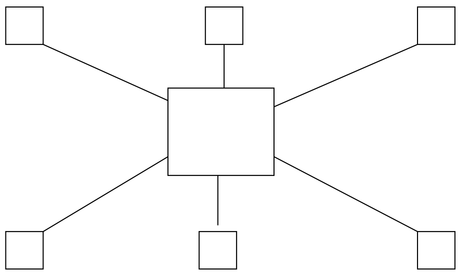


Figure 2.2: Star topology

- like ATM (there is one at Hatfield, in the middle of lots of ethernets), can be used for local or metropolitan or wide area nets,
- more scalable, fewer connections,
- the switch might provide some concurrent connections but it is less parallel than point-to-point,
- needs some form of addressing, so virtual circuits can be set up between communicating machines or packets can be directed to the correct recipient,

The *shared bus* topology, all machines connect to a common carrier,

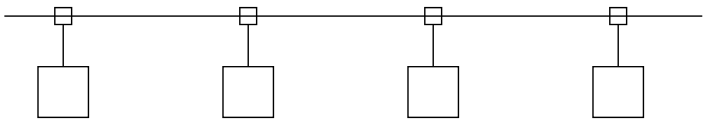


Figure 2.3: Multiaccess shared bus topology

Multi-access nets where lots of machines are connected to the same carrier cable (it works a bit like a computer bus). These are the commonest for local area networks. The different types include *token rings* like FDDI or single lines like Ethernet. Their difference lies in the way they compete for and schedule access to the common carrier between the different machines. There performance is between 10 and 1000m bps. The performance of some ethernets is over 1Gb, these use a similar protocol but they are not really shared bus architectures.

- used for local-area networks, the famous ethernet, not used for metropolitan or wide-area nets,
- very simple, very scalable, very cheap
- requires hardware addresses so the receiver can recognise its data,
- lots of contention, only one message between two systems at any time, requires a fast medium

The *store-and-forward packet switched* network, the switches are high performance purpose built boxes (by CISCO or 3COM or ..), they link with arbitrary toplogies to other switches OR they have “outside” links to host computers, or other networks.

- very expensive, used for wide-area networks or metropolitan nets, they form the backbone of large internets so they need inter-switch connections and ways of connecting to other nets.
- they usually work by switching *packets* of information, which can be briefly stored and forwarded when a link is free,
- they must do routing: how to get from a machine or LAN on one side to a LAN or machine on the other side,

2.2.1 Note on real topologies

The preceding descriptions of topologies are over-simplified logical structures. In reality there are many variations and alternatives, and sometimes a difference between the apparent physical topology and the logical topology of operation of the network. For example:

- many store and forward WAN are made out of multiple point-to-point connections,

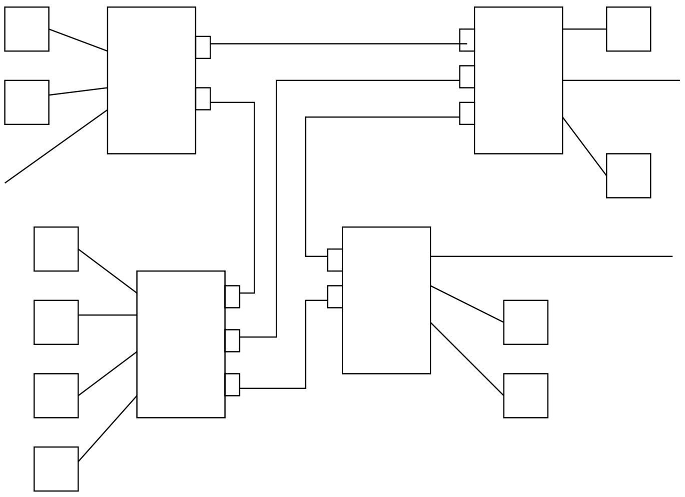


Figure 2.4: Store and forward WAN topology

- ATM networks can be connected to produce a structure that doesn't look like a star but resembles the store-and-forward organisation,
- 100Mb ethernets that use *hubs* (more later) to connect them look like physically like a star but really do function as a broadcast shared bus toplogy,
- 100Mb ethernets that use *switches* (more later) to connect them look like physically like a star AND really do function as a star network NOT a shared bus architecture,

2.3 Data transmission

The first problem is how are "bits" of digital data sent, this is the problem of data transmission. This is an enormous subject that will not be dealt with here. It includes:

- the data transmission medium: radio signals, copper wires (twisted or not), fibre optic cables etc.,
- the performance of the different media and their properties,
- the problem of "noise" and how much information can be sent. This is a big topic and can involve quite a lot of mathematical analysis,
- how data are represented: amplitude modulation, one signal strength for a "1" and a different signal strength for "0", phase modulation using a sine wave and changing the phase of the oscillation where the change represents a bit, or frequency modulation using a sine wave and changing the frequency of oscillation to indicate a bit.

Just ignore this for now, but you must know that the topic of data transmission is a major subject in its own right and an area of overlap between the concerns of electrical and electronic engineers and computer scientists. We will only assume that some how ones and zeros can be represented and transmitted.

2.4 Encoding

To send a binary digit along a carrier the sender can vary the voltage or frequency for a fixed period of time, the receiver must detect this change. To do this they must synchronize clocks so the receiver samples at the right time and duration.

The clock is probably a transition from one level to another and triggers the sampling of the line. If the line is at one level to long then the clocks at each end might drift.

There are various forms of encoding:

- NRZ low level for 0, high for 1. But the signal can stay too long in one state.
- NRZI change level for a 1, unchanged signal for 0. Solves problem for 1s but not 0s.
- Manchester encoding which does an XOR of the bit with the clock signal (which changes *every* interval). Clearly produces lots of transitions but clearly only provides half the bit rate for any Baud rate (the maximum number of transitions the line can make in a second).

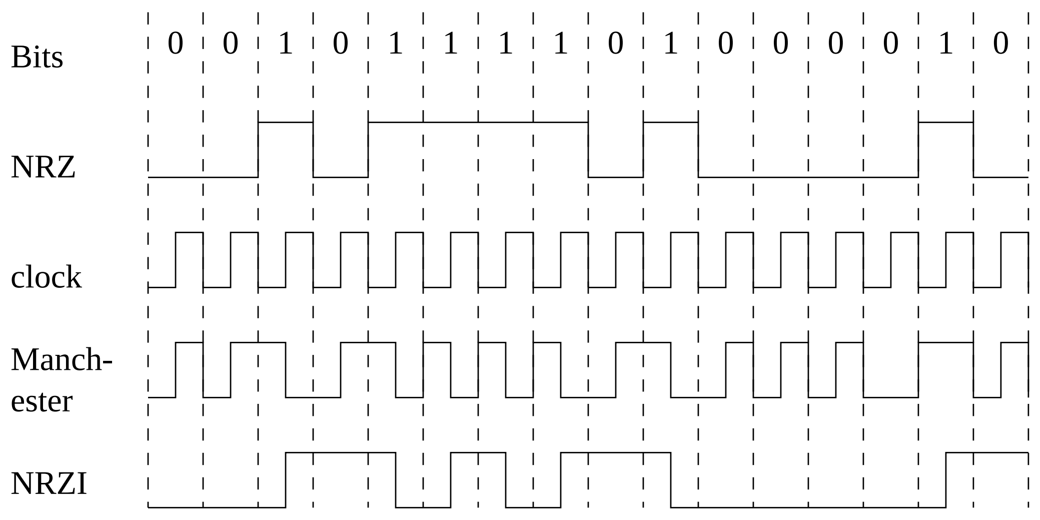


Figure 2.5: Simple digital encoding

- 4B/5B, every 4 bits of data are encoded in 5 bits of signal: `0000` as `11110`, `1111` as `11101`, `0001` as `01001`. The codes are chosen to guarantee that there can be no long sequence of 1s or 0s no matter what the data is. FDDI uses this.

2.5 Error detection

Electrical signals can be corrupted or misread so it is necessary to have a way of detecting any corruption. This is usually done by computing and sending *redundant* information, the receiver recalculates and checks. The amount of redundant information and how it is calculated affect the likelihood of detecting errors.

- Parity, add one extra bit for every byte (or whatever) so there is an even (or odd) number of 1s. Not very strong.
- Checksum, add up all the bytes in a message and send the sum. Better.
- CRC (cyclic redundancy check), treat n bits of data as being represented by an $n-1$ bit polynomial, divide this by some smaller (carefully chosen) polynomial and use this to check. (I don't understand the maths!). This can give quite strong checking of upto 12000 bits with just 32 bits of redundancy.

2.6 Framing

How are bits of data sent? The receiver needs to know how to interpret the sequence. One bit by itself provides little information, it is necessary to send sequences of bits to represent useful data. The solution is to send data in *frames* with a given format. The next problem is to know when the sequence, the frame, starts and when it ends, there are three main ways:

- always send a fixed size frame, this is used by fast backbone network protocols like SONET where there is always loads of traffic,
- start with a marker pattern (a special byte) so the receiver will find the start, then be followed by a byte count, then the data. This is not so often used because it can be hard for the receiver to recover if there is an error in the count (so it is said). All the bits must be sent as bytes so the counting can work. One such protocol was DDCMP used by DEC. More commonly:
- send a special marker (a sequence of bits), then the data and terminate the sequence with another (or the same) special sequence. These protocols can be either *byte-oriented* or *bit-oriented*: bits can be sent as bytes (always multiples of 8 bits), or as an arbitrary sequence of bits representing binary or character data. So PPP is a byte oriented protocol (always multiples of 8) and uses the special byte `01111110` as both the start and end marker. IBM designed SDLC for medium distance links and it was later standardised as HDLC, it is bit oriented, it uses the bit sequence `01111110` (like PPP) as both start and end markers. Figure 2.6 is an HDLC frame.

With any method that uses an end marker there is a problem that if the value of the end marker character or byte sequence occurs in the data being transmitted then the receiving hardware will believe that the frame has ended prematurely. To solve the problem with byte oriented protocols a technique called *byte stuffing* is

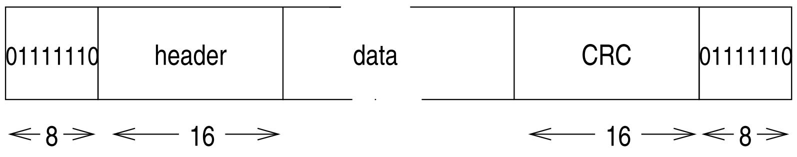


Figure 2.6: HDLC packet format

used: a special *escape* character (DLE in ASCII) is used. Whenever the end marker value occurs in the data it is replaced by the escape character followed by a code indicating that the end marker was replaced. When the receiver detects the escape it removes it and the following character and replaces it with the original code required. (If the value of the escape occurs in the input then it will be replaced by some other escape sequence. NB this is just like the use, in C, of the \ escape character, where `\n` is a newline `\\` is \ etc.)

Things are simpler with bit oriented protocols, they use *bit stuffing*. The sender, to avoid the termination sequence, for example `01111110`, being sent in the data, will if five ones occur (`11111`) just stick in an extra 0. The receiver will be given the data and will remove any zero that occurs after five ones. It is now OK because the start and end markers are the only things that will have six ones.

2.7 Reliable transmission

Depending on the networking system being used, it might be important for the data-link layer to be reliable (not in the TCP context, but maybe others). The simplest solution is to used an *acknowledgement*, *timeout* and *retransmit* system. This is done in the HDLC protocol. It will not be described here because it is dealt with in chapter 6 on TCP.

2.8 Local area networks including ethernet

There have been many forms of local area network architecture: token ring, FDDI, ATM, ethernet and now wireless networks. However the one used most widely is ethernet (and increasingly wireless).

2.8.1 Standards

The IEEE, American Institute of Electrical and Electronic Engineers, has many standards that have become international standards, (the “Unix” standard called POSIX is an IEEE standard). IEEE have a set of standards called 802 that cover many aspects of local area networks (and some wider network issues):

802.2	logical link layer, interface to layers above
802.3	CSMA/CD, the ethernet family, many sub-standards
802.3u	100Mbps ethernet
802.3z	1000Mbps ethernet
802.5	token ring network
802.11	wireless LAN
802.11x	802.11a, 802.11b, 802.11g etc. different wireless frequencies

in the 802 family there is an important distinction between:

- the LLC, the logical link control sub-layer, which specifies the interface to the network layer in the protocol stack. This is independent of the underlying network type and will be the same for all. And
- the MAC, medium access control sub-layer, which specifies the operation of the protocol, data format and data transmission. This is medium dependent and will be different for different network types.

This distinction is used in the 802.11 wireless protocol, the network layer (usually IP) communicates with the LLC layer which then passes LLC frames down to the 802.11 MAC layer. However this distinction is not made by ethernet (802.3) because its design pre-dates the introduction of 802.2. So ethernet packets do not encapsulate or contain LLC packets, higher levels (like IP) interact directly with 802.3 not with 802.2.

2.8.2 802.3 (ethernet) features

Ethernet is a form of *carrier sense multi-access* network with *collision detection* or CSMA/CD, “Ethernet” was a brand name belonging to Xerox but it is so common it is nearly always used as the name instead of CSMA/CD.

Since many machines can connect to the same Ethernet cable they have to use source and destination addressing. The address is 48 bits long and is built into each Ethernet card or device when it is manufactured and is assumed to be unique. An Ethernet packet contains a preamble which is a standard recognisable sequence of bits so that devices detect the start of a packet, the destination and source addresses, a field

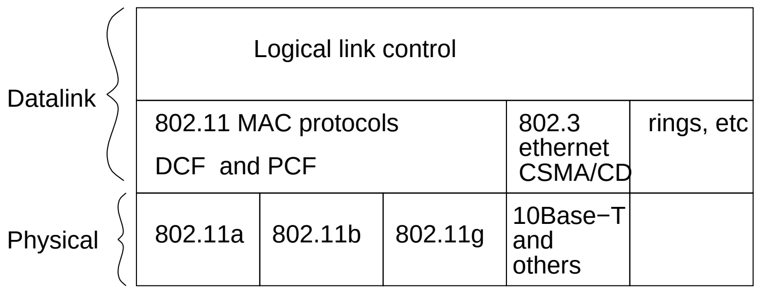


Figure 2.7: IEEE 802 protocol stack

identifying the protocol of the message in the data, ie. IP or something else, so it can be passed to the right layer above.

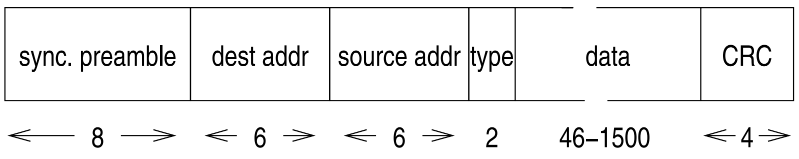


Figure 2.8: Ethernet packet format (sizes in bytes)

2.8.3 Ethernet operation

Another problem arising from having lots of machines on the same cable is synchronising the use of it, when one device puts a packet on the cable no other machine can. In other words "collisions" can occur and must be dealt with. The operation of sending is as follows:

1. if the carrier is busy (ie. some other computer is sending) then wait, or,
2. if the carrier is idle then start sending bits,
3. while sending, monitor the carrier to see if any other bits appear, if not then done.
4. otherwise there is a collision, some other device transmitted at the same time; if so stop, put error bits on the carrier (*jamming* signal) so all other devices know there is an error and then wait a variable time before going to step 1. The length of delay is random and increases with repeated collisions, it is called *exponential back-off*.

Ethernets are very successful and very widely used but they perform very badly if they get much more the half their load. This is because the rate of collisions rises exponentially as the load increases, and also the consequent increase in re-transmissions.

2.8.4 Ethernet cable length

The method depends on a host being able to detect the collision before it stops sending, otherwise a collision might have occurred at the receiver but the sender will not realise and not re-transmit. Consequently there is a maximum length for a 10Mbps ethernet network of 2500m *and* a minimum length of frame of 512 bits (64 bytes). Assume the worst case:

- the sender is at one end of a 2500m cable, and a second sender is at the other end,
- the sender transmits at time t,
- the frame starts to arrive at the other sender at time $t+d$, where d is the latency (time to reach the other end), just after the second sender started to transmit,
- now the second sender will detect the collision and jam
- it will require another d micro-seconds for the second sender's message to arrive at the first sender, at time $t+2*d$, the first sender must still be transmitting at this time or it will not detect the collision.

The time, *d*, taken for a bit to travel 2500m is 25.6 micro-secs, so the first sender must be still be sending after *2*d*, 51.2 micro-seconds, On a 10Mbps ethernet 512 bits are transmitted in 51.2 micro-seconds so in order to still be sending and detect the collision the *minimum* packet length must be 512 bits.

This problem still applies for 100Mbps and 1000Mbps ethernets, they have maximum cable and minimum packet size limits. They also use additional ways to detect collisions, but the basic problem is the same. So the 100Mbps system using hubs and switches and running 10 times faster can either have a minimum frame length of 5120 bits *or* a maximum length of 250m, it shortened the maximum length.

2.8.5 Ethernet bridges

A *bridge* is a way of joining two or more ethernets. It appears to the connected hosts that there is only one network, they address, transmit and receive data in the same way, it doesn't affect them if the receiver is on the same or the other side of the bridge. The bridge works by receiving all packets from all networks, buffering them and passing them on to the other networks. This has the very important consequence that the combined networks can be more than 2500m. This is because the bridge deals with the carrier sense, collision detection and, if necessary, re-transmission on the other ethernets.

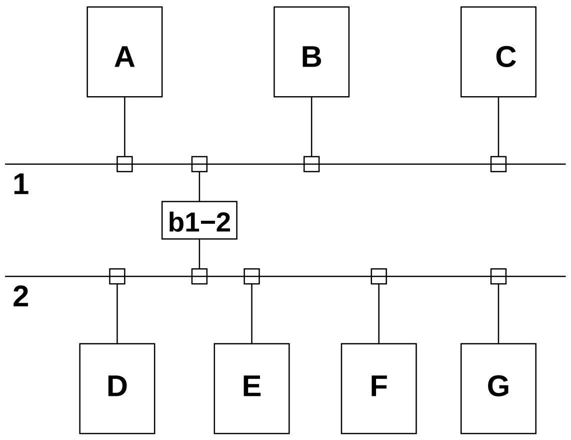


Figure 2.9: Ethernet bridge

So if host A on ethernet 1 sends a packet to host F using F's address it will be intercepted by the bridge b1-2 (because it grabs everything), retransmitted unchanged by the bridge on ethernet 2, and finally get to F.

Most bridges are *adaptive learning bridges*. Their basic operation is the same but they also record all the *sender* addresses of all the packets sent on each ethernet, this way they learn which ethernet each host is attached to. Then, when the must pass on a packet, they examine the *destination* address and only forward it to the network that the destination host is on. So if host C sends to host A it will be intercepted by the bridge but it will not be forwarded on network 2 because the bridge has learnt that host A is on network 1.

2.8.6 Ethernet physical topologies

The basic original topology of the 10Mbps ethernet was the shared bus structure, a coaxial cable, to which every host is attached, see figure 2.10.

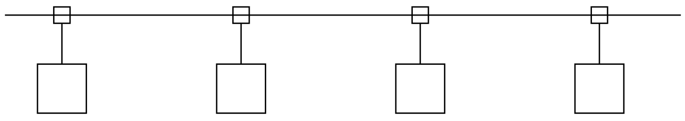


Figure 2.10: Original ethernet topology

The 100Mbps uses UTP (twisted pair) cables that plug into a a box, either a *hub* or a *switch*. The hubs or switches can be connected together in a hierarchy or using 10Mbps links, see figure 2.11.

this looks like a star network topology, it is physically but not logically. Logically and functionally it is still a shared bus. When one host sends a packet it goes to all the other hosts.

Notice, in figure 2.12, that the link goes up the twisted pair, into the hub, back down one link in the next twisted pair and back to the hub again. In other words it works exactly like the shared bus. Hubs can have between 4 and 64 ports.

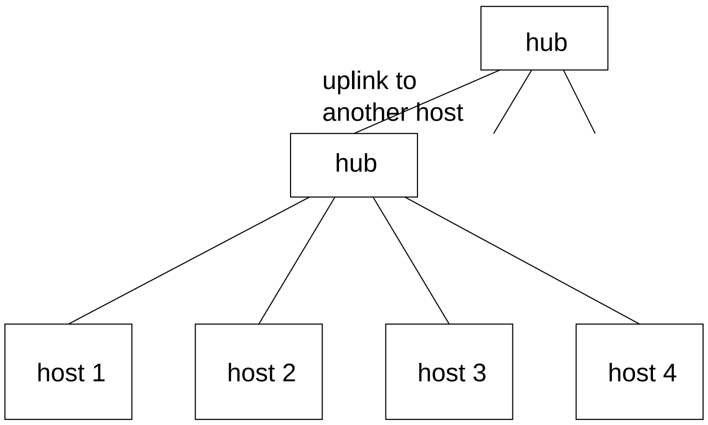


Figure 2.11: Ethernet hub

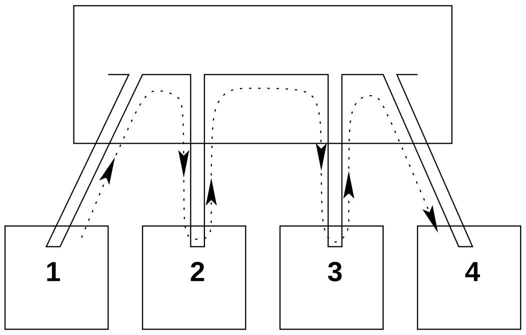


Figure 2.12: Inside an ethernet hub

With a hub there is still contention, while one host is using the hub no other host can. By spending a bit more money you can get a *switch*. A switch looks like a hub but internally it is totally different. A switch still appears the same as any ethernet to the host but it is almost as every host is on its own separate ethernet with bridging between them, see figure 2.13.

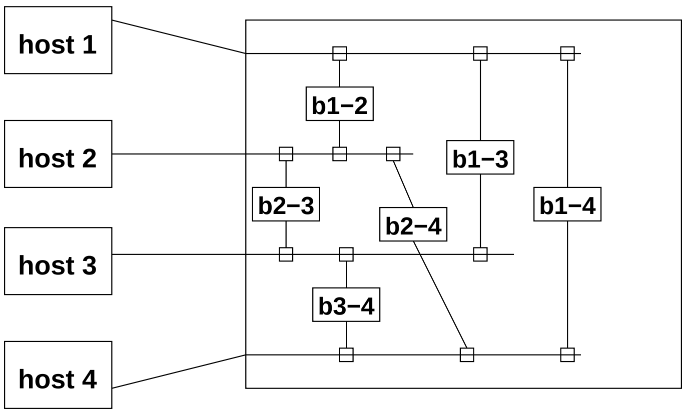


Figure 2.13: An ethernet switch

So if host 1 is sending to host 3 the packets go through a type of internal adaptive bridge b1-3 and because b1-2 and b1-4 are adaptive they will not forward the packet. This means that host 2 can communicate with host 4 at the same time without collisions.

Chapter 3

802.11 Local Area Wireless Networks

3.1 The 802.11 standard

There are various forms of “wireless” networking, they use different frequencies, they work over different distances, they use different techniques and they are used for different types of network. There are long distance links using micro-waves, they are infra-red links between laptops and desktop machines and there are wireless local area networks based on the IEEE 802.11 standard (the one considered here). The standard is a *data-link* protocol, it defines:

- the services and behaviour provided to the layer above (to the *network* layer), hiding the lower details, this is common to all 802 LAN standards (like ethernet, rings, etc.),
- the MAC (*medium access control*) protocols, ie. how the connected systems cooperate together to exchange data. This includes messages to support movement of one station between cells (networks) and support for authentication and privacy,
- it also specifies hardware behaviour, frequencies, encodings, modulation etc.

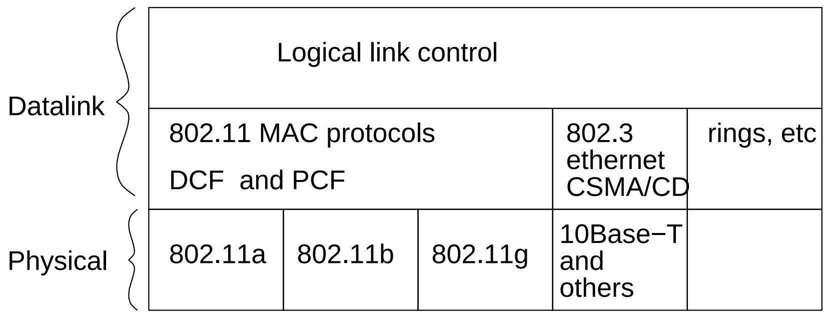


Figure 3.1: 802 Protocol layers

There are various alternative 802.11 standards: 802.11 upto 2Mbps, 802.11a (using orthogonal frequency division multiplexing) upto 54Mbps, 802.11b (using direct sequence spread spectrum) upto 11Mbps, and 802.11g upto 54Mbps. They all have similar MAC protocols and only differ in the hardware behaviour.

3.2 802.11 architecture

A *cell* is a group of *stations* (computers) that can communicate with each other using wireless transmission. A cell is also called a BSS, *basic service set* in 802.11.

A cell can have an *access point*, AP (often called a “base station”), which connects it to another network, usually a LAN like ethernet. The LAN to which a cell is connected is called a distribution system or DS. A cell with an AP connection is called an *infrastructure* BSS. Both cell A and cell B in figure 3.2 are infrastructure BSSs.

A cell with no AP is called an *independent* BSS, also sometimes called an *ad hoc network*. In figure 3.2 stations 11 and 12 are part of an independent BSS.

In picture 3.2 stations 2 and 3 can communicate directly in cell A, in cell B stations 6 and 8 are too distant but can communicate via the base station. All the stations in cell A and cell B can communicate with the rest of the world using their APs and the DS.

3.2.1 Connection between wireless and ethernet

How does the AP access the DS? How do packets from the wireless network travel via the AP over the ethernet? They have a different format. Are they encapsulated, like IP packets in data-link packets? No,

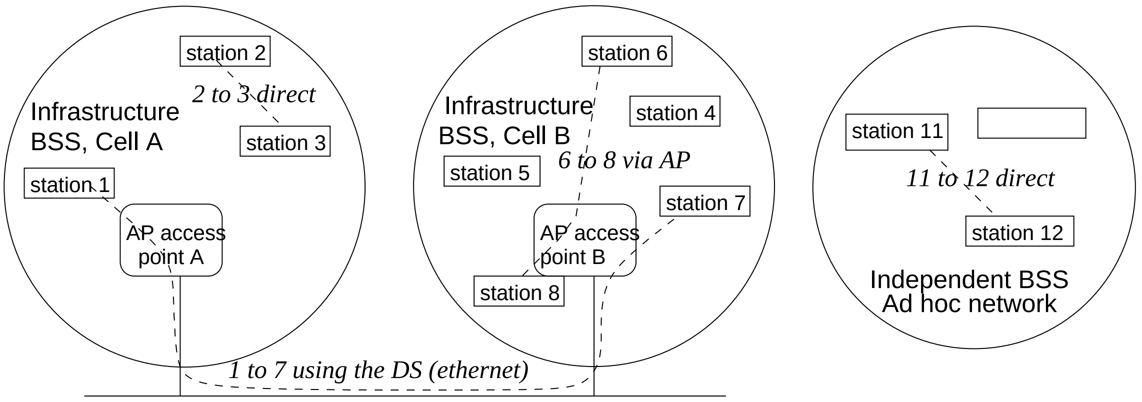


Figure 3.2: 802.11 cell architecture

both 802.11 and 802.3 are data link layers. Does it use some form of routing? No, the AP doesn't look inside for IP addresses.

The AP works in a way similar to an ethernet bridge. The wireless uses the same type of MAC address as ethernet. If the destination MAC address is on the other side of the AP the AP passes it on.

There is only one problem: the format is different. The AP must translate the format of the message from 802.11 to 802.3 and vice versa.

3.3 Services and protocols

In order to cope with the special problems of wireless transmission the 802.11 protocols are quite complicated. They include:

- *association* and *reassociation*, this is to enable stations (such as laptops) to find base stations when they join or leave cells, this supports mobility,
- *authentication* and *encryption*, because wireless nets are so intrinsically insecure this allows passwords and encryption to be used at the MAC level,
- *distribution* and *integration*, this determines how to route frames either via base stations or directly, and also how they frames should be carried over an ordinary ethernet if they must be routed between cells,
- *transmission protocols* (MAC) to send packets, there is a basic set of CSMA/CA rules and two more advanced protocols:
  - DCF *distributed coordination function* to allow packets to be sent directly between any two stations or the base station. These notes will treat the DCF in two stages:
    - basic CSMA/CA protocol to avoid packet collisions (or at least reduce them), and
    - the RTS/CTS exchange which improves collision avoidance. This is required from all 802.11 implementations, but does not have to be used,
  - PCF *point coordination function* this is when the base station takes charge of data transfer for inter-cell or intra-cell transfer. The base station "polls" each station in turn to see if they have any data to transfer and it manages the transfer. It is called the "*contention free*" part of the protocol. It is optional and as far as I can tell (in 2004) it is almost unused, why it is not used I do not know since it can actually prevent collisions.

3.4 802.11 frame formats

The packet (in 802.11 they are called *frames* but I can't help saying packet) format is very complicated, firstly there are alternative formats for different purposes, and even within one format the meaning and use of the fields changes depending on what type of packet it is.
In figure 3.3 the top frame is the most general form of packet, data packets are like this, the lower part of the picture is an expansion of the frame control field. Only notice:

- the frame headers (and FCS) are very long, an overhead of about 34 bytes. There is no preamble (the 8 bytes of "101010...") because, unlike ethernet, it is sent by the hardware and not treated as part of the data link packet,

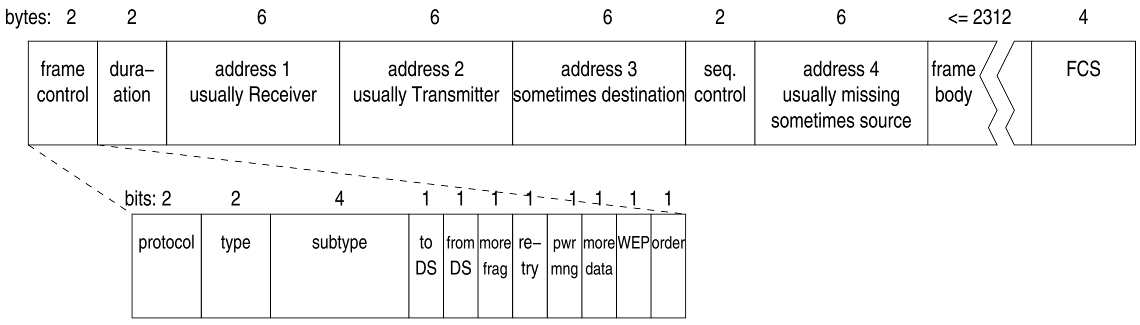

bytes: 2 2 6 6 6 2 6 <= 2312 4
frame control
dura–ation
address 1 usually Receiver
address 2 usually Transmitter
address 3 sometimes destination
seq. control
address 4 usually missing sometimes source
frame body
FCS
bits: 2 2 4 1 1 1 1 1 1 1 1
protocol
type
subtype
to DS
from DS
more frag
re–try
pwr mng
more data
WEP
order

Figure 3.3: A common 802.11 frame (packet) format

- the frame format is given by the type field in the frame control field (see figure 3.3):

  1. management, these are normally used for communication with the AP (access point, the base station): there are frames for new stations to *associate* with the network, and for *authentication*,
  2. control, these are used during data transfers but don't contain data, these include *acknowledgements* and the RTS and CTS messages of DCF (see section 3.7)
  3. data, this is like the top packet in figure 3.3, however there are some variations for combining the control functions of PCF with data.

- the duration field "reserves" the carrier for the length of time of the transfer and sometimes subsequent packets in a transaction, see a later section about NAV,

- why four addresses?

  – For many transfers only two are needed, for example transfers between stations in one cell only need two addresses, the first address is the receiver of the wireless signal and is also the final destination, the second is the wireless transmitter and also the sender.

  – If a station, STA1, sends to the MAC address of a system, SVR1, on the DS (distribution system) it must go via the AP see figure 3.4, address 1 is the wireless receiver MAC of the AP, but it is not the final destination, that MAC address is put in the field address 3, the transmitter address and the sender STA1 MAC are the same in field address 2 like between stations in the same cell. When a station receives from an outside system via the AP the use of addresses is switched: address 1 is the destination and the receiver, address 2, the transmitter is the AP MAC address, and address 3,

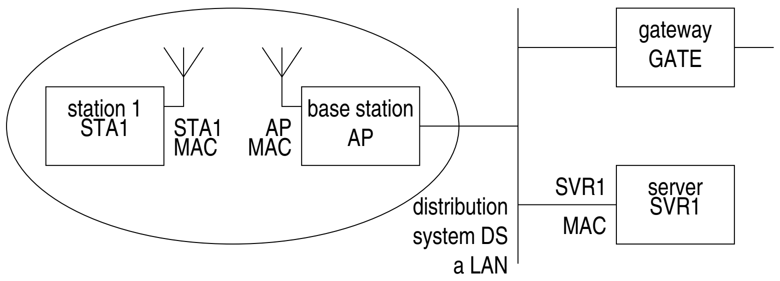


Figure 3.4: Transfers to and from the distribution system

  – four addresses are needed if a wireless network is used as a "bridge" between two LANs, see figure 3.5. The wireless nodes are "transparently" passing on packets from LAN1 to LAN2. It is too long to explain but in the packet sent between STA1 and STA2 the destination and sender addresses in address fields 3 and 4 are the MAC addresses of the systems HO1 and HO2, the MAC addresses in fields 1 and 2 are the MAC addresses of the receiver and transmitter, STA1 and STA2.

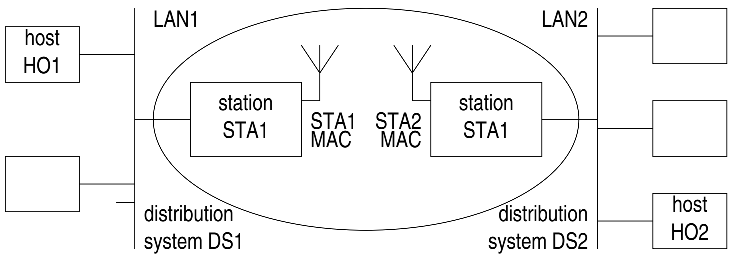


Figure 3.5: Using a wirelesswork net to join two LANs

3.5 CSMA/CA and the problems of wireless MAC

A wired shared medium protocol like ethernet uses CSMA/CD: Carrier Sense Multi-Access with Collision Detection, the wireless protocol uses CSMA/CA: Carrier Sense Multi-Access with Collision Avoidance (it is also known as MACAW, Multi Access with Collision Avoidance, for Wireless). What this means is:

multi-access -> collisions like an ethernet, wireless is a shared transmission medium, lots of stations use the same frequencies (instead of same wire) to send data. Consequently there is the possibility of two or more stations sending at the same time and scrambling the signals, this is a *collision*,

carrier sense use hardware to listen for signals, if there is traffic, wait until it finishes. Only send when the carrier is idle,

collision avoidance don't just detect collisions and then recover like ethernet, instead try to avoid collisions.

The only difference is how they deal with collisions, with ethernet collisions are easy to detect but with wireless detecting collisions is difficult:

- there are weak signals, echoes, and interference so detecting a colliding signal is hard,
- in order to detect a collision it is necessary to be "receiving" at the same time as transmitting, (this is called *full duplex*, send and receive at the same time), this is expensive, very few wireless cards can do it, nearly all are *half-duplex*
- and transmission distance problems, the remote system might get a collision but the sender will not.

Consequently wireless has a protocol that tries to *avoid* collisions.

There are further problems due to wireless transmission, one is the unreliable transmission. With a wired ethernet the chances of a packet becoming corrupted during transmission are very low, with wireless the chances of a packet becoming corrupted are very high. This requires changes to the basic protocol, see the next section 3.6.

3.6 The basic DCF CSMA/CA protocol

The MAC protocol operates at the next level above the hardware, it specifies how data are transmitted, packaged and how the stations respond. Basic rules of sending:

acknowledgements every packet sent and successfully received must be immediately acknowledged. If after a short timeout period the sender doesn't get an acknowledgement message it will retransmit the packet. Every time a packet is re-sent the sender increments a counter, if the counter reaches some limit the 802.11 data link tells the higher layer software (usually IP) that the transmission failed. This is necessary because wireless cannot detect collisions.

sending when the carrier is idle a station is able to send, but it cannot send immediately, it must wait for a short period of time, called the DIFS (to be explained very soon). If two or more stations have been waiting to send then when the carrier has been idle for a DIFS time they will all send at the same time and cause a collision. So they all add an extra random time to reduce the chance of collision.

backoffs when a sender doesn't get an acknowledgement (probably due to a collision so there will be other stations also getting failures) it will retransmit. When the carrier is idle it will wait for a DIFS (not sure, EIFS?) period to which it adds a further random time but the random time will probably be longer—for every retransmission the range of values used for the random time is increased. This increasing range of delays is called the *contention window*. When the packet is acknowledged, or it gives up trying, the contention window is reset to its starting value.

SIFS,PIFS,DIFS & EIFS between any two packet transmissions of any type there must be a short delay called an *inter-frame space* IFS. There are 4 different IFS times: SIFS, PIFS, DIFS and EIFS. The reason for having four times is to permit higher priority transmissions to use the carrier. When a station wants to send a new packet it waits for a DCF IFS (DIFS) time. When a receiver sends an acknowledgement it waits for a *short* IFS (SIFS). This guarantees that the acknowledgement will be sent with no collisions from other packets as the SIFS is shorter than the DIFS. The lengths of the intervals, in increasing time delay, are:

SIPS	short IFS	used for acknowledgments and fragments
PIFS	PCF IFS	used by the base station polling
DIFS	DCF IFS	the “normal” delay
EIFS	extended IFS	used after errors in transmission

The PIFS is between the SIFS and DIFS and is used when the base station is coordinating all stations by polling, it won’t preempt acknowledgements but it will override ordinary transmissions.

In addition to the basic parts of the protocol that allow any stations to send packets there are some extra parts of the protocol to help reduce collisions or to cope with packet loss. These extra rules are required by all wireless networks.

virtual sensing, NAV nearly all packet transmissions “reserve” time by including a *duration* field in the packet, all other stations detecting a transmission set their *network allocation vector*, NAV, to this value. The NAV is basically a timer, once set it counts down to zero. A station will not even try to do carrier sense if its NAV is non-zero, it is a sort of *virtual carrier sense*. Why does this help? Some MAC operations require more than one packet so this stops other stations starting to send in the middle of a transaction, for example a data packet sets a duration time that is the sum of times for the packet transfer *and* the acknowledgement. It is also used for for fragments, see next item and for RTS-CTS, see next section 3.7,

packet fragmentation Because there is a low probability that a long data frame will be sent successfully 802.11 allows long frames to be broken into fragments and sent and acknowleged separately. Each fragment will be sent and acknowledged separately so that only a single damaged fragment needs resending. The sender only pauses for a SIFS interval after the acknowledgement before sending the next fragment (as always the receiver acknowledges after a SIFS), this way the sender keeps the channel. In addition each fragment contains a *duration* covering the time for the following fragment and acknowledgement, so all other stations will set their NAV and not interfere.

3.7 The RTS/CTS part of the DCF protocol

DCF uses RTS/CTS to improve avoidance and solve the *hidden station* problem. The picture 3.6 shows the ranges of station A and station C, which both reach B but not each other. If A wants to send to B and carrier sense shows that the medium is idle then it will send, C also wants to send to B, it detects no traffic and will send to B aswell, unfortunately B gets the scrambled signal from both. This is called the *hidden station* problem.

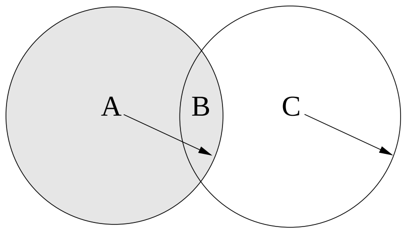


Figure 3.6: Host transmission ranges

This can be avoided in the DCF protocol which uses the following messages:

- RTS (request to send), if station A wants to send to B it waits for no traffic then sends RTS to B. The RTS contains a duration value covering the whole time of the remaining steps of the transaction (SIFS+CTS time+SIFS+data frame time+SIFS+Ack time) so other stations will set their NAVs. It then waits,

- CTS (clear to send), if B accepts the request it sends CTS back to A, it also sends the NAV duration for the remaining time (same as RTS NAV minus time the CTS takes: SIFS+data frame time+SIFS+Ack time),
- when A receives the CTS from B it will send the data to B,
- ACK, when the data arrives successfully at B it will send an acknowledgement ACK back to A. The transfer is complete.
- between each message there is a SIFS delay so no other stations can interrupt.

If any station hears an RTS from another station it will wait for a time long enough to allow the message to finish before attempting to send. If a station hears a CTS from another station it will wait for a suitable length of time. This will *avoid* collisions. If collisions occur when two stations send RTS they will not know, because they don't try to detect it, but the intended receiver(s) will fail to receive the RTS because of the collision so it/they will not send a CTS, consequently the original senders of the RTS will know it failed and they must retry.

Now consider how this deals with the problems of the "hidden station" above, if A sends RTS to B it will not be detected by C, but C will detect the CTS that B sends back to A and will therefore set its NAV and wait until the transfer is over.

Chapter 4

The network layer (IP)

4.1 The Internet

What “internet” means is interconnected networks, but what happens if you join up a few thousand ethernets, point to point links, star networks (like ATM), etc.? Nothing, they all have different packet formats, addresses, protocols and capabilities, so they cannot exchange data. It is necessary to have software on every machine (hosts on networks and on machines that join networks) that can make them work together—this software is IP. It is the network layer protocol IP that *is* the Internet. How it works:

- every network has a unique address, every machine on each network has a unique address. These two addresses are combined together as the *IP address*,
- all machines that will use the network have the *IP protocol* software installed,
- data is sent it a fixed format “packet” known as an *IP datagram*,
- each separate network is joined to one or more other networks by one or more *routers* that know how to reach any network on the Internet,
- when an ordinary host sends a packet to an IP address the IP protocol software consults its local *forwarding table* that tells it whether to send it direct to a machine on the local network, or to send it to a router.

All these topics will be discussed in the rest of this chapter. But first a bit of terminology because the word “network” is used in different ways:

general usage a *network* is any collection of interconnected computers, but this is too imprecise so…

physical a *network* is a just those computers connected by a physical network, ie. all machines on one ethernet, the two machines at either end of a PPP (point to point connection). This is what “network” means when IP software connects two different data-link networks, but…

administrative usage a *network* is the collection of hosts with the same IP network address. This is another way the word is used about the Internet. A network number is allocated to a company or organisation and they have the responsibility of allocating the host numbers to their computers. Such a network will probably consist of many *physical networks*, and they will be called *subnets* in this context.

there are different important usages, there isn’t one meaning, so be aware of the context when you meet the word.

4.2 IP addresses

Every host connected to an internet must have a unique IP address on that network. The address in IPv4 is a 32 bit number. It is usually represented as 4, 8 bit numbers separated by dots, for example: `147.197.205.211` In order to address different networks on an internet the address is structured into a network part and a host part. So the University of Hertfordshire network address is `147.197` and one host on it is `205.211`. Not all networks have a 16 bit address. The NIC allocates network addresses to organisations which in turn are responsible for allocating their own host addresses.

type A If the first bit is 0 (the first 8 bit field is less than 127) then that’s the network address and the host address is 24 bits, there are only just over 100 of these and each can have over 16 million hosts on their nets,

Type B If the first two bits are “10” then the network address is the next 14 bits that means there are about 16000 of these networks, each with upto 65000 hosts,

Type C For smaller organisations if the first 2 bits are “110” than the network address is the following 22 bits and there is only an 8 bit host number, (work it out!).

Type D and E If the first 3 bits are "111" then the remaining bits are used for special broadcast and multicast addressing

This is the original basis of network address allocation but now (2004) type A address ranges are split to make more network numbers available. This means finding the network part of the address is not quite so simple, the new way is used by CIDR (classless internet domain routing), which you follow up if you wish.

4.3 IP packets

The IP layer on one machine must send packets to the IP layers on other machines, to do this it uses the IPv4 (and eventually IPv6) protocol. The format of an IPv4 message is shown in figure 4.1. The important

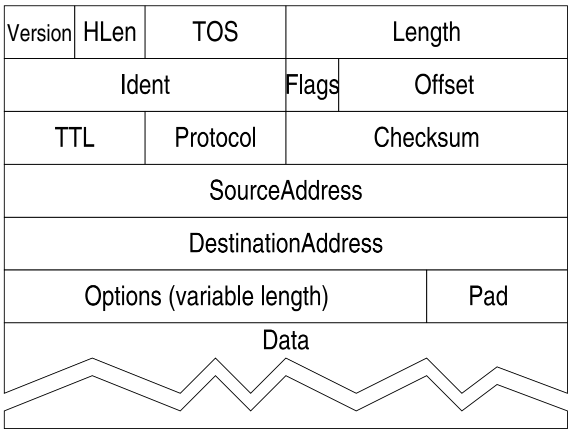


Figure 4.1: IP packet format

fields shown in figure 4.1 are:

HLen The length of the header, can vary because of options,

Length The length of the whole packet,

Flags One job they have is to indicate if this packet was broken up into fragments because it was larger than the maximum size allowed for some physical network, if so the offset field is used to indicate which fragment.

Protocol Can be TCP or UDP so IP knows which higher layer to pass it to.

TTL "Time To Live", it is hop-count, every IP layer in each router it passes through decrements it by 1, when the count reaches 0 the packet is discarded.

Checksum Computed across the header.

4.4 Forwarding tables

Not all machines are directly connected to all others, so how does a machine that is only indirectly connected to another know which intermediate machine to send to first? They look up the address of the destination network in a *forwarding table*, which tells them where to send the packet on the first step of its journey. In a bit more detail:

- all forwarding is to *networks*, once the packet gets to the right network it can be directly delivered,
- every host has a *forwarding table* (sometimes called a routing table) that lists how to get to other networks on the Internet,
- a forwarding table specifies for every network what the *next-hop* is,
- for ordinary hosts on a *stub* network (that's us) the forwarding table will have: its own network and then any other networks that are linked by routers on its local net, then there will be a *default* route where all other packets are sent, this is usually the organisations internet gateway,
- every machine that is connected to more than one network is a *router*, on the main backbone of the internet routers have gigantic forwarding tables that include the next-hop for *every* network attached to the internet, they don't have default routes.

This is the *vital* function of IP, getting packets across one physical network to another thereby creating an *internet*.

4.5 Example of using forwarding tables

This is a simplified example where the "internet" is just two networks connected via a router. The picture 4.2 illustrates packet forwarding, where 131.9.0.8 (aka. 62.0.0.1) is the router attached to both net 62.0.0.0 and 131.9.0.0. Note that on an internet a system has one IP address for *each* network it is connected to. All systems have a *forwarding table* with all the networks it can reach. In this example there are only two networks, 131.9.0.0 and 62.0.0.0 so each table has two entries. The format of forwarding table columns:

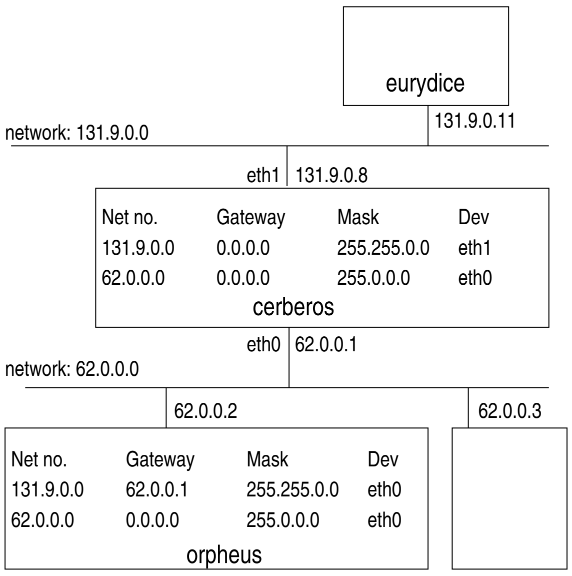


Figure 4.2: Packet forwarding example

- The first field of a forwarding table is the destination network. Every IP address has two parts, network and host. Note that 131.9 is a type B address and 62 is type A,
- The second table field contains the *gateway* to use, this is the *next hop*, usually a router, if the system is directly attached to the network the gateway is 0.0.0.0. In cerberos which is directly connected to both networks both gateway fields are 0.0.0.0. In orpheus the gateway for network 131.9.0.0 has the address of cerberos,
- The third entry is a *network mask*. It is used by the forwarding software to find which entry to use. The destination address of every incoming packet is *and-ed* with each mask in turn and the result compared with the first column network number to get a match. Because of the use of subnets in networks and the splitting of type A addresses it is not possible to use the type A, B, or C bits to determine the network part, so every network destination has its own mask. In this example it is easy, 131.9.0.0 is a type B address and the mask is 255.255.0.0 (first 16 bits all binary "1", last 16 bits all "0") which means any any address such as 131.9.0.11 and-ed with the mask will leave just the top 16 bits, 131.9.0.0, for comparison.
- The last field in the forwarding table is the NIC (network interface card) address, in other words it tells the IP software which datalink to use.

Assume that orpheus, 62.0.0.2, wishes to send to eurydice, 131.9.0.11, then:

- the transport layer passes an IP datagram to the IP software on 62.0.0.2,
- the destination address 131.9.0.11 is compared with each line of the forwarding table in turn (top down, order matters). Each time the mask is applied, so:

  $$131.9.0.11 \wedge 255.255.0.0 = 131.9.0.0$$

  this matches on line one, so the packet is sent to 62.0.0.1 via device eth0. NB this doesn't change the destination address, it is still 131.9.0.11, just where it is sent.

- When the packet arrives at 62.0.0.1 the same procedure is applied, it masks the address 131.9.0.11 and matches on the first line of the table which says there is no gateway, just send it on datalink eth1, and it arrives at the destination.

4.6 Sending on an ethernet: ARP

If the forwarding (routing) lookup finds the IP address of the next hop is on the same LAN, eg. ethernet, then it is necessary to find its ethernet address. This is not done by the data-link layer it is the job of software in the IP layer (though not the IP protocol itself).

Ethernet MAC addresses are 48 bit numbers built into the hardware of the controllers, they have no relationship to the IP addresses being used by the network level.

One solution would be for every machine to have a fixed table mapping IP addresses to Ethernet ones for its network. However every time systems were added or removed from the net all tables would need updating.

Instead the sending system uses a special protocol called ARP (Address Resolution Protocol) which sends an ethernet broadcast message to the whole LAN saying:

Who is 147.197.236.236?

All systems on the ethernet must check all ARP packets for their number, if it is their's they will respond with their Ethernet address, saying:

I am 147.197.236.236, my MAC is: 00:01:02:AE:95:BE

This information is used and then câched in an *ARP table* by the sender so it won't need to ask again for sometime.

4.7 Building forwarding tables: routing

There must be a way of constructing the forwarding tables. The simplest method that is suitable for many systems on local ethernets with one link to the internet is to manually add (or use the DHCP protocol—look it up!) a *default* route.

```
Kernel IP routing table
Destination   Gateway       Genmask        Metr Iface
147.197.232.0 *             255.255.248.0  0    eth0
default       147.197.232.1 0.0.0.0        0    eth0
```

Which means any address that matches 147.197.232.0 (ie. anything on a local ethernet) is sent directly. But anything else default is send to 147.197.232.1.

If there are lots of separate ethernets or other LANs joined together as subnets of a larger network then creating the tables manually won't work, instead each system must run a *routing* program that can talk to other routing programs and together they can build their forwarding tables. For small autonomous systems there are two protocols often used: RIP, old and weak but simple, and OSPF which is much better but more complicated. In the case of main backbone internet routers completely different routing programs are needed, they must have enormous tables so they know for every network which next router to send to. The current method is called BGP4 (Border Gateway Protocol 4).

4.7.1 A routing simplification

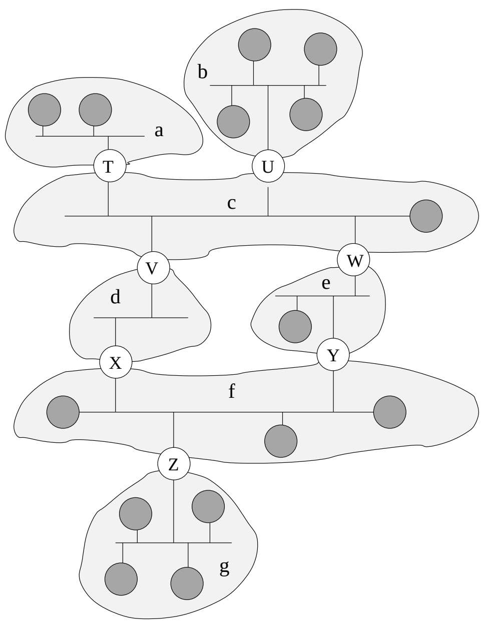
Internet routing is between separate networks or subnets and is done by *routers* to *networks* not hosts. The following sections present the principles of routing algorithms and it is easier to treat routing as occurring between host computers. However the principles of routing algorithms are applicable to real networks situations.

Figure 4.3 shows a collection of networks joined by routers. Network d is connected to network c by router V, but this is simplified in the graph on the right and is show as a connection (link, edge, arc ...) between d and c. In other words the networks have become nodes and the routers are links. But in the following notes these nodes will often be called "computers" or "hosts" not "networks", however the routing issues are still the same.

4.7.2 Note about "distances"

Most routing decisions depend on the "cost" of using a link between any pair of systems, so that they can work out the best route. The costs that can be used vary:

- Money cost of using a link
- Speed of the link, so the fastest links are preferred,



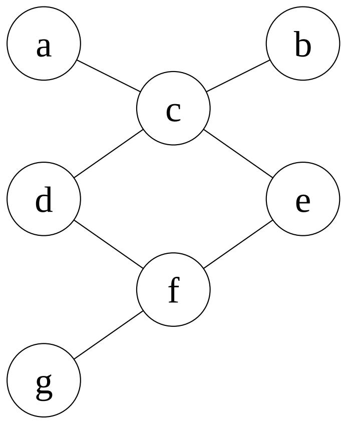


Figure 4.3: Two representations of connections between networks

- Delay, even though some links are fast they might be overloaded as the “cost” to be minimized is delay time,
- The number of links that must be crossed to reach the destination, where the cost of every link is 1, this is called *hop count* and is the commonest.

4.8 Shortest route or link-state routing

A network can be represented by an undirected graph, where each *node* represents a host and each *edge* is a network connection, we are using the simplification described in section 4.7.1.

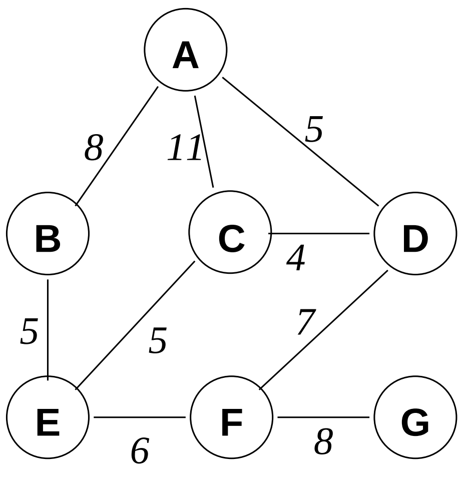


Figure 4.4: Example network

In figure4.4 node A is connected to node B with a “cost” of 8, (note: cost might be financial, time-delay or physical distance), this is written as cost(A,B)=8.

If a host has all of the above information it can compute the best next-hop for every node in the network using Dijkstra’s shortest route algorithm developed in the 1960s for any graph, not just computer networks.

The algorithm keeps a set *S* of “open” or “unexplored” nodes, an array *Dist* of distances from the start to each node, and an array *Rt* of the next-hop to all nodes. The arrays are indexed by the node names or numbers. On each cycle of the algorithm the closest “unexplored” node is chosen, it is called *u*, then each of the open nodes *v* adjacent to *u* are examined to see if there is a shorter route to them via *u*. After the closest node *u* has been examined it is “closed”, ie. removed from the set *S*.

```
Initialize set S to contain all nodes except source;
Initialize array Dist so Dist[v] is the "cost" of the edge
  from source to v, set to infinity if no edge to v;
Initialize array Rt so Rt[v] is set to v if there is
  an edge from source to v, and set to 0 otherwise;

while(! S.empty() ) {
  select a node u from S so that Dist[u] is minimum;
  if( Dist[u]==infinity ) {
    fail: no path to all nodes in S; exit;
  }
  S.remove(u); // remove u from S
  foreach node v such that there is an edge (u,v) {
    if( S.member(v) ) {
      cost = Dist[u] + cost(u,v);
      if(cost < Dist[v]) {
        Rt[v] = Rt[u];    Dist[v] = cost;
      }
    }
  }
}   // done forwarding table is Rt
```

Now if the algorithm is applied for a couple of iterations:

1. initialise S, $Dist$ and Rt
   giving: $S = \{B,C,D,E,F,G\}$

	A	B	C	D	E	F	G
$Dist$:	0	8	11	5	∞	∞	∞
Rt :	A	B	C	D	-	-	-

2. Choose $u = D$, remove D from S,
   consider $v = C$: $cost = Dist[D] + cost(D,C) = 5 + 4 = 9 < 11$,
   so: $Rt[C] = Rt[D]$, $Dist[C] = 9$
   consider $v = F$: $cost = Dist[D] + cost(D,F) = 5 + 7 = 12 < \infty$,
   so: $Rt[F] = Rt[D]$, $Dist[F] = 12$
   giving: $S = \{B,C,E,F,G\}$

	A	B	C	D	E	F	G
$Dist$=	0	8	9	5	∞	12	∞
Rt=	A	B	D	D	-	D	-

3. Choose $u = B$, remove B from S,
   consider $v = E$: $cost = Dist[B] + cost(B,E) = 8 + 5 = 13 < \infty$,
   so: $Rt[E] = Rt[B]$, $Dist[E] = 13$
   giving: $S = \{C,E,F,G\}$

	A	B	C	D	E	F	G
$Dist$=	0	8	9	5	13	12	∞
Rt=	A	B	D	D	B	D	-

4. Choose $u = C$, remove C from S,
   consider $v = D$: ignore, not in S
   consider $v = E$: $cost = 9 + 5 = 14 \not< 13$,
   so: no change
   giving: $S = \{E,F,G\}$

	A	B	C	D	E	F	G
$Dist$=	0	8	9	5	13	12	∞
Rt=	A	B	D	D	B	D	-

5. Choose $u = F$, remove F from S,
   consider $v = D$: ignore, not in S
   consider $v = E$: $cost = 12 + 6 = 18 \not< 13$, so: no change
   consider $v = G$: $cost = D[F] + cost(F,G) = 12 + 8 = 20 < \infty$,
   so: $Rt[G] = Rt[F]$, $Dist[E] = 20$
   giving: $S = \{E,G\}$

	A	B	C	D	E	F	G
$Dist$=	0	8	9	5	13	12	20
Rt=	A	B	D	D	B	D	D

   (NOTE: $Rt[G] = Rt[F] = D$, since we want the "next hop", although we found the route to G from F we use *the route to* F not F itself.)

6. Choose $u = E$, no changes ...
7. Choose $u = G$, no changes ...

The algorithm continues until there are no nodes left in S with a value less than ∞.

4.8.1 Using shortest route algorithm for routing

The shortest path algorithm is *not*, by itself, a routing algorithm or protocol.

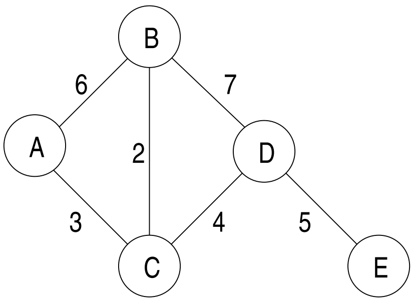
The main problem is that the information about all the link costs that each node uses to find the shortest paths is unknown, each node only knows about its own immediate connections. To be useful as a routing method there must be a way to collect all link costs. One way to do this is to have a protocol where every node sends packets about its links to all its neighbours, they in turn pass these packets on unchanged. All systems learn about all the links. In order to stop the packets circulating for ever each has a counter (a TTL, time-to-live) that is decremented each time it is passed on, when it is zero the packet is dropped. This is called *reliable flooding*.

There is a practical routing technique called OSPF (open, shortest path first) that uses the shortest route algorithm, and includes a protocol to periodically collect information about network changes using reliable flooding. It can be used on quite complicated networks (in the administrative sense) consisting of many subnets (networks in the physical sense). Because it is designed for large networks OSPF supports hierarchical structures of networks. Even OSPF is not suitable for the backbone of the internet, it cannot route between administrative networks, only within them.

4.9 Distance vector routing

The following is a simplified description of a routing algorithm. It is called a *distance vector* method. RIP uses a method a bit like this (but note this is not RIP which has additional features). The whole algorithm doesn't require a global picture, all participating routers only know about their direct connections to their neighbours and no others. Finding the shortest route is a *distributed* task, all routers exchange information and incrementally improve their forwarding tables until they are stable.

In this treatment it is assumed that hosts are connected to hosts as described in section 4.7.1. The format of the *forwarding table* used here is:



(a) The network

dest	cost	goto
A	0	A
B	6	B
C	3	C
D	∞	
E	∞	

(b) Initial table at A

dest	cost	goto
A	0	A
B	5	C
C	3	C
D	7	C
E	12	C

(c) Final state at A

Figure 4.5: Simple example network

In figure 4.5 a simple network is shown with versions of the forwarding table from one node, A. The first table shows an initial state based only on knowledge of the hardware connections. The second table represents the optimal routes, the ones we hope will result from a successful routing algorithm, from A to all other nodes in the net. Remember that the forwarding table only shows the first node on the best path, the *next hop*. Initially The entry for B says the route is cost 6 and go straight to B. If the route is unknown it is infinity ∞. However after the routing algorithm the entry for B says the route is cost 5 and go to C first. Notice further that the final state has the routes to all other nodes and the costs. In the forwarding table the cost from a node to itself, at A to get to , is 0.

4.9.1 An example network

Figure 4.6 shows a simple network, it will be the example to explain distance vector routing.

Notes about the bits in each host that will be used for routing:

- Each node contains at the bottom centre its forwarding table. In figure 4.6 infinity, ∞ is shown as . Since this first map shows an initial state only directly connected systems are known.
- There is also a small connection list giving the hardware links a node has to other nodes and the "cost" of the link.

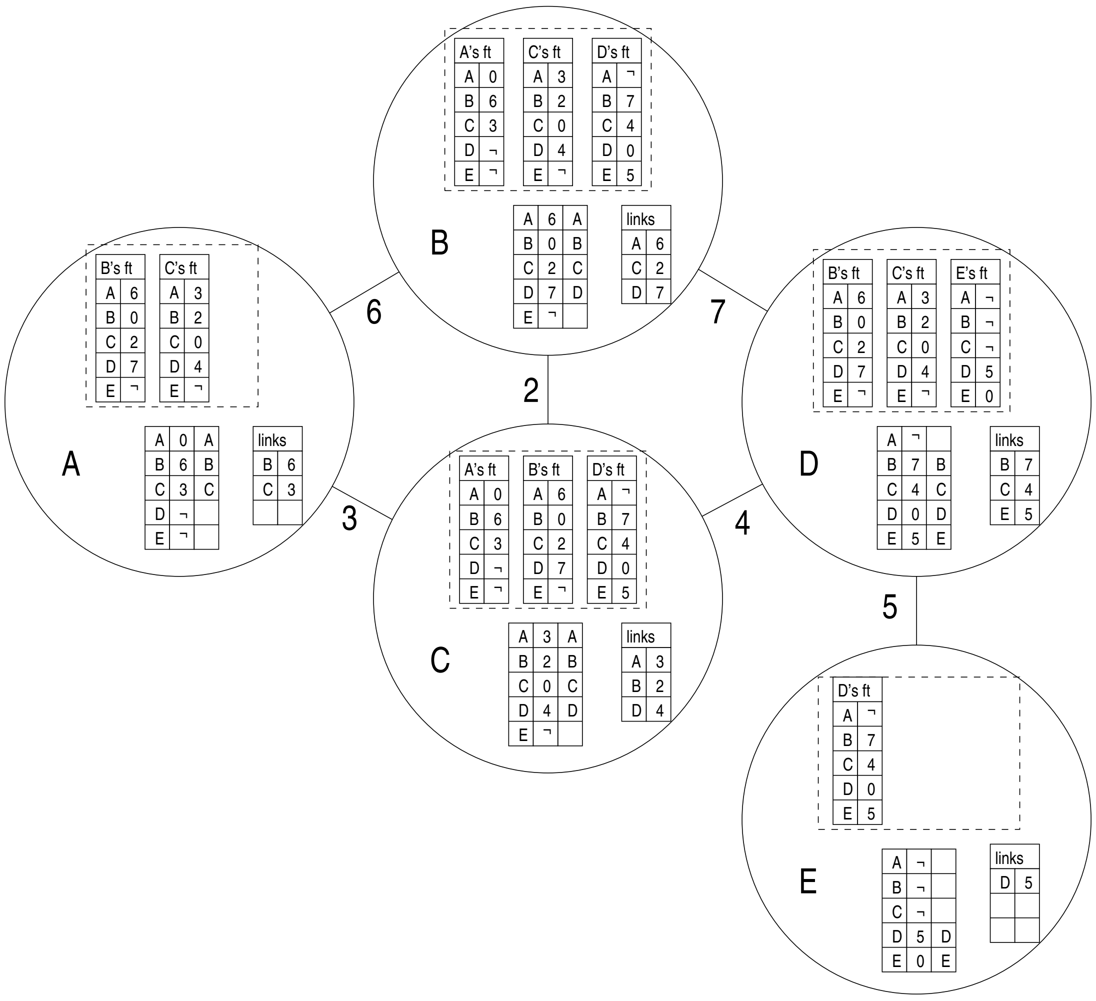


Figure 4.6: An example network for routing

- At the “top” of each host is a list of the forwarding tables sent to a node by each of its immediate neighbours. So that as A only has links to B and C (2 entries in the connections table) it has copies of their tables, but E only has one neighbour, D, so it has only received one forwarding table. Notice that initially each system has its neighbours tables but it hasn’t yet used them to update its own table, see the next section 4.9.2.

4.9.2 The algorithm

The basis of the algorithm is:

> if your immediately connected neighbours have routes and distances to a place X and you add your link cost to each of those distances and select the smallest then your best route will be via the neighbour whose distance plus the link cost was the least.

An algorithm based on this idea is called a Bellman-Ford algorithm after two of the inventors. But how do they get the shortest routes? Answer: all the nodes in a network do this minimising, basing their forwarding table on tables from their neighbours, and in turn sending their forwarding table. This is called *distributed* Bellman-Ford or *distance vector*. Proving that the distributed version is correct is hard but has been done.

The steps of the algorithm. Every node *x* will:

1. initially set its forwarding table distance to the link cost of the direct connections: to the link cost from the node *x* to each neighbour node *v*, All other entries are set to ∞, infinity.

2. at fixed intervals repeat:

   (a) send a copy of its forwarding table to all its neighbours,

   (b) receive copies of the forwarding tables from all neighbours,

   (c) for each destination on the net y find each neighbour's cost to y (from the copy of their table) and add the cost of the link to the neighbour,
   (d) select the minimum of all these sums and set x's forwarding table entry for y to the minimum distance and set the next hop to the neighbour whose table gave the smallest sum.

Another way of expressing the algorithm. Where $d_x(y)$ is the forwarding table distance at node x to node y; $c(x,v)$ is cost of the direct link from x to neighbour v. Every node x will:

1. initially set $d_x(v)$ to $c(x,v)$, All other entries in d not in c are set to ∞.
2. at fixed intervals repeat:
   (a) send a copy of d_x to all direct neighbours, v,
   (b) receive tables d_v from all neighbours v,
   (c) for each y select the minimum of $c(x,v) + d_v(y)$ for all neighbours v. Set $d_x(y)$ to the minimum:
   $$d_x(y) = min_v(c(x,v) + d_v(y))$$
   Set the next hop to v.

4.9.3 Using the algorithm with example net

1. Consider the network in figure 4.6, look at node A, it has initialised its forwarding table to the links to neighbours. Further it has just started the first cycle and received copies of the tables from B and C (with the "next hops" removed since they are not used).
2. it will consider each host in turn:
   (a) A, don't bother we can't get a shorter route, this is us,
   (b) B, consider tables:
   B's DV to B is 0 + link(B,6) = 6,
   C's DV to B is 2 + link(C,3) = 5,
   C is the minimum so set table distance to B to 5 and the next hop to C,
   (c) C, find the minimum, via B it is 2 + 6, via C it is 0 + 3, it doesn't change,
   (d) D, consider tables:
   B's DV to D is 7 + link(B,6) = 13,
   C's DV to D is 4 + link(C,3) = 7,
   C is the minimum so set table distance to D to 7 and the next hop to C,
   (e) E, both table copies from B and C for D are infinity.

   This produces the new table at A:

A	0	A
B	5	C
C	3	C
D	7	C
E	∞	

   this is the end of cycle one on A.

3. cycle two starts, however realise that A won't get the same forwarding tables a second time from B and C because they too have, in parallel, updated their forwarding tables. The forwarding tables at B and C now are:

On host B

A	5	C
B	0	B
C	2	C
D	6	C
E	12	D

On host C

A	3	A
B	2	B
C	0	C
D	4	D
E	9	D

4. A now sends its new table and receives the new tables from B and C.

5. it will consider each host in turn:
   (a) A, don't bother we can't get a shorter route, this is us,
   (b) B, the copied tables are the same as last time for B so the result will be the same, distance 5 next hop C,
   (c) C, find the minimum, via B it is 2 + 6, via C it is 0 + 3, it doesn't change, same as last time,
   (d) D, NB. B's table has changed:
   B's DV to D is 6 + link(B,6) = 12,
   C's DV to D is 4 + link(C,3) = 7,
   C is the minimum so set table distance to D to 7 and the next hop to C, but the outcome is the same,
   (e) E, consider tables:
   B's DV to E is 12 + link(B,6) = 18,
   C's DV to B is 9 + link(C,3) = 12,
   C is the minimum so set table distance to B to 12 and the next hop to C,

   This produces the new table at A:

A	0	A
B	5	C
C	3	C
D	7	C
E	12	C

   this is the end of cycle two on A.

6. this can continue but unless there are changes in the network the tables won't change.

4.9.4 Another way to visualise the algorithm

Instead of dealing with one node step by step it is possible to picture the tables of all nodes at once. In some ways this is more appropriate since all the updates take place concurrently. For the network of picture 4.6 the initial state can be shown as:

On host A			On host B			On host C			On host D			On host E		
A	0	A	A	6	A	A	3	A	A	∞		A	∞	
B	6	B	B	0	B	B	2	B	B	7	B	B	∞	
C	3	C	C	2	C	C	0	C	C	4	C	C	∞	
D	∞		D	7	D	D	4	D	D	0	D	D	5	D
E	∞		E	∞		E	∞		E	5	D	E	0	E

then after all the systems send their tables, and do their updates once the new state of all the systems is:

On host A			On host B			On host C			On host D			On host E		
A	0	A	A	5	C	A	3	A	A	7	C	A	∞	
B	5	C	B	0	B	B	2	B	B	6	C	B	12	D
C	3	C	C	2	C	C	0	C	C	4	C	C	9	D
D	7	C	D	6	C	D	4	D	D	0	D	D	5	D
E	∞		E	12	D	E	9	D	E	5	D	E	0	E

after one cycle quite a lot of information has propogated but A and E still don't know about each other.

On host A			On host B			On host C			On host D			On host E		
A	0	A	A	5	C	A	3	A	A	7	C	A	12	D
B	5	C	B	0	B	B	2	B	B	6	C	B	12	D
C	3	C	C	2	C	C	0	C	C	4	C	C	9	D
D	7	C	D	6	C	D	4	D	D	0	D	D	5	D
E	12	C	E	11	C	E	9	D	E	5	D	E	0	E

the tables have reached a stable state. The normal rule for operation as part of a real routing protocol would be to periodically send the table to neighbours, or whenever a change occurs.

4.9.5 Broken links

Nodes monitor their direct connections and if a node goes down they reset their connections table. This also means they won't receive a copy of the forwarding table from the node at the end of the broken link. This means that when they calculate their new forwarding table it will not use the broken route.

Sometimes other nodes can pass back incorrect routes to the one that lost a link. Consider that if D's link to E is broken, on the next cycle C will send its table to D saying that it can get to E with a cost of 9. The simplicity of the algorithm doesn't let D know that the route to E learnt from C actually goes through D. This leads to instabilities that take some time to settle down. It is sometimes called the *count to infinity problem*.

In order to reduce instability a technique called *split horizon* is used where a node doesn't tell another about a route that involves it, ie. when copies of a forwarding table are passed on by node x to neighbour v, remove all entries where the next hop is v. So, for example, C will never pass D routes where D is the next hop. This can help prevent the algorithm becoming unstable after breaks. There is another version where a route is sent to the neighbour that is the next-hop but it contains ∞so it will never be used, this is called *split horizon with poison reverse*.

Chapter 5

More about the network layer

These are a few additional notes about the network layer, IP. There is a loose structure: subnets, subnet masks, CIDR, and routing on the backbone of the internet.

5.1 Subnets and subnet routing

Many Internet networks, in particular type A and type B, can be quite large with many hosts, they must be separated into *sub-nets*, because it is not workable to have thousands of hosts on one physical LAN. In many ways one administrative internet network (an *autonomous system*) with subnets is itself an *internet*, there must be subnet routers.

5.1.1 Subnet addresses

The first problem is to divide the host address space, this must (like type A, B and C nets) be a power of two. Consider figure 5.1. So if the herts.ac.uk net address is `147.197.0.0`, 16 bits give the network address and 16 bits the host, the host is further divided into a 5 bit subnet number (giving upto 32 subnets) and an 11 bit host address (giving upto 2048 hosts on each subnet).

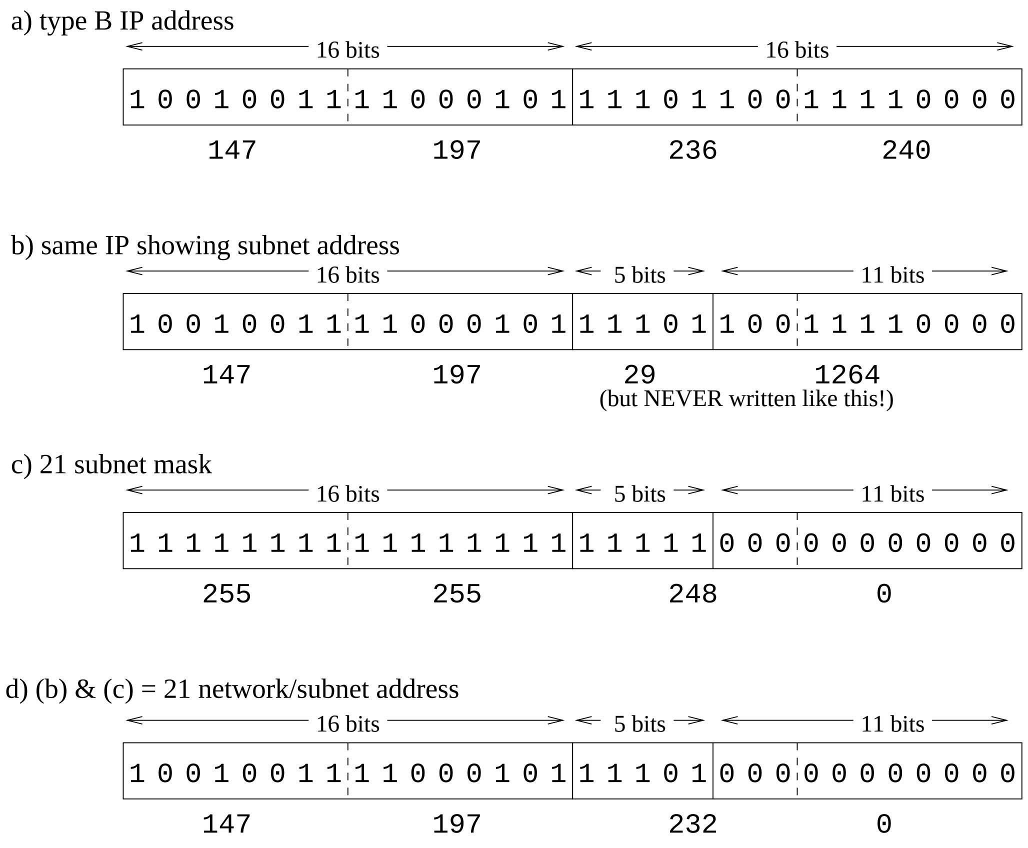


Figure 5.1: Subnet addresses

An example herts.ac.uk address (B type) is given in part (a) of the picture, `147.197.236.240`. The subnet part is shown in part (b), note that this is just a simple 32 bit number, it is only by convention that it is written as four 8 bit numbers in decimal, therefore we could say this is subnet 29 (the 5 bits), host 1264 (given by the 11 bits), but that would be confusing so it is still written conventionally.

5.1.2 Packet forwarding with subnets

The rest of the Internet doesn't know or care about the subnets on individual networks, routing from outside is still to the whole network but all the systems on the network must be aware of the subnets—they must forward to the correct subnet.

The way that packet forwarding occurs is to compare the *network* part of the address with entries in the forwarding table to select the destination, but what is the network part?

It is only possible to send *directly* to a system on a LAN if it is on the same *subnet*, so it is necessary to examine the net and subnet number, at Hatfield the network and subnet part is 21 bits long, but how does the IP routing software know? It must be provided with a *mask* that when and-ed with the address leaves only the net+subnet part which can be compared with the network numbers. For the Hatfield subnet the subnet mask is 21 bits long, when written in conventional IP notation is is `255.255.248.0`, sub-picture (c) shows the binary value of the mask. The result of and-ing the mask with the example address `147.197.236.240` is shown in binary in (d), in conventional IP notation it is `147.197.232.0`.

It is also possible to examine the forwarding table on host `149.197.236.240`, it shows the local subnet number and the subnet mask applied to destination addresses.

```
Destination     Gateway         Genmask        Flags Metric Use Iface
147.197.232.0   0.0.0.0         255.255.248.0   U     0      0 eth1
0.0.0.0         147.197.232.1   0.0.0.0         UG    0      0 eth1
```

If this machine `149.197.236.240` sends to `149.197.239.69` then the forwarding table will mask the destination address with the subnet mask `255.255.248.0` giving `147.197.232.0` which will be sent out directly (no gateway). If, however, the destination is `147.197.200.44` the mask will produce `147.197.200.0`, this won't match the first network destination so the last line will be used instead and the packet will be forwarded to the gateway `147.197.232.1`. Note that this treats the problem of routing to other subnets and to other networks in the same way, in both cases the packets go to the gateway and it must decide to forward to another subnet or go out to the Internet.

5.1.3 Another notation for subnet addresses

Note that forwarding with subnets blurs the distinction between the *network* and *host* parts of an address. If subnets are used it is not enough to recognise a type A, B or C address and know what the network address is. Consequently there is a different way to write network addresses that makes absolutely clear what the network (maybe with subnet) part is:

full-network-address/number-of-bits-of-network-part

for example the address of the subnet my machine uses is: `147.197.232.0/21` which gives the length of the network+subnet part. It gives two things: the subnet mask (length 21), ie. `255.255.248.0`, and it gives the value of the 21 bits—the network number.

5.2 The backbone of the Internet

There is an important concept on the backbone of the internet: that of an *autonomous system* (abbreviated as "AS"), which is a network or group of networks administered collectively. Autonomous systems are of two main types:

stub this is an autonomous system, usually of only one network, with only one router connection to the rest of the internet, a network like Hatfield, or an ISP that just supports direct customer lines,

transit this is an autonomous system, usually made of many networks, that has many connections to other ASs and its primary job is to carry through traffic (usually for profit),

there are some autonomous systems that are hybrid, called *multi-homed*, they have more than one connection to the rest of the internet but they don't permit through traffic.

Figure 5.2 shows an internet with stub and transit ASs. It tries to show that the transit ASs on the backbone of the internet have complicated internal structure consisting of many networks each with its own internal structure. In addition the geography is not localised, some long haul telecomms companies have autonomous systems that span continents. Also notice that each autonomous system has a unique 16 bit network number, an ASN, only transit AS need numbers, they are not assigned to stub ASs. Another thing to note about the picture is that within backbone networks only some routers are connected to neighbouring networks, others are purely internal.

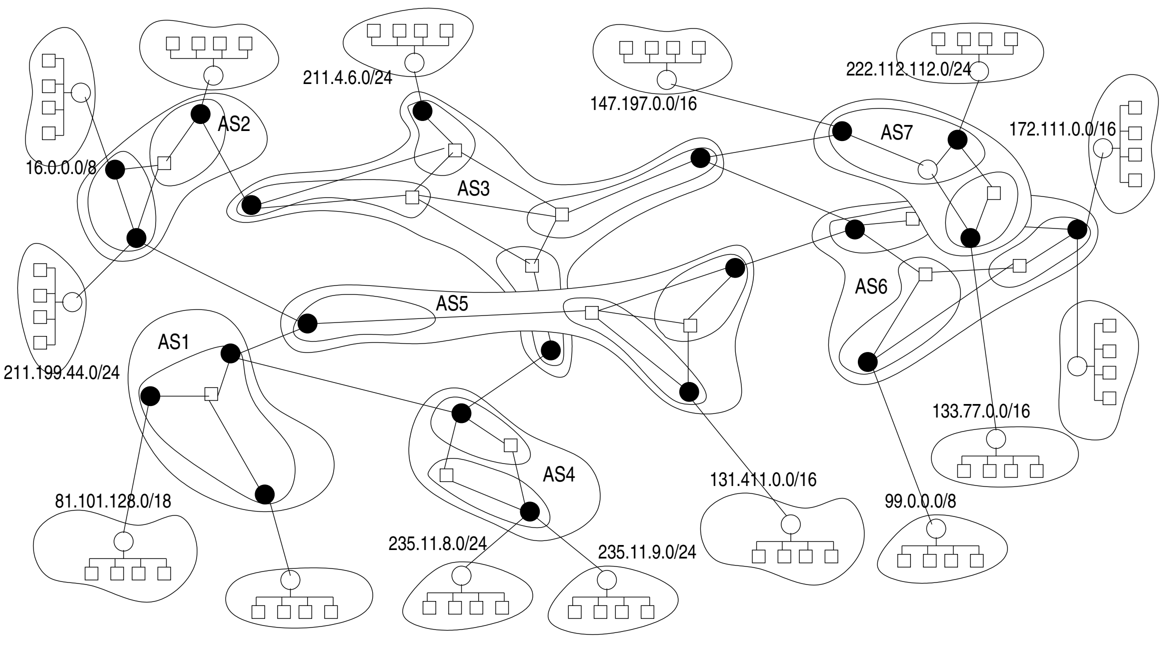


Figure 5.2: The structure of Internet

5.2.1 Routing on the backbone of the Internet

From the point of view of routing all the stub networks are just destinations they do not participate in the routing, only the transit ASs do internet routing. The job of routing on the backbone of the internet is two-level: firstly there are routes *between* ASs, this routing is called *exterior routing*, and then there is the problem of routing *within* each AS, this is called *interior routing*. There need to be two levels of routing protocol:

- to manage the complexity, any router that handles inter-AS routing needs a forwarding table of all the possible network destinations, currently (2004) about 150,000, to make thousands of routers handle this and exchange the information would be impossible, so restricting it to a few makes it more manageable,

- because each AS is managed by a different organisation and therefore runs its own internal networks differently, the routing algorithms within adjacent ASs might be incompatible consequently the separation is necessary, and

- because interior protocols within one organisation just find the best route but exterior routing needs protocols that can implement policies, for example: “don’t use AS9999 because it hasn’t paid us for six months”, or “don’t send US government traffic through an AS in Iran”.

The interior routing protocols can be whatever the operator of the autonomous system wants, but OPSF is the most widely used, it is powerful enough to cope with routing between and within the separate networks that might make up one autonomous system. The current (2004) exterior routing protocol used on the internet is called BGP-4 (the Border Gateway Protocol). It is sometimes called a *path vector* protocol it has some similarities with *distance vector* like exchanging table changes with its neighbours *but* it exchanges the full paths to destinations not just the next-hops.

5.2.2 How BGP-4 works

Each BGP-4 router has a forwarding table with an entry for every distinct network address on the internet, each entry has a path of AS numbers between itself and the destination. The reason for the path is so that policy decisions can be made by the administrator of the AS, chosing a route through certain systems and avoiding others. For example, here is an edited textual representation of part of a BGP table:

```
PREFIX: 147.197.0.0/16
FROM: 129.250.0.232 AS2914
ASPATH: 2914 3356 786
NEXT_HOP: 129.250.0.232
...
```

```
PREFIX: 147.197.0.0/16
FROM: 168.209.255.2 AS3741
ASPATH: 3741 702 786
NEXT_HOP: 168.209.255.2
...
PREFIX: 147.198.0.0/16
FROM: 64.211.147.146 AS3549
ASPATH: 3549 209 568 721 1505
NEXT_HOP: 64.211.147.146
...
```

There are no metrics or costs (or in other words the metric is always 1), this is because they are meaningless, each “hop” means crossing a whole AS which could be using any interior routing protocol that attached totally different meaning to its metrics from any other AS.

Each AS has at least one *BGP speaker* that exchanges information with BGP speakers in other autonomous systems; there may be many more BGP routers in an AS but not all will exchange information with neighbouring ASs. Each BGP speaker establishes semi-permanent TCP connections to its neighbour AS BGP speakers to exchange information. If changes occur to a table it will pass the changes to its neighbours, they will update their tables to find alternative routes that satisfy their policies. Note that a whole AS becomes just one point in an AS route, so from a routing point of view figure 5.3 is equivalent to the previous picture: As an example of route propogation:

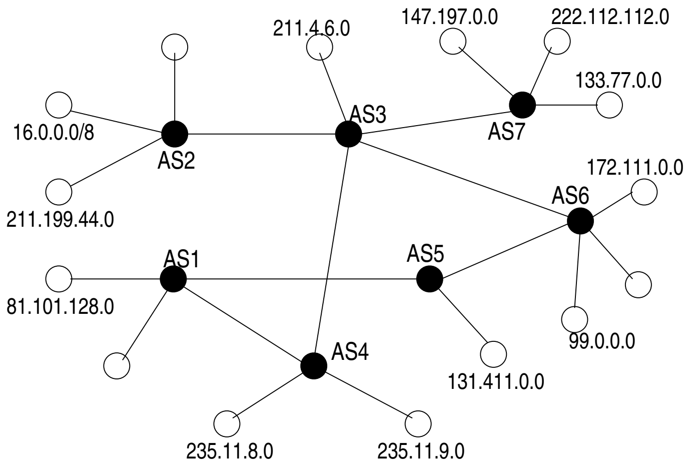


Figure 5.3: Routing through Autonomous Systems

- AS6 will tell its neighbours, AS3 and AS5 that it has a network `172.111.0.0`,
- AS3 will tell AS4, AS7, AS2 that it has a path:

  ```
  172.111.0.0: AS3, AS6
  ```

- AS4 will in turn tell AS1 that it has the path:

  ```
  172.111.0.0: AS4, AS3, AS6
  ```

- Also, AS5 will tell AS1 it has a route:

  ```
  172.111.0.0: AS5, AS6
  ```

- now AS1 has a choice of routes: [AS5, AS6] or [AS4, AS3, AS6] and it will choose one depending on its site’s policy.

Also notice that the sending of full paths makes the protocol quite stable, if an AS receives a route that contains its own number it will discard the route.

The figure 5.4 shows the paths from AS786 (Janet) to other transit ASs, notice the average AS path length is only 3 or 4.

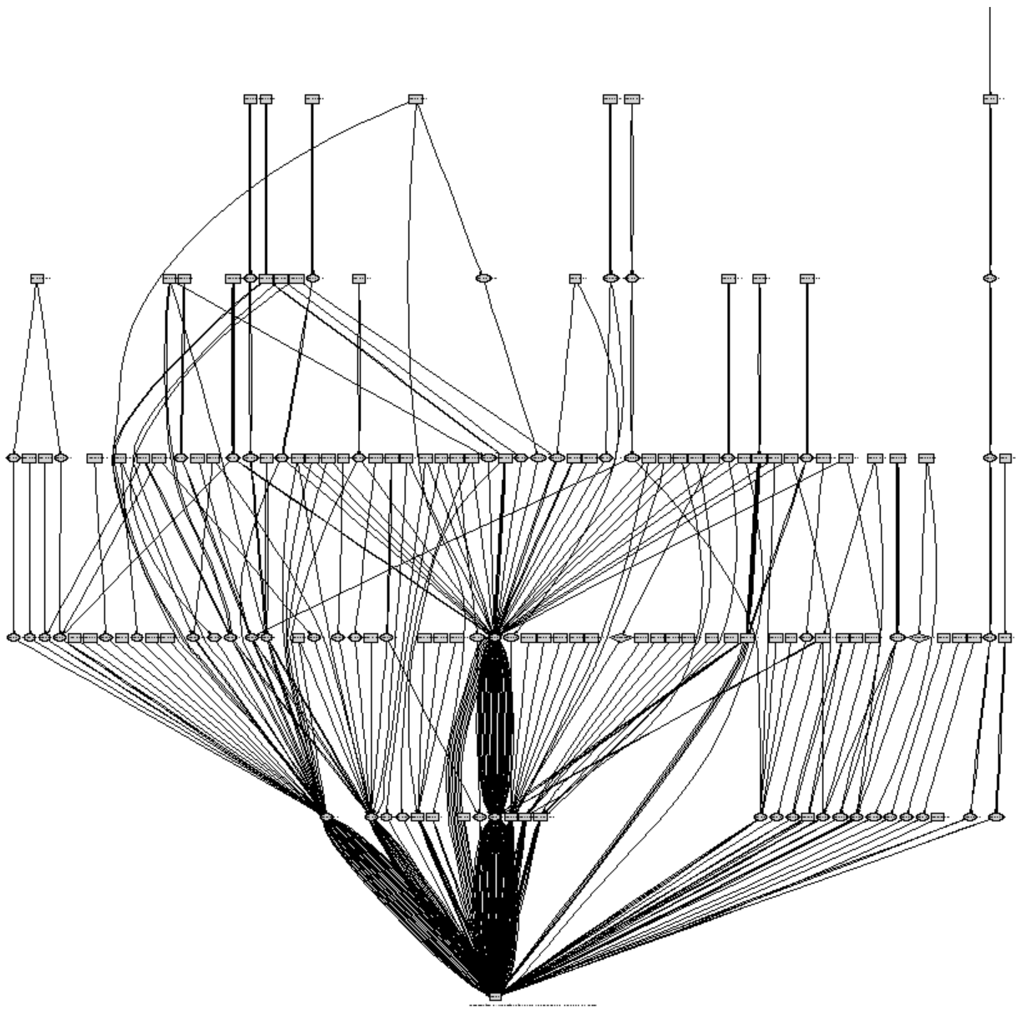


Figure 5.4: AS routes from Janet

5.3 Address space exhaustion

In the 1990s it was realised that the internet would run out of addresses, so a new internet protocol was agreed to replace the IPv4 protocol which used 32 bit addresses. The new protocol is IPv6 which uses 128 bit addresses, however there has been a delay in moving to the new standard. In the meantime two measures have enabled the internet to keep growing:

- CIDR, Classless Internet Domain Routing, which allows routing to occur to network addresses that do not conform to the standard IP address classes, and
- *connection sharing* or *masquerading* which allow a small network to share (or hide behind) one internet address.

5.3.1 CIDR

BGP-4 doesn't use the address classes to select network addresses, all destination networks are written in the subnet format: *network-number/length-of-address*. When the forwarding table is searched for a destination address every entry has an implied mask which is used to mask the incoming address and see if it matches the table entry. Longer masks are always tested first to ensure that small networks are not missed. This means that a fragment of a type A address can be allocated as a new network and will be found in the routing table.

One reason for the exhaustion of addresses was the wasteful allocation of type A and B network addresses (7 and 14 bits respectively) to organisations that would never fully use them. CIDR has allowed some of these to be sold off and broken into smaller network ranges, for example here are some real network numbers taken from part of one type A address space:

```
12.0.17.0/24
12.0.19.0/24
12.0.28.0/24
```

```
12.0.48.0/20
12.0.153.0/24
12.0.252.0/23
12.1.83.0/24
...
```

This works well but it does have the consequence that BGP routers have a very very difficult job to match an address, the incoming address must be masked with masks generated from each (or many) entries and the result compared with the table entry. It is no longer possible to select a network number by looking at the first bit or the first two bits. Many BGP routers have special hardware to help them search their tables.

It also means that tables get longer as type A networks are fragmented. However using CIDR can in some cases shorten the tables, consider the previous sample network picture, in the AS4 there are two close type C addresses: `235.11.8.0/24` and `235.11.9.0/24`. These have the same network prefix if the a 23 bit mask is used, they are both: `235.11.8.0/23`. This is called *aggregation*, all other ASs' routers need only one entry in their tables:

```
235.11.8.0/23: ..., AS4
```

because it will match both network addresses and forward them towards AS4, when the destination is reached AS4 can use a 24 bit mask to find the correct one.

5.3.2 Connection sharing

There are some addresses called "private" addresses that can be used for "disconnected" networks, they must not be used on the internet, `192.168.10.0` is one of them. A *firewall* or *gateway* has a single legal IP address and a private network behind it, see figure 5.5.

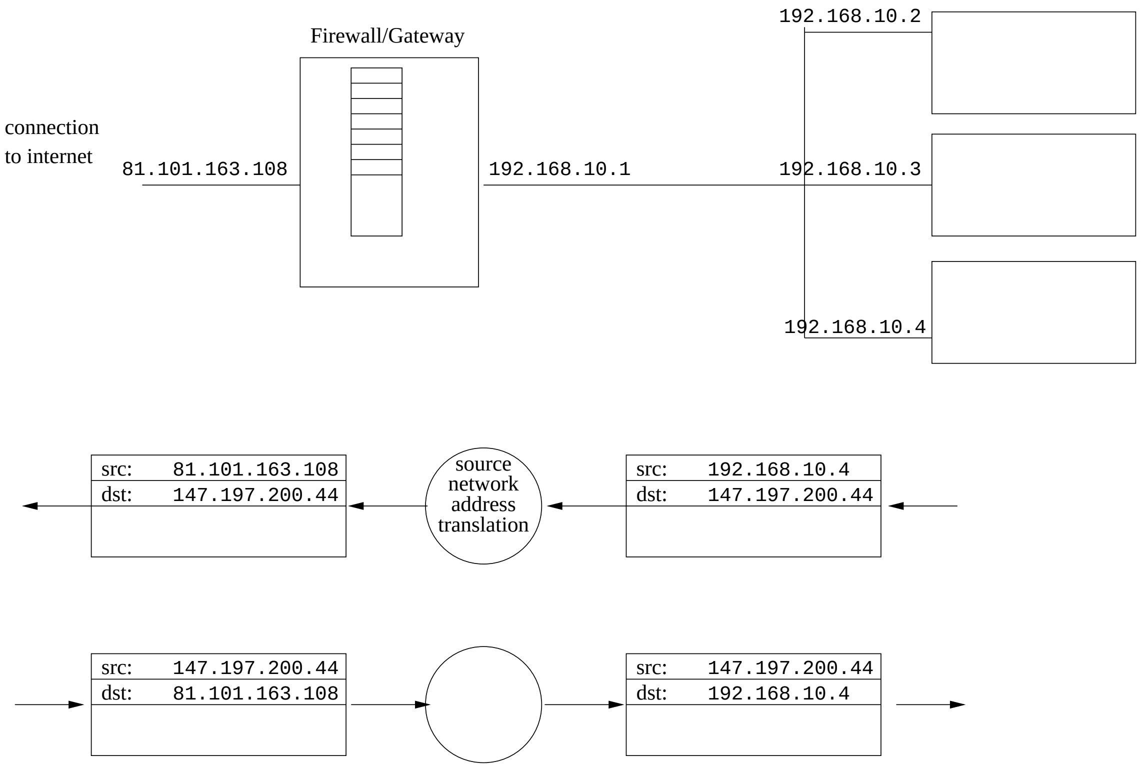


Figure 5.5: Connection sharing

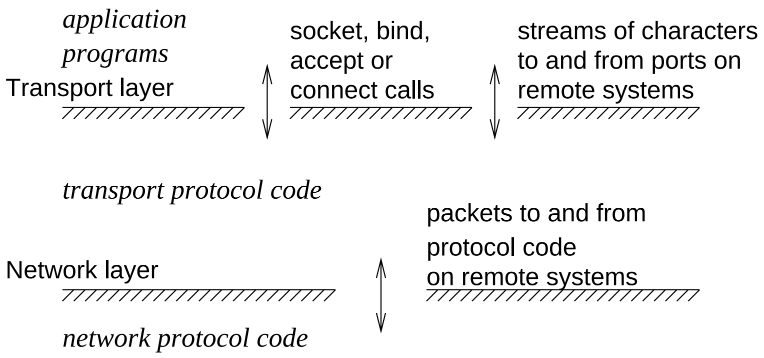
The gateway machine translates the *sender* address of every packet sent from the private net to the internet. It changes the private network address to its own IP address and records this in a *network address translation* table. When reply packets arrive back it looks up the table and reverses the translation, changing the destination from its own IP address to the correct private network address.

Chapter 6

The transport layer (TCP & UDP)

6.1 The function of the TCP layer

From "above" application programs require that the transport layer provide reliable streams of data to specific services on specific systems. The network layer (IP), "below", provides for the *unreliable* transmission of *fixed-sized* packets, in *any order* to specific remote *systems* (not ports) and *any* protocol family (not just TCP). It is the job of the transport protocol software to bridge the gap.


application programs
Transport layer
socket, bind, accept or connect calls
streams of characters to and from ports on remote systems
transport protocol code
packets to and from protocol code on remote systems
Network layer
network protocol code

The functions of the TCP protocol software are therefore:

- create and bind sockets for local applications and await connection request packets from remote programs,
- to establish connections from local programs to remote sockets,
- from programs, accept streams of characters on established connections and reliably transmit them to remote programs using "unreliable" packets provided by the network layer below.

6.2 End-to-end communication: ports

TCP connections are between processes, IP datagrams are between hosts. Therefore the TCP layer must support distributing the arriving datagrams to the appropriate server program. It uses *port numbers*, This is not a process number because they are transitory and vary, it is a "conventional" number that selects a service, there are fixed numbers for well known services like 21 for FTP, 80 for WWW and 23 for telnet. Numbers below 1024 are reserved, higher numbers can be used by anybody (but might clash with existing services, see the file /etc/services). A process that provides a service informs the system that it will accept connections to a given port number. When a remote process tries to ask for a service on the machine it must give the port number aswell as the address and the transport layer uses this to select which process to connect to.

TCP must record for every connection which process is bound to a port. It uses a unique 4 tuple to identify all connections:

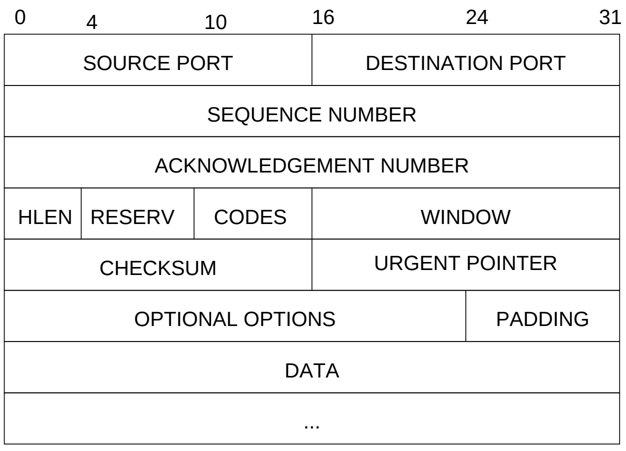
< *src-port, src-ipaddr, dst-port, dst-ipaddr* >

a server port is obvious but the port of the client is not obvious, what TCP does is to create a unique port number for every outgoing client connection. Consequently if one computer makes 2 telnet connections to the same remote machine each connection will have a different 4-tuple to identify it, here is part of the output from the `netstat` program:

```
rabbit(318)$ more netstat-n.out
Active Internet connections (w/o servers)
Pro RQ SQ Local Address     Foreign Address   State
tcp  0  0 192.168.1.2:1513  192.168.1.1:23    ESTAB
tcp  0  0 192.168.1.2:1514  192.168.1.1:23    ESTAB
tcp 32  0 62.252.84.12:1486 62.253.162.16:119 CLS_WT
...
```

6.3 TCP message format

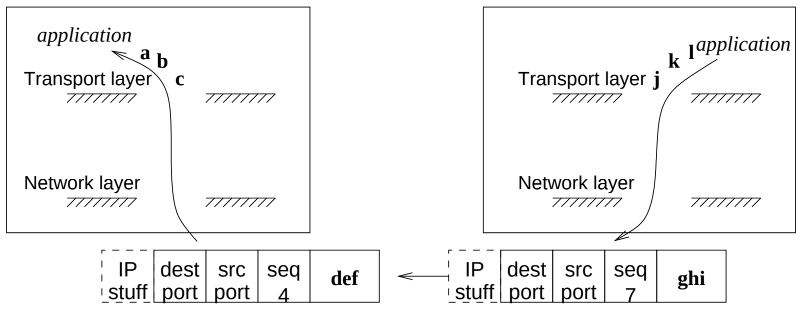
In order to create the reliable data stream the TCP layer exchanges messages with its “peer”. These messages are sent in IP datagrams and have a fixed format. They are used to establish connections, send data, send acknowledgements and close connections.



6.4 Streams in packets

The transport software, if it only has a packet-based network layer, must accept characters from the layer above and send them in a sequence of packets. However because on wide area networks there are alternative routes the first packet sent might arrive after the second one so for checking it will include a sequence number. In addition the network link (IP) can only send to a remote system so the transport layer must include the sender and recipient port numbers.

In the following picture the application on the right is attempting to send the stream of characters: `abcdefghijk` to a program on the system on the left:

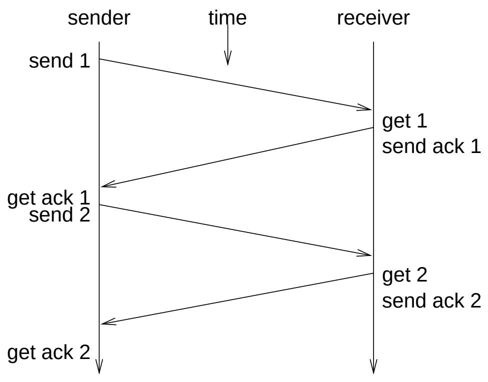


6.4.1 Problems

There can be problems: (i) the packets can go out of sequence, (ii) packets might get lost and never arrive, (iii) they might arrive but be corrupt, and (iv) packets might arrive faster than the receiver can deal with them. All of these are problems that must be solved by the transport (TCP) level software. The solution is to require acknowledgement of receipt of the packets and to retransmit them if they are not acknowledged.

6.5 Packet acknowledgement & retransmission

The simplest solution is that the sender can only transmit the next packet when the previous one has been acknowledged (ACK-ed):

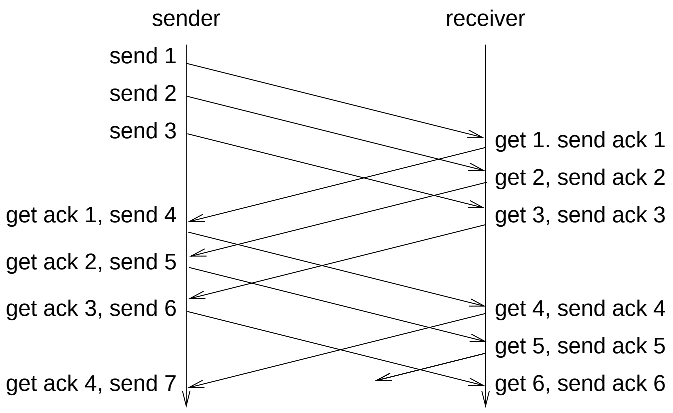


Whenever a packet is sent a timer is started, if a timeout occurs before an acknowledgement is received (which suggests a lost packet) the sender must re-transmit the last packet.

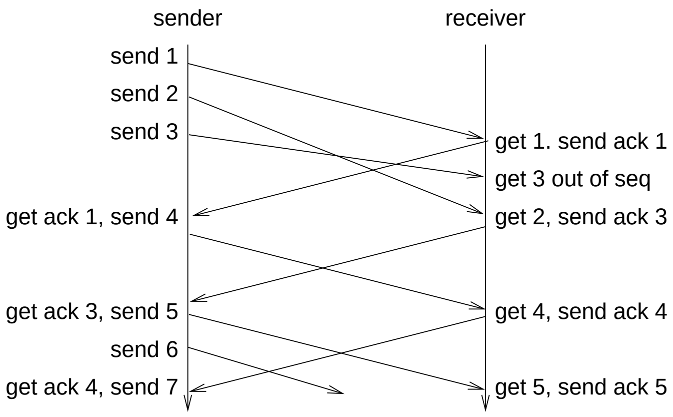
However this is very slow and wasteful, there might be several packets to send but they cannot be sent until the ACK is returned, that means waiting for the full RTT (round trip time) for the packet to reach the receiver and the ACK to get back.

6.6 Packet “windows”, the concept

An improvement is to have a *window* of packets awaiting acknowledgement. Both ends will agree a window size of *n* packets (in the next example *n*=3), this is the number of packets the sender can send without an acknowledgement. In the following diagram the sender sends 3 packets but must then wait for an ACK for packet 1 from the receiver. As soon as it gets the ACK it can continue to transmit packet 4.

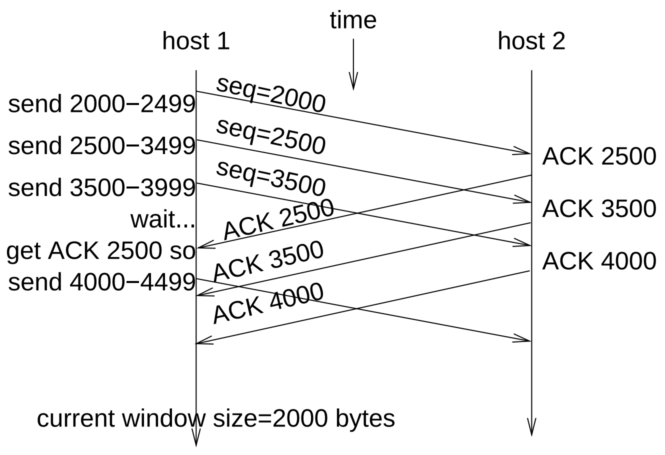


The additional overheads of this are that the sender must keep all the packets sent but not yet acknowledged. Packet windows also cope with out of sequence packets. This requires that the receiver will save packets got ahead of the sequence number it expects and further that it doesn't acknowledge until it has got all the ones up to the current sequence number.



6.7 Packet “windows” in TCP

TCP does not use a packet count for its sliding window, it uses the number of bytes in the stream of data it is sending; the acknowledgements are not for packets, they are for receipt of all bytes upto a position in the sequence. Depending on the speed of generation of data and the maximum packet size a sliding window of 4000 bytes might go in 40 packets or 1000 packets. However the basic operation is exactly the same as for packet based windows.



In the above picture the sequence at the start is 2000. The window size is 2000 bytes, sent in 3 packets: 500 bytes, 1000 bytes and 500 bytes, then the sender had to wait until the acknowledgement of the bytes 2000–2499 (they were acknowledged by sending the number of the next byte expected: 2500). When the ACK was received the sender could send upto 500 more bytes.

6.8 End to end flow control

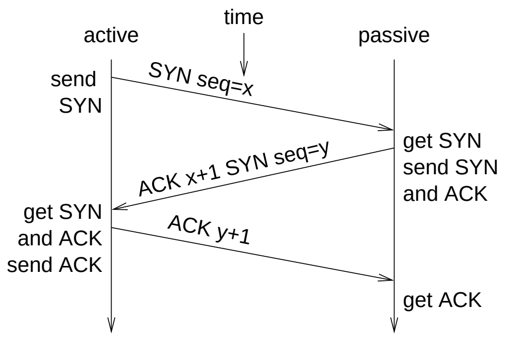
If the receiving host on one connection cannot keep up with the rate of arrival of packets because it has limited buffer space and its application isn't consuming the data fast enough then it can ask the sender to reduce the window size so it will not receive so much data, it can, if necessary, reduce the window to zero. It does this by using the WINDOW field in ACK packets.

6.9 Network congestion

DO NOT confuse with flow control. Sometimes Internet IP packet routers get overloaded and congested, if that happens they will have to discard some packets. What could happen, if the sliding window packet retransmission software is too simple, is that it will immediately respond by retransmitting all the lost packets. This will make the congestion worse! All "good" implementations of TCP should respond more gently—if packets timeout then the TCP sender will reduce the window size and delay before retransmitting, if it still has timeouts it delay even longer. It will only start increasing the window size and cutting the delay when it starts receiving acknowledgements again.

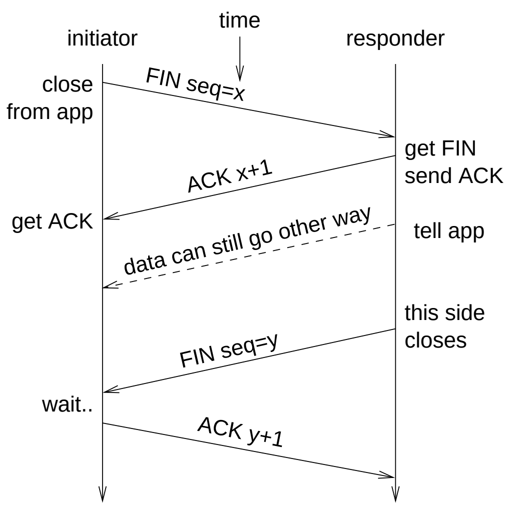
6.10 Opening and closing connections

To open a TCP connection the server executes `listen` and `accept`, this causes the TCP layer to "passively" open a connection, later a client executes `connect`, this is an "active". The TCP code carries out a 3-way hand-shake:



- A special packet flag is used SYN,
- each participant must select a random starting number for its sequence number (reduces risk of accidental capture of old packets from previous connections),
- the 3 messages ensure both sides know the connection is established, a lost SYN or ACK will cause retransmission.

Closing a connection is even more complicated:



- the close sequence uses a special flag: FIN,
- a close is only complete when *both* ends agree to close it,
- a connection is full duplex, one side might close its sending end of a connection if it has no more to send, but the other side might continue to send to it until it is finished,
- delays are needed to guarantee that no final packets are wandering around and might be picked up by a later connection,
- the intermediate ACK, from responder, even though it is not ready to send a FIN is to prevent the initiator resending the FIN.

Chapter 7

Java Network programming with sockets

The "*socket*" interface to TCP/IP dates from the early BSD Unix systems that first implemented TCP/IP about 1980. It is the primary interface between application programs and the transport layer. The transport layer is usually in the kernel of operating systems whereas higher level protocols are implemented by programs so the *socket* interface is usually a set of *system calls* (although on some systems like Sun Solaris or Windows Winsock it is a library with slightly different transport layer system calls below). In Java the socket library provides a slightly higher level view of sockets but is still quite close to the underlying system calls.

7.1 Addressing

A server must offer a service on a *port* address, and a client must connect to the servers *host* address and *port*.

7.1.1 The host address

Is a 32 bit number. It is usually represented as 4, 8 bit numbers separated by dots, for example:

```
147.197.205.101
```

all TCP/IP socket connections only use the IP number, there are no host names in TCP/IP (they are provided for users by a higher level application protocol). However under certain circumstances Java allows names or numbers to be used.

7.1.2 The port number:

The port number is used to select a process on a host. It is a "conventional" number that selects a service, there are fixed numbers for well known services like 21 for FTP, 80 for WWW and 23 for telnet. Numbers below 1024 are reserved, higher numbers can be used by anybody (but might clash with existing services, see the file /etc/services). A process that provides a service informs the system that it will accept connections to a given port number. When a remote process asks for a service on the machine it must give the port number aswell as the address and the transport layer uses this to select which process to connect to.

7.2 Socket usage is asymmetric

No matter whether the network application is *client-server* or *peer-to-peer* whenever one program must contact another there is asymmetry in the use of sockets. One will wait to *accept* a connection and another must *connect* to it.

7.3 Socket streams and datagrams

There are 2 forms of transport level network interprocess connection with the TCP/IP family of protocols:

TCP a bi-directional stream connection. The stream is "reliable" which means the underlying network level requires acknowledgement of each packet sent in the stream, if any are lost then they are re-transmitted transparently to the process using the stream.

UDP a connectionless single message, or datagram. There is no guarantee of delivery of a UDP datagram (although in practice nearly all packets get through).

7.4 Unix sockets system call interface

A *socket* appears to a user process as a file descriptor on which `read`s and `write`s can be performed. There are various calls to set up a connection on a socket and use it:

`fd=socket(proto,type,?)` creates an unconnected socket, `bind(fd,struct sockaddr *ptr,len)` associates a port number with a socket. It is used by a process to inform the operating system it will deal with any connections to a port and provide the service.

`listen(fd,conn_q)` used by a process to indicate that it is prepared to receive connections, that it is a server. It doesn't wait, `accept` does that ...

`fd2=accept(fd1,struct sockaddr *sender,len)` this causes a process to wait for a connection. When if arrives the connecting process's address is returned in the `sockaddr` address structure. Also a new file descriptor is created that can be used to talk to the remote process,

`connect(fd,struct sockaddr,len)` this is used by a process to make a connection on a socket to an address contained in the `sockaddr` structure.

Once the connection is established characters can be written to and read from it using the `read()` and `write` and other system calls.

Notice that the asymmetry of the client server communication is reflected in which system calls are used. This is is illustrated in the picture 7.1.

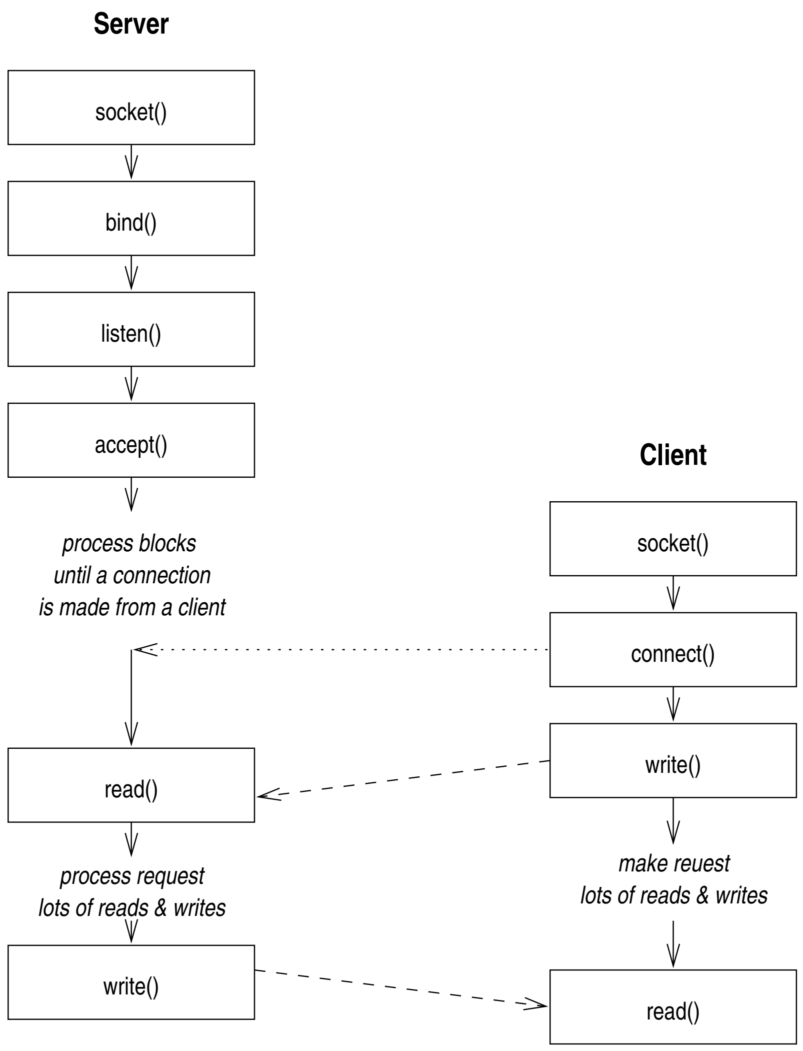


Figure 7.1: System call sequence

7.5 Java sockets API

The BSD sockets are available in Java through the java.net.* package. There are two main classes: `Socket` for connected sockets, and `ServerSocket` for listening sockets.

- `ServerSocket` when created it is bound to a port and it will receive incoming connections to that port. It uses the BSD calls: `socket`, `bind` and `listen`. The main operation is:

  ```
  connSock = serverSock.accept();
  ```

which waits for an incoming connection. When one arrives it returns an ordinary `Socket` connected to the remote program.

- `Socket` a connected socket, a bi-directional communication stream between two possibly remote programs. There are 2 ways to create a connected `Socket`:
  - get one back from a `ServerSocket accept`,
  - to create one and attempt to connect to a remote system:

    ```
    Socket sock;
    sock = new Socket(hostname,port);
    ```

    which will attempt to establish a connection to the remote system `hostname` on their `port`.
- whichever way a connected socket is produced there are methods to get an `InputStream` and an `OutputStream` from it using `getInputStream` and an `getOutputStream` respectively. These streams are exactly the same as the streams returned when you open files, and they can be used in the same way with `read` and `write`. Except `reading` and `writeing` these streams will receive and send data to the other program to which the socket is connected.

7.6 A client example

The following example just illustrates a simple client program, it takes as arguments: an internet address and a WWW page name.

```
import java.io.*;
import java.net.*;

public class HTTPGet2 {
   public static void main(String[] args) {

      final int BUFSIZ=8192;
      Socket socket = null;
      OutputStream toServer = null;
      InputStream fromServer = null;
      int rc, port = 0;
      String request;
      byte buffer[] = new byte[BUFSIZ];

      if( args.length == 0 ) {
         System.out.println(
              "Usage: HTTPGet2 server file [port]");
         System.exit(1);
      } else if( args.length == 3 ) {
         port = Integer.parseInt(args[2]);
      } else {
         port = 80;
      }
      try {
         socket = new Socket(args[0], port);
         toServer = socket.getOutputStream();
         fromServer =  socket.getInputStream();
         request = "GET " + args[1] + " HTTP/1.1\r\n"
             + "Host: " + args[0] + "\r\n"
             + "Connection: Close\r\n\r\n";

         toServer.write(request.getBytes());

         rc = fromServer.read(buffer,0,BUFSIZ);
         while (rc > 0) {
            System.out.write(buffer,0,rc);
            rc = fromServer.read(buffer,0,BUFSIZ);
         }
         toServer.close();
         fromServer.close();
         socket.close();
```

```
      } catch (UnknownHostException e) {
         System.err.println("Can’t find: " + args[0]);
         System.exit(1);
      } catch (IOException e) {
         System.err.println("IO error");
         System.exit(1);
      }
    }
}
```

The program is in the file HTTPGet2.java. This program will act as a dumb client. It will send a request to a remote http server. To compile and run the program:

```
 sally(373)$ javac HTTPGet2.java
sally(374)$ java HTTPGet2 slink.feis.herts.ac.uk /tiny.html
HTTP/1.1 200 OK
Date: Sun, 16 Mar 2003 23:32:10 GMT
Server: Apache/1.3.26 (Unix) Debian GNU/Linux
Last-Modified: Wed, 08 May 2002 23:45:10 GMT
Accept-Ranges: bytes
Content-Length: 492
Content-Type: text/html; charset=iso-8859-1
Connection: close

<H1> Example Page </H1>
This is the first paragraph, it is terminated by a
...
```

which will get tiny.html from slink.feis.herts.ac.uk. Notes:

- first it checks the command line arguments, if there is no port number provided the program will use `80`,
- all the code to open the connection and read and write the streams might produce horrible *exceptions* so the body of the program is surrounded by `try{..}catch{..}`,
- first attempt to connect to the server by creating a new socket using the remote system name (or number) and the port:

  ```
       socket = new Socket(args[0], port);
  ```

  if this fails an exception will be raised,
- now extract the input and output streams:

  ```
     toServer = socket.getOutputStream();
     fromServer =  socket.getInputStream();
  ```

- now build a full HTTP file request as a string in `request`,
- and send it to the server:

  ```
     toServer.write(request.getBytes());
  ```

  note that since it is a stream we use `write` which requires an array of bytes, `getBytes` will get such an array out of the string `request`. Now the message is sent to the server,
- if the server exists and if it reads the request, and if it thinks our request well-formed and if it has such a file then it will send it down the same connected socket. We must `read` the socket to get the returning file:

  ```
      rc = fromServer.read(buffer,0,BUFSIZ);
  ```

  `read` puts the characters read into a pre-allocated array of bytes, here called `buffer`. The return result, put in `rc`, is the number of characters actually put in `buffer`. The client cannot know how big the file is (if it’s an MPEG video it might be megabytes), so it reads in “chunks” of 8k, that is why there is a loop, that reads and then prints to `System.out`,
- when we can read no more (`rc > 0` is not true) we close everything and finish.

7.7 A cutdown version

This is the same as the previous version but all the checking of arguments and exception handling is removed. Not good, but maybe it is easier to focus on the network code:

```
import java.io.*;
import java.net.*;

public class HTTPGet0 {
   public static void main(String args[])
                           throws Exception {
      Socket socket = null;
      OutputStream toServer = null;
      InputStream fromServer = null;
      int rc, port = Integer.parseInt(args[2]);
      String request;
      byte buffer[] = new byte[8192];

      socket = new Socket(args[0], port);
      toServer = socket.getOutputStream();
      fromServer =  socket.getInputStream();

      request = "GET " + args[1] + " HTTP/1.1\r\n"
              + "Host: " + args[0] + "\r\n"
              + "Connection: Close\r\n\r\n";

      toServer.write(request.getBytes());

      rc = fromServer.read(buffer,0,8192);
      while (rc > 0) {
         System.out.write(buffer,0,rc);
         rc = fromServer.read(buffer,0,8192);
      }
      toServer.close();
      fromServer.close();
      socket.close();
   }
}
```

The program is in the file HTTPGet0.java.

7.8 Client server example `echo`

This example consists of a server and a client. They do very little except show how a stream connection is set up. The server awaits (`accept`) a connection, reads lines from the client and immediately sends them back again. When the connection from a client is closed (a `null` return from `readLine`) the server loops to `accept` the next connection from another client. The client makes a connection and then loops each time: reading from the user, writing this text to the server, reading the server's response (which should be the same) and then printing it. The server:

```
import java.io.*;
import java.net.*;

public class EchoServer {
   public static void main(String[] args) {

      ServerSocket serverSock = null;
      Socket connSock = null;
      PrintWriter out = null;
      BufferedReader in = null;
      int echoPort = -1;
      String fromUser;

      if( args.length != 1 ) {
         System.out.println("Usage: EchoServer port");
         System.exit(1);
      } else {
         echoPort = Integer.parseInt(args[0]);
      }
      try {
         serverSock = new ServerSocket(echoPort, 10);

         while(true) {
            connSock = serverSock.accept();
            System.out.println("Got connection from "
             + connSock.getInetAddress().getHostName());
            out = new PrintWriter(
                     connSock.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(
                            connSock.getInputStream()));

            fromUser = in.readLine();
            while (fromUser != null) {
               out.println(fromUser);
               fromUser = in.readLine();
            }
            out.close();
            in.close();
            connSock.close();
         }
      } catch (IOException e) {
         System.err.println("EchoServer: error opening,"
                      + " accepting or reading socket");
         System.exit(1);
      }
   }
}
```

The program is in the file EchoServer.java.

- notice that the server loops forever:

  ```
  while(true) {
    ...
  ```

  nearly all servers are like this, deal with one request and loop to "`accept`" the next,

- this is a server so it must create a `ServerSocket` bound to a port number. The port number to use is provided as an argument. It then waits by calling `accept`,

- this network program reads and writes lines not single characters, it could have used characters but I thought a bit of variety would be fun. So it has to create a `BufferedReader` and a `PrintWriter`,

- it then loops reading lines from the client and writing them back again. When it gets `null` from `readLine` (which would be end of file for a file) it means the client closed the connection.

Now the client, this is a cutdown, non-error checking one:

```
import java.io.*;
import java.net.*;

public class EchoClient0 {
   public static void main(String[] args)throws Exception{
      Socket echoSocket = null;
      PrintWriter out = null;
      BufferedReader in = null;

      echoSocket = new Socket(args[0],
                              Integer.parseInt(args[1]));
      out = new PrintWriter(
                  echoSocket.getOutputStream(), true);
      in = new BufferedReader(new InputStreamReader(
                              echoSocket.getInputStream()));

      BufferedReader stdIn = new
          BufferedReader(new InputStreamReader(System.in));
```

```
      String userInput;

      userInput = stdIn.readLine();
      while (userInput != null) {
          out.println(userInput);
          System.out.println("echo: " + in.readLine());
          userInput = stdIn.readLine();
      }
      out.close();
      in.close();
      stdIn.close();
      echoSocket.close();
   }
}
```

The program is in the file EchoClient0.java. This is very similar to HTTPGet.java except, of course, it reads from a user, writes to the server, reads the response and displays it. To test the client server programs: compile them both, run the server with an arbitrary port number:

```
  tink(257)$ java EchoServer 3333
  Got connection from 147.197.236.188
```

Then in another xterm run the client:

```
  slink(258)$ java EchoClient0 tink 3333
  hello
  echo: hello
  ...
```

the line "`hello`" is read from the user, sent to the server returned by it, read from the socket by the client and then printed "`echo:  hello`". Unlike the server the client only deals with one session, it only has one loop to read and echo, when the user finishes (by typing control-d "`^d`" on Unix) the loop finishes and the program finishes.

7.9 Threads

A *thread* enables one part of a program to be executed logically in parallel with another part. If we create a new thread and start it then it will share CPU time with the main program (also a thread) and any other threads. There are two ways to write Java threads (i) to *implement* the `Runnable` interface, or (ii) to inherit from the `Thread` class. We will show the second because it is slightly simpler.

In order to write a thread it is necessary to provide (i) a constructor to set any attributes, and (ii) a single function: `public void run()` which will be the separately scheduled code. Here is a very simple example that declares one thread class, then creates and starts two thread objects:

```
import java.net.*;
import java.io.*;

class Loopy extends Thread {
   String message;

   Loopy(String mess) {
      message = mess;
   }
   public void run() {
      while(true) {
         System.out.println(message);
      }
   }
}
public class Threads0 {
   public static void main(String[] args) {
      Thread thread1 = new Loopy("One        ");
      Thread thread2 = new Loopy("         Two");
      thread1.start();
      thread2.start();
   }
}
```

The program is in the file Threads0.java. When the threads are started they execute their `run` routine for ever repeatedly printing out their message. If this is compiled and run it can produce almost any output sequences deoending on how the threads are *scheduled*, which means how they are allocated a share of the CPU time. Here is part of one sequence:

```
         Two
         Two
         Two
         Two
One
         Two
One
         Two
One
```

7.10 A concurrent server

If a server has to deal with a long transaction for a client, involving lots of waits for reading and writing files and sockets, it will be unable to `accept` new requests. One simple solution is to change the server so that after the `accept` it creates a “child” *thread*. This new thread uses the new “connected” socket to service the clients request, and then dies. The parent thread goes back to `accept` to await another connection. This is called a *concurrent server*.

```
import java.io.*;
import java.net.*;

class ServiceThread extends Thread {
   Socket conn;

   public ServiceThread(Socket c) {
      super("EchoServer service thread");
      conn = c;
   }
   public void run()  {
      String fromUser;
      PrintWriter out = null;
      BufferedReader in = null;
      try {
         System.out.println("Got connection from "
            + conn.getInetAddress().getHostName());
         out = new PrintWriter(conn.getOutputStream(), true);
         in = new BufferedReader(
              new InputStreamReader(conn.getInputStream()));

         fromUser = in.readLine();
         while (fromUser != null) {
            out.println(fromUser);
            fromUser = in.readLine();
         }
         out.close();
         in.close();
         conn.close();
      } catch (IOException e) {
         System.err.println("EchoServer: socket error");
         System.exit(1);
      }
   }
}
public class EchoServerConc0 {
   public static void main(String[] args)throws Exception{

      Socket connSock = null;
      int echoPort = Integer.parseInt(args[0]);
      ServerSocket serverSock = new ServerSocket(echoPort, 10);
      ServiceThread serve = null;

      while(true) {
            connSock = serverSock.accept();
            serve = new ServiceThread(connSock);
            serve.start();
        }
    }
}
```

The program is in the file EchoServerConc0.java.

- After the `accept` a new thread is created and given the connected socket `connSock`,
- the new thread is now `start`ed, and runs in parallel with other threads and `main`,
- the parent (main) thread loops to do `accept` again,
- the child thread runs and handles the client transaction, when this is finished it reaches the end and terminates.

Chapter 8

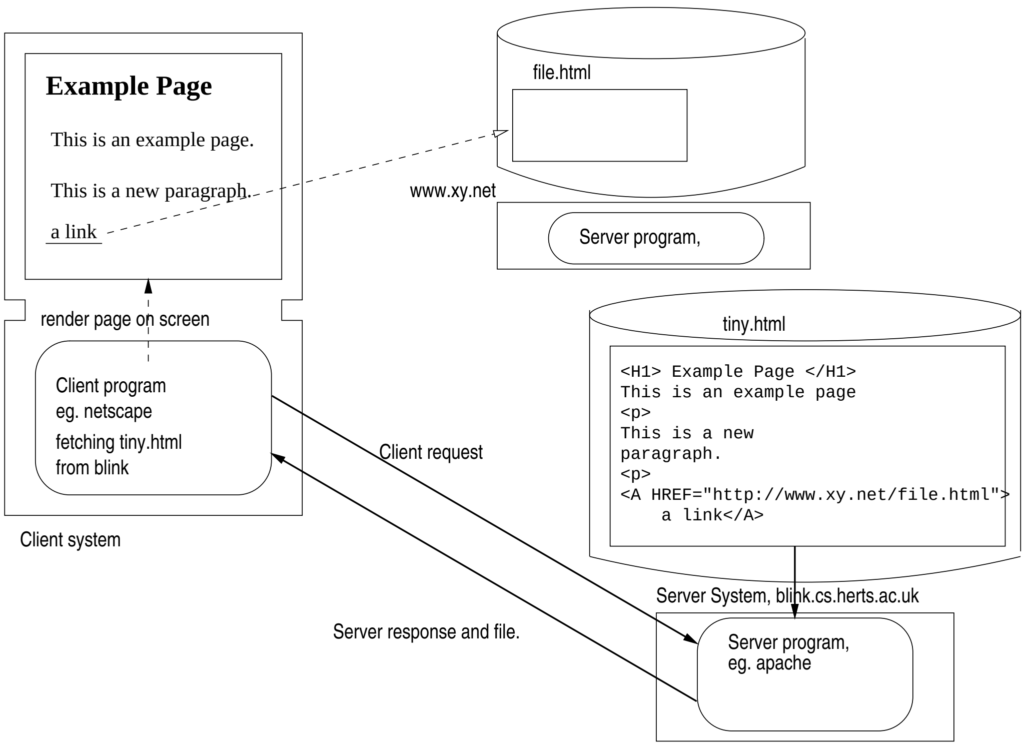
WWW, HTTP, HTML, CGI and PHP

These notes are an introduction to how the World-wide-web works. The treatment of topics is not uniform, the notes are meant to survey nearly all aspects of the Web and in addition a more detailed treatment of CGI programs work. The material is organised as follows:

- A brief description of HTML, used for writing Web files,
- Something on HTTP,
- Quite a lot about CGI, the way in which programs are executed on a web server.

8.1 Overview of WWW

The World-Wide Web is based on a simple protocol called HTTP that allows browser programs such as netscape, kfm or internet explorer, to fetch files from remote server programs, for example apache, and to view them (the files are often called *pages*, which is odd because they are files!). WWW files are named by URIs which have a special format that includes the remote server name and the name of the file (note: a URI is sometimes called a URL). The files can be written in a special document description language called HTML that is interpreted by the browser to give an appealing visual effect on a graphical display. The HTML files can contain embedded URIs that refer to other WWW files, these are usually highlighted by the browser and if selected will cause retrieval of the named file. Such references are sometimes called *hyper-links*, it is the use of these that produce a "web" and give the web (HTTP, URIs, and HTML) its power.



```
<H1> Example Page </H1>
This is an example page
<p>
This is a new
paragraph.
<p>
<A HREF="http://www.xy.net/file.html">
    a link</A>
```

Figure 8.1: A client and a server

Figure 8.1 shows a client program requesting a file, tiny.html from a server system. The file is a text file on the server's disc, it contains source HTML. The client has sent an HTTP protocol request to the server, the server sent the file to the client, and the client program (netscape) has interpreted the HTML and displayed the result on the client computer's screen. The file tiny.html contains an `HREF`, a hyper-link:

```
<A HREF="http://www.xy.net/file.html"> a link</A>
```

that if selected will cause the browser to retrieve a file from www.xy.net.

8.2 HTML

Any type of file can be retrieved by a browser from a server: images, sound files, text files, or PDF; the action taken depends on settings in the browser and its capabilities. For example most browsers can interpret JPEG files themselves but with an MPEG movie they will execute a separate viewer program.

However, by far the commonest content of web files is HTML. HTML is a *mark-up* language, which means it describes the layout of pages that can be interpreted to produce a readable image. Other examples of mark-up language are $\TeX$, $\LaTeX$, SGML (which is a meta markup language) and XML. Nearly all web browsers can interpret and display HTML, though there are some text-oriented browsers like lynx that interpret HTML but only display the results in a non-bitmapped display form.

HTML is a simple language:

- ordinary text is interpreted as itself and rendered in the current font and size,
- anything surrounded by `<..>` brackets is a formatting instruction.

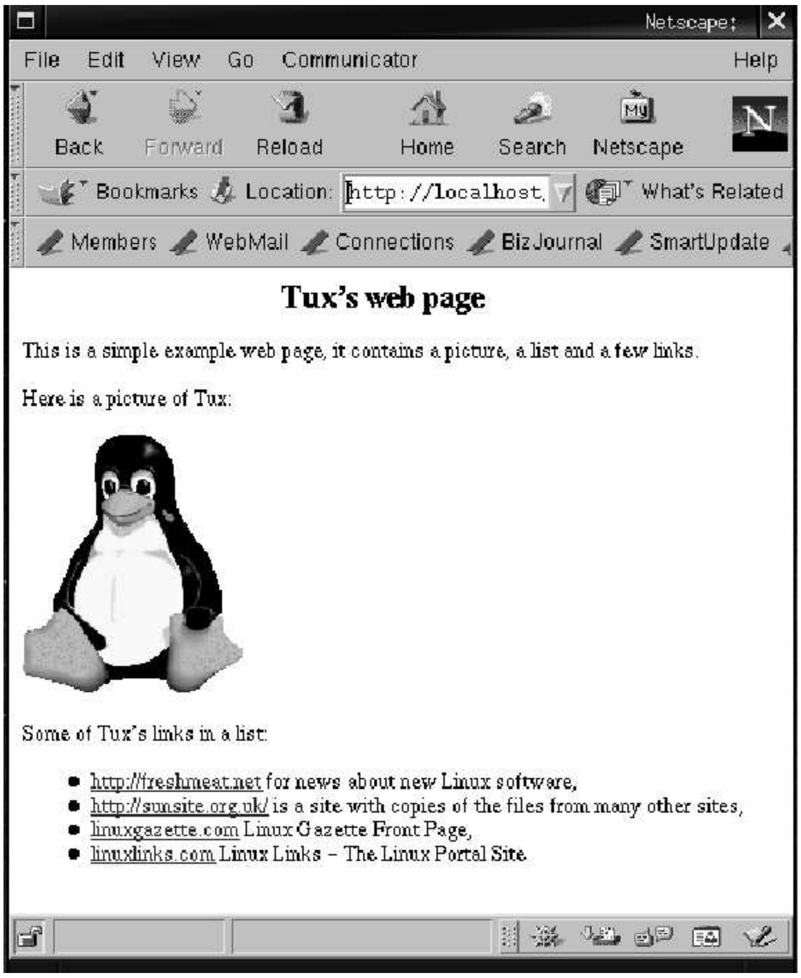
Many formatting instructions "bracket" the text they apply to. For example:

```
<H1> This is a Heading </H1>
```

Will cause the text `This is a Heading` to be set as a "level one heading", meaning bold and large. Notice that the formatting is introduced by `<H1>` and ended by the same directive with "`/`" in front: `</H1>`.

8.2.1 HTML file example

These notes are not intended to provide a proper introduction to HTML, they are just an overview so the simplest way is by example. The result of looking at the file with netscape is presented first followed by the text of the file:



Tux's web page

This is a simple example web page, it contains a picture, a list and a few links.

Here is a picture of Tux:

Some of Tux's links in a list:

- http://freshmeat.net for news about new Linux software,
- http://sunsite.org.uk/ is a site with copies of the files from many other sites,
- linuxgazette.com Linux Gazette Front Page,
- linuxlinks.com Linux Links – The Linux Portal Site.

The above picture used the URI http://localhost/example.html to retrieve the file from the server on my own machine but http://blink.feis.herts.ac.uk/example.html should get a very similar file. Now the source of the file:

```
<body  bgcolor="#FFFFFF" >

 <h1 align="center"> Tux's web page </h1>
 <p>
  This is a simple example web page, it contains a picture,
  a list and a few links.
 <p>
  Here is a picture of Tux:
 <p>
 <img width=128 height=150 src="PenguinMascot.gif">
 <p>
 Some of Tux's links in a list:
 <ul>
  <li>
   <A HREF="http://freshmeat.net/"> http://freshmeat.net</A>
    for news about new Linux software,
  <li>
   <a href="http://sunsite.org.uk/"> http://sunsite.org.uk/</A>
    is a site with copies of the files from many other sites,
  <li>
   <A HREF="http://www.linuxgazette.com/"> linuxgazette.com</A>
    Linux Gazette Front Page,
  <li>
   <A HREF="http://www.linuxlinks.com/"> linuxlinks.com</A>
   Linux Links - The Linux Portal Site
 </ul>

</body>
```

Notes:

1. The language is not case-sensitive, `<H1>` means the same as `<h1>`.
2. The file contents are surrounded by `<body> .. </body>`; the opening declaration is followed by an option that sets the background colour: `<body bgcolor="#FFFFFF">`. Other options can be set.
3. The `<h1>` surrounds text that will be set large and bold, `<h2>` is slightly less large, etc. Like the `body` declaration it can be followed by an option, in this case to centre the heading.
4. The `<p>` starts a new paragraph, it is one of the few directives that doesn't need a matching "slash" terminator.
5. Images can be included using the `<img ..>` directive. Once again there is no terminator.
6. The `<ul>..</ul>` is a "bullet" list. Each item of the list is introduced by `</li>`.
7. The `<a..>..</a>` is a link. The `HREF` selects the destination of the link, the rest of the text between `<a ..>` and `</a>` is displayed underlined so:

   ```
   <a href="http://freshmeat.net/"> freshmeat</A>
   ```
   will display: <u>freshmeat</u> on the screen, which will, if clicked, retrieve the index file from freshmeat.net.

8.3 URIs and where files are kept

8.3.1 URI

URI stands for universal resource identifier, they are are often called URLs but according to the HTTP standard they are URIs and there is no important difference. The format is:

scheme `://` *hostname* `[ :` *port* `] /` *path*

where *scheme* is the protocol, `http` or `ftp`, *hostname* can be a fully qualified domain name or a numeric IP address, the port number is optional and if omitted defaults to 80, and *path* is a "`/`" separated list of names selecting the required file or directory. For example:

```
http://humbolt.nl.linux.org/Linux-MM/internals.html
```

The URI is taken apart by the browser which uses the scheme to select the protocol, the hostname and port to make the connection so all it actually sends in a request is the path.

8.3.2 Where files are stored

The server program chooses how to interpret the path. Usually it has a special directory tree where all its files are kept and the requested path is prefixed by that. The "root" can be anywhere, some common examples are: /home/www, /usr/local/htdocs. So the requested path /Linux-MM/internals.html might map to a host file /home/www/Linux-MM/internals.html.

Some servers allow files to be requested from users "home" directories. If the path contains a "~*username*" this is interpreted as a request for a file in *username*'s home directory. To avoid remote access to all a user's files the request is usually mapped to a sub-directory of the home directory called public_html, so:

```
http://blink.cs.herts.ac.uk/~aa9zz/my.html
```

might be mapped to:

```
/home/student/aa9zz/public_html/my.html
```

8.3.3 Directories and index.html

Very often URI request paths actually name directories. What is returned in these cases? Some servers will look for a file called index.html in the named directory and return that file. So the simple request `http:/www.w3.org/` will retrieve a file called index.html from the "root" of `www.w3.org`'s server file hierarchy. If a directory is named and there is no index.html then some servers will read the directory contents and turn it into HTML form with each name turned into an "href" and return that.

8.4 HTTP

HTTP is the protocol used to communicate between a client and a server. HTTP defines what characters can be sent along the socket stream connection.

The basic protocol is *request* and *response*. The server accepts a connection and the client sends a request command line, various optional MIME lines and then a blank. The server must then send a response line giving a success or failure code, followed by additional optional lines, then the blank line and finally, if a file was successfully requested, the file contents (whether HTML, GIF or whatever).

That's it. Except to look at a request and a response...

Using a "dumb server" it is possible to capture and print the HTTP sent by clients. The program binds a high numbered port, say 8080, and accepts connections. It then just reads all the data from the socket and prints it on the standard output. Then it just closes the socket and causes the client to report an error.

This is the HTTP request and options sent from netscape when it was given a URI like:

```
http://localhost:8080/abc.html
```

The standard output from the "dumb server" was:

```
GET /abc.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.7 [en] (X11; I; Linux 2.3.34 i686)
Host: localhost:8080
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

There are only 2 really essential parts:

1. the request line. It include the command, in this case `GET`, the request file name, here `/abc.html` and the protocol version.
2. the other essential part is the blank line at the end.

The other lines are optional. Some are very important and useful but are not obligatory.

Similarly it is possible to examine the *responses* from servers by using a "dumb client" that sends a request to a server and prints out the complete response. This will show the HTTP response line with the status code, various optional lines, a blank line then the retrieved file if any. It is not possible to see the reponse from servers using a normal browser because they process the response lines and don't show them. The example apache responded to:

```
GET /tiny.html HTTP/1.0
```

By sending back:

```
HTTP/1.1 200 OK
Date: Wed, 19 Jan 2000 01:43:00 GMT
Server: Apache/1.3.9 (Unix)
Last-Modified: Sun, 09 Jan 2000 23:41:23 GMT
ETag: "d112-1ec-38791ca3"
Accept-Ranges: bytes
Content-Length: 492
Connection: close
Content-Type: text/html

<H1> Example Page </H1>

This is the first
paragraph, it is terminated by a
...
```

8.5 Client and server additional services

Very early in the history of the web it was discovered that just returning HTML pages was very limited. People wanted to add more computational power so that users could interact with the web. One of the first such additional features added were CGI programs that allow browser requests to cause programs to execute on the web server. This revolution enabled people to develop search engines, database access through the web, and sites providing e-commerce. In addition more interesting web pages were provided by allowing programs, sent in web pages, to be executed by the browser. This allowed animation and other dynamic features. There are different forms of these extensions to web functionality:

- *server-side* facilities, these allow programs to be executed on the web server that can access and update databases or carry out financial transactions. There are two main ways this is now provided:
  - CGI programs, these can be written in any language, and, within certain security limits, carry out almost any task. When a client request is sent the URI can name a CGI program rather than an HTML page, it is then executed. They are very powerful but can be hard to write.
  - *server-side includes*, or SSI, these are HTML files with special additions that allow other files to included. This facility permits, for example, a site to use standard headers and footers on all their pages and to change the appearance of all of them without having to edit them all separately. However they do not have the power to execute programs, they provide a different functionality.
  - *executable web pages*, also called *active-server pages*. These consist of HTML and a programming language interleaved in the same file (or page). When one of these files is named by a URI, sent with a client request, the server program itself (or a special interpreter run by the server program) "executes" the page. This "execution" involves sending any HTML straight back to the client browser and executing any bits of the programming language found. This allows a simple way of sending back HTML and and at the same time executing commands that can for example access a database.

    There are alternative languages available, some examples are:
    - PHP, an open source, free system that works with the Apache web server on any platform,
    - Microsoft's ASP which uses VBScript, and
    - JSP, Java Server Pages.
- *client-side* services, these involve extensions to, or commands in, the HTML file sent to the client's browser. When the browser encounters these it will "execute" them. This has entirely different advantages from the server side facilities: they can be used to animate pages, they can check user's input before it is sent back to the server, and many other tasks. They cannot be an alternative to central server programs, they are executed in the browser. There are two forms:
  - languages that can be embedded in the page and executed by the browser, Javascript (not related to Java) is the most widely used example of this,
  - special purpose languages or programs that need a browser "plugin" to interpret them. Flash is one example of this as is Java.

In the following sections some of these will be examined a bit further. First, CGI, and then some PHP, a bit of SSI and lastly a tiny bit of Javascript.

8.6 Server side: using forms for interaction

Before starting on the details of CGI or PHP this section will summarise how “executable” server-side features are invoked and how users interact with them. In nearly all cases server-side programs require some input from the user, this must be sent from the browser, the commonest method is to use the HTML form. A form displays boxes or buttons on the browser screen that the user can fill in. There is a button to send the values from the form back to the server along with a URI as part of an HTTP request. The requested file (page) is usually an executable (CGI, PHP or ASP), the server program executes it and gives it the input from the form. The executable will run, carrying out its task, and send some HTML back to the browser as a response. See figure 8.2. Here is an example of a very simple HTML file with a form in it:

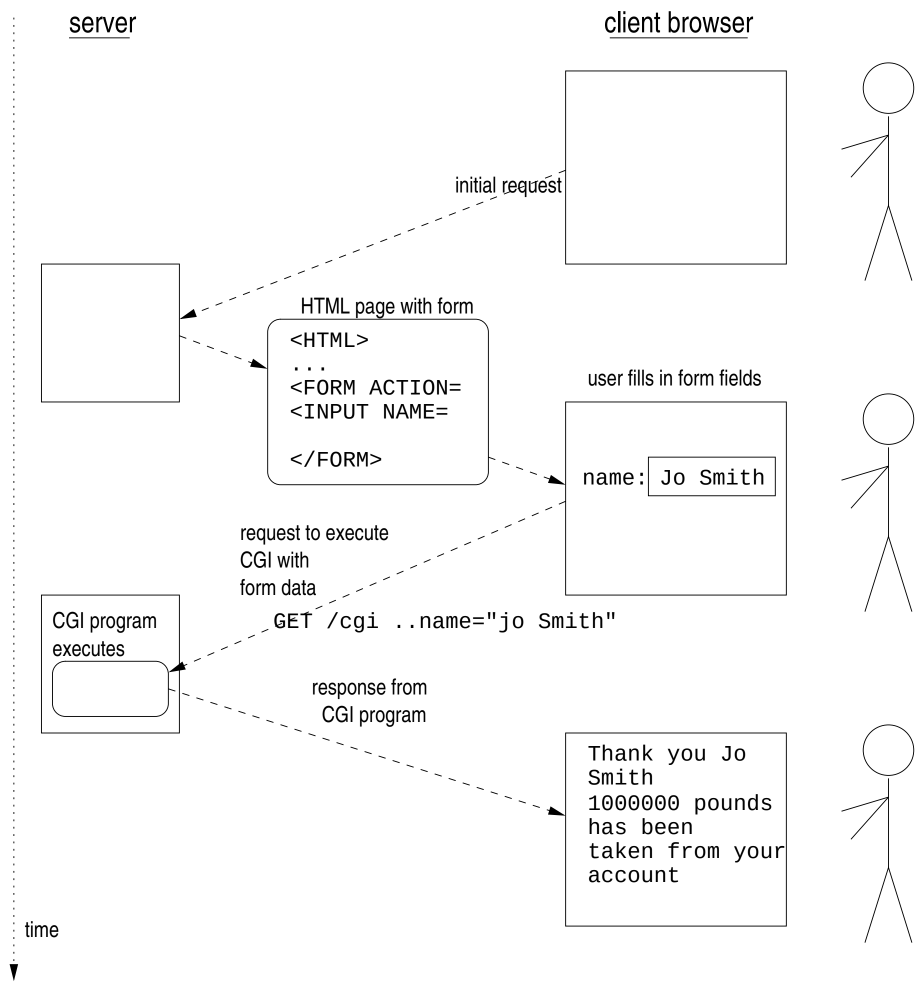
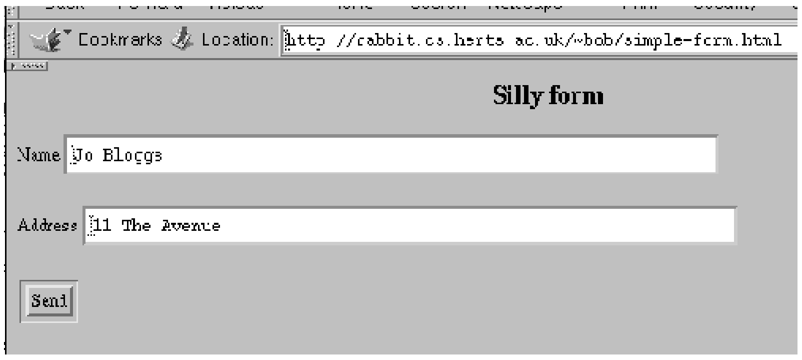

server
client browser
initial request
HTML page with form
<HTML>
...
<FORM ACTION=
<INPUT NAME=
</FORM>
user fills in form fields
name: Jo Smith
request to execute CGI with form data
GET /cgi ..name="jo Smith"
CGI program executes
response from CGI program
Thank you Jo Smith 1000000 pounds has been taken from your account
time

Figure 8.2: Interaction using a form

```
<H1 ALIGN="CENTER">Silly form</H1>

  <FORM ACTION="http://localhost/~bob/cpp-print.cgi/" METHOD=GET>
    Name    <INPUT NAME="name" SIZE=64> <P>
    Address <INPUT NAME="address" SIZE=64> <P>
    <INPUT TYPE=SUBMIT VALUE="Send"><P>
  </FORM>
```

the above form is the sort of thing sent to the browser first. This produces a simple screen like:


Bookmarks
Location: http://rabbit.cs.herts.ac.uk/~bob/simple-form.html
Silly form
Name
Jo Bloggs
Address
11 The Avenue
Send

If data is entered and the "Send" button is pressed the browser will generate a GET request and send `name` and `address` values to `localhost`. The requested URI will normally be for an "executable", CGI or PHP, which will run, get the arguments (see later for how it gets them), and send back so response to the client.

8.7 Server side: CGI programs

Very early on in the history of the web additional functionality was added to the server. One of the first simple enhancements was the Common Gateway Interface (CGI) to enable programs to be executed on the server and their output sent back to the browser. The programs are started by the server but not interpreted "inside" the server so they can be written in any language that will execute on the machine that runs the server. CGI programs can be used to access central databases, send back the results of searches, carry out online transactions and many other jobs.

8.7.1 Starting a CGI program

The browser sends a normal HTTP request but the *name* of the requested file is used to decide if it is a CGI program. There are different ways CGI programs are named:

- Historically servers have a special directory called /cgi-bin/ where programs are kept and any request for one of those files results in its execution. So:

  ```
  http://blink.cs.herts.ac.uk/cgi-bin/printenv
  ```

  would result in running the program printenv (if there is one on blink). Normally this server directory is protected from users.

- Some servers enable ordinary users to have cgi-bin sub-directories in their public_html directories. This is not always permitted on some safety conscious systems because CGI programs are regarded as potential security risks. So if allowed:

  ```
  http://blink.cs.herts.ac.uk/~fred/cgi-bin/hello
  ```

  would run user fred's hello CGI program.

- Lastly files in any accessible directory with a name ending in the extension .cgi are treated as CGI programs. Once again this is sometimes not allowed for security reasons. So:

  ```
  http://www.cs.herts.ac.uk/~bill/test.cgi
  ```

  might run the program test.cgi from bill's public_html directory.

8.7.2 "scripts"

On Unix systems there are lots of types of file that can be executed in addition to binary machine code "a.out" files. When a file is "exec-ed" the kernel examines the first line of the file to find the name of a language interpreter, if there is one it is run and given the file to interpret. The format of the first line is `!#` followed by the full path to the interpreter, so:

```
#!/bin/bash
..
```

will cause the shell bash to be executed and given the file of shell commands to interpret.

Very often such programs in interpreted languages are called *scripts*, it is because such "scripts" are often used for CGI that the programs are sometimes called "cgi-bin scripts". There are loads of interpreted languages used on Unix: Unix shell (or bash) command files, PERL, TCL, Awk, Python, and many more.

8.7.3 A small CGI program

The following shell command script hello.cgi will be used as an example:

```
#!/bin/bash
echo "Content-type: text/html"
echo
echo "<H1> Hello </H1>"
echo "<H3> from $SERVER_NAME </h3>"
echo "<p>"
echo "the date and time are: ‘date‘"
```

Notes:

- For now ignore `Content-type:`, that will be discussed soon.
- The `echo` command just writes its arguments to *standard output*; the server must put the connected client socket on the standard output (using `dup` or `dup2`) for the script before it is executed (using `exec`) so that all the standard output will go down the connection to the client.
- Unix shells have variables (and *environment* variables, more later) their value can be accessed by preceding them with a dollar, so

  ```
  $SERVER_NAME
  ```

  is replaced by the value of `SERVER_NAME` which is set by the server to the hostname.
- In a Unix shell script ‘*prog-name*‘ is replaced by the standard output that results from executing the command named: *prog-name*. Amazing! (Well I think so). The command:

  ```
  echo "the date and time are: ‘date‘"
  ```

  will be transformed during execution to:

  ```
  echo "the date and time are: Wed Jan 19 10:30:07 GMT 2000"
  ```

  and then of course written to the standard output.
- for a shell script to be run by the server it must have execute permissions set for all:

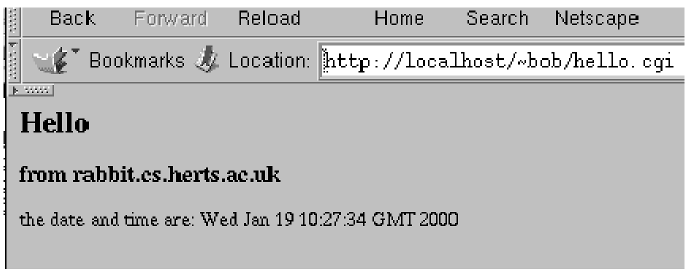
  ```
  chmod a+x hello.cgi
  ```

If, to test your script, you run it directly from your home directory, the output will be:

```
rabbit(2133)$ public_html/hello.cgi
Content-type: text/html

<H1> Hello </H1>
<H3> from  </h3>
<p>
the date and time are: Wed Jan 19 10:30:07 GMT 2000
```

Notice that there was no value for `SERVER_NAME` because it was not executed by a web server. If you invoke it via a browser it might look like:



Hello

from rabbit.cs.herts.ac.uk

the date and time are: Wed Jan 19 10:27:34 GMT 2000

Alternatively you can use binary executable files instead of shell scripts. The following C++ program, cpp-hello.cc, will produce output almost identical to the hello script.

```
#include <iostream.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    cout << "Content-type: text/html\n";
    cout << "\n";
    cout << "<H1> C++ Hello </H1>\n";
    cout << "<H3> from " << getenv("SERVER_NAME") << "</h3>\n";
    cout << "<p>\n";
    cout << "the date and time are: "; cout.flush();
    system("date");
    cout << "\n";
}
```

Note:

- If users' home directories are networked and NFS mounted by different types of machine there can be problems with binary executable files. If you compile the C++ program on a Sun computer but test it by calling a web server on an Intel system it will fail! Wrong binary machine instructions. So make sure that both the system you compile on and the system the server runs on are the same.
- The system library routine `system(..)` causes the named shell command to be executed (by a hidden sub-shell) and the results sent to the standard output.
- The system function `getenv()` returns the string value (actually it's `char *`) of the named environment variable. (more on environment variables next).

8.7.4 The program environment and Environment variables

In the high virtual memory of every process there is a list of pairs of names and values called the *environment variables*. A program can lookup the value of a variable and might then use the value to change its behaviour. The current settings of all environment variables can be examined with the shell command `printenv`, try it.

The variables are used to modify or tailor a user's programming environment. One very important variable is `PATH` which is used by shells (and other programs) to search for executable programs. If the user types `g++ ..` to a shell prompt the shell will use the value of `PATH` to look for `g++`. This is necessary because there are many directories that hold programs. A typical value might be:

```
rabbit(2121)$ echo $PATH
/usr/local/bin:/usr/X11R6/bin:/bin:/usr/bin:/usr/local/java/bin:.
```

Note that this `PATH` contains "`.`" which means the shell will look in the current directory. Some systems don't, by default, have ".", you must add it to your own start-up dot files. Environment variables can be set in bash by using `export`:

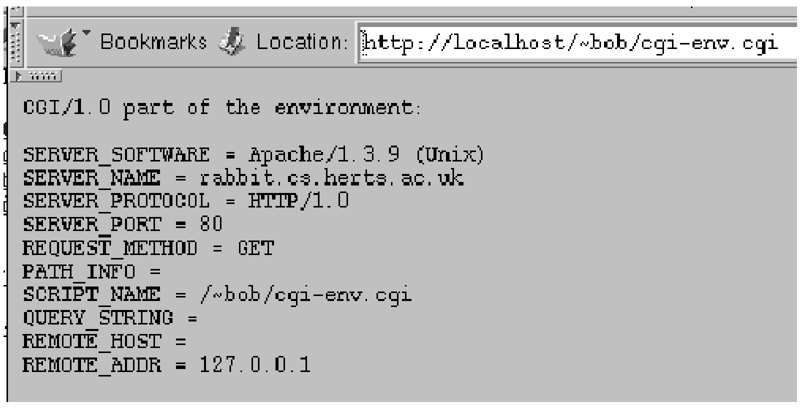
```
export PATH=~/bin:$PATH
```

will prefix the bin directory in your home directory to the current value of `PATH` and then re-assign to `PATH`. Environment variables are automatically "inherited" from the parent process whenever a new process is started. So environment variables usually only need to be set once during login, they are then passed automatically to every program run thereafter. Users normally use the file `.bash_profile` or `.profile` to set their environment. However if necessary a program can add or change environment variables after `fork` but before `exec` using the system library routine `putenv` so that the environment of the new process will be different, or to pass extra information to it.

Web servers must set certain environment variables for CGI programs. Here is a little CGI program that prints out some of the environment variables set by the server:

```
#!/bin/sh
echo Content-type: text/plain
echo
echo CGI/1.0 part of the environment:
echo
echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo SCRIPT_NAME = "$SCRIPT_NAME"
echo QUERY_STRING = "$QUERY_STRING"
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
```

And its output in a browser:



```
CGI/1.0 part of the environment:

SERVER_SOFTWARE = Apache/1.3.9 (Unix)
SERVER_NAME = rabbit.cs.herts.ac.uk
SERVER_PROTOCOL = HTTP/1.0
SERVER_PORT = 80
REQUEST_METHOD = GET
PATH_INFO =
SCRIPT_NAME = /~bob/cgi-env.cgi
QUERY_STRING =
REMOTE_HOST =
REMOTE_ADDR = 127.0.0.1
```

Notice that this program doesn't send HTML and therefore was considerate enough to tell the browser by sending `Content-type: text/plain` and not `text/html`.

8.7.5 How CGI programs are executed

When a server receives a request and determines that it is for a CGI program it must:

- `fork` to produce a child process (it may already have done this to deal with the request if it is a simple concurrent server, if so it doesn't need to do it again).
- Check what sort of HTTP request it is. It might be the GET or the POST method (for the CS2 coursework assume it can only be GET, say "not implemented" otherwise).
- It then prepares the environment by setting special environment variables, eg:

  ```
  putenv("SERVER_SOFTWARE=MyServer version 0.1");
  ```

  or if a value is in a variable read from the connection:

  ```
  char env_str[64];
  sprintf(env_str,"REQUEST_URI=%s",file); putenv(env_str);
  ```

- Send the correct HTTP response down the new socket to the client. Eg:

  ```
  HTTP/1.0 200 OK
  Date: Wed, 19 Jan 2000 13:13:44 GMT
  Server: MyServer version 0.1
  ```

  NB it is the job of the server to send the response line and maybe a couple of MIME lines. But it doesn't send the vital `Content-type:` and blank line, it can't, it doesn't know what content will be generated by the CGI program. These lines *must* be sent by the CGI program immediately it starts, that's why all the scripts start with:

  ```
  echo "Content-type: text/html"
  echo
  ```

- "re-plumb" the input and output for the CGI program. This will involve closing and duplicating file descriptors. At the very least put the new socket on the standard output, `dup2(newsock,1)`.
- Finally `exec` the requested program.

8.7.6 CGI input, forms, GET and POST

It is important and useful for input or arguments to be passed from the client to the program. This is solved by providing extra data from the client at the end of the URI. Here is an example of the type of URI generated for a search engine request:

```
http://www.altavista.com/cgi-bin/query?pg=q&what=web&q=j+s+bach
```

- The proper URI is terminated by "?",
- The actual path sent in the GET request will not have the hostname etc., it is just:

  ```
  cgi-bin/query?pg=q&what=web&q=j+s+bach
  ```

- The query consists of name value pairs: `pg=q`, `what=web` and `q=j+s+bach`, The pairs are separated by “&”.
- Spaces have been replaced by “+”.

The query string is split from the program file name by the server and given to the program via an environment variable: `QUERY_STRING`. There are numerous packages and library functions available for CGI programs to carry out the separation of all the name value pairs and the re-replacement of “+” by spaces.

HOW FORMS GENERATE THE GET QUERY STRING
Because it is so complicated to formulate the query strings in the client there is a facility in HTML to get input from the user and send it to a remote CGI program, it is the `<form>..</form>`. Here is an example of a very simple HTML file with a form in it:

```
<H1 ALIGN="CENTER">Silly form</H1>
  <FORM ACTION="http://localhost/~bob/cpp-print.cgi/" METHOD=GET>
    Name    <INPUT NAME="name" SIZE=64> <P>
    Address <INPUT NAME="address" SIZE=64> <P>
    <INPUT TYPE=SUBMIT VALUE="Send"><P>
  </FORM>
```

this is the same form as used in section 8.6. If data is entered and the “Send” button is pressed the browser will generate the following URI query string:

```
/~bob/cpp-print.cgi/?name=Jo+Bloggs&address=11+The+Avenue
```

and send it in a GET command to `localhost`.

HOW FORMS SEND DATA WITH POST
An alternative way to send data to a CGI, ASP or PHP program is to use the `POST` in HTTP, this is similar to `GET` but is normally only used to invoke executable pages and send them data. The POST does not encode the data as an extension to the URI but rather it sends it in the body of the request. It can be used to send larger quantities of more complicated data. So if the previous little form was changed to:

```
<H1 ALIGN="CENTER">Silly form</H1>
  <FORM ACTION="http://localhost/~bob/cpp-print.cgi/" METHOD=POST>
    Name    <INPUT NAME="name" SIZE=64> <P>
    ...
```

everything else is the same but the `METHOD` attribute has been changed to `POST`. If this is filled in and then sent by a browser the HTTP request might look like this:

```
  POST /cpp-print.cgi HTTP/1.1
  Host: localhost
  User-Agent: Mozilla/5.0 Gecko/20030624 Netscape/7.1
  Accept: text/xml,application/xml,...
  ...
  Connection: keep-alive
  Referer: http://localhost/~bob/fp.html
  Content-Type: application/x-www-form-urlencoded
  Content-Length: 41

  name=Tony+Blair&address=10+Downing+Street
```

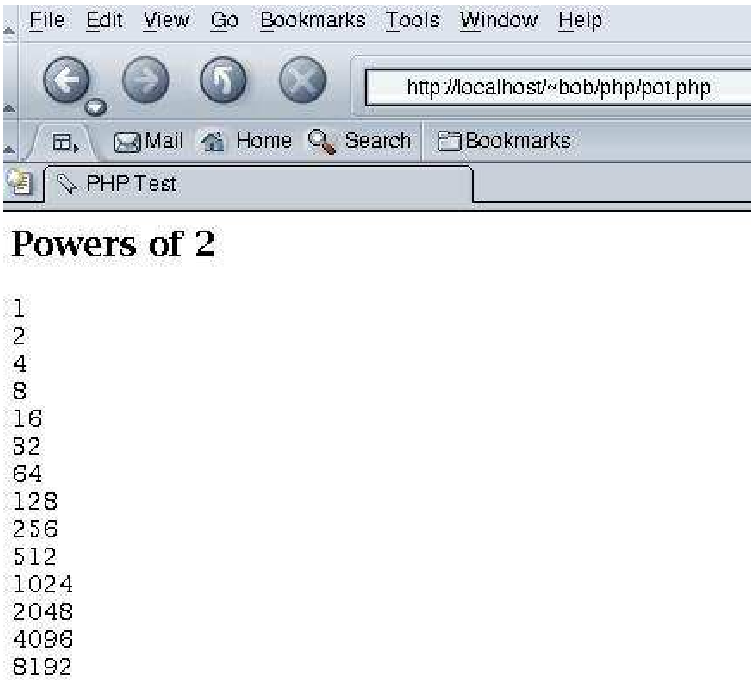
the CGI, PHP or JSP program must know how the data is sent, or check the method used.

8.8 Server side: PHP

PHP is a programming language, it looks a bit like C (as do many programming languages), it has dynamic typing (a variable can hold any type, the type is checked at runtime). What makes it different is that is is designed to be embedded in HTML files (pages). The PHP interpreter processes the file, any HTML is sent to standard output (connected by the web server to the client browser), any PHP is executed. Here is a simple example:

```
<html>
 <head>  <title>PHP Test</title> </head>
 <body>
   <h2> Powers of 2 </h2>
   <p>
   <?php
      $pot = 1;
      while($pot < 10000) {
         print("   $pot <br>\n");
         $pot = $pot * 2;
      }
   ?>
 </body>
</html>
```

and here is the output when it is requested from a browser:


File Edit View Go Bookmarks Tools Window Help
http://localhost/~bob/php/pot.php
Mail Home Search Bookmarks
PHP Test

Powers of 2

```
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
```

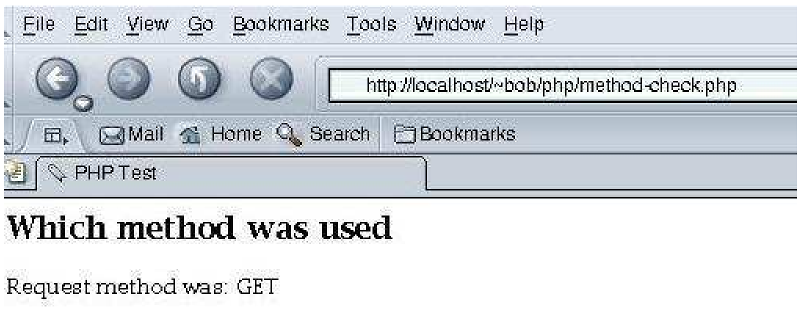
Note that:

- a PHP file is basically HTML with bits of code in the middle,
- PHP code is surrounded by:
  ```
  <?php
     ...
  ?>
  ```
- variable names are preceded by `$`, and they don't need to be declared,
- the output of the print statement goes down the connection to the client with the surrounding HTML.

Here is another example, this one examines an element in a pre-defined array. When PHP programs are executed many special values are set, this one is the type of the HTTP request, either `GET` or `POST`. Further note that PHP arrays can be indexed by numbers or by strings (this type of array is sometimes called an associative array).

```
<html>
 <head>  <title>PHP Test</title> </head>
 <body>
   <h2> Which method was used </h2>
   <p>
   <?php
        $rm = $_SERVER["REQUEST_METHOD"];
        if( $rm ) {
            print("Request method was: $rm <br>\n" );
        } else {
            print("REQUEST_METHOD not set <br>\n" );
        }
   ?>
 </body>
</html>
```

and here is the output when it is requested from a browser:


File Edit View Go Bookmarks Tools Window Help
http://localhost/~bob/php/method-check.php
Mail Home Search Bookmarks
PHP Test

Which method was used

Request method was: GET

The PHP interpreter can be run outside the web server. It can be a good way to debug programs. Also, in this case, it shows the HTML being sent to the standard output which will normally be the browser connection, but here is the console.

```
sally(309)$ php4 method-check.php
X-Powered-By: PHP/4.1.2
Content-type: text/html

<html>
 <head>  <title>PHP Test</title> </head>
 <body>
   <h2> Which method was used </h2>
   <p>
   REQUEST_METHOD not set <br>
 </body>
</html>
sally(310)$
```

8.9 Client side (browser) services

Client side web facilities are sent from the server but they are executed or interpreted in the browser.

Javascript which is a language that can be embedded in HTML code between `<script>` and `</script>`. Javascript source code is interpreted by the browser. The language has no existence outside HTML. It is usually used to add checking or animation to an HTML file. All attributes of the currently displayed HTML: links, images, colours etc., are accessible from Javascript making it a very powerful tool for manipulating pages.

browser plugins these vary from movie players that are run when a video is downloaded, to complicated interpreters for animations like flash that are integrated into the display. In fact Java is implemented using a Java byte code interpreter plugin.

Java Java is a complete programming language, it exists outside browsers and HTML. However most browser have a built-in interpreter for the byte-code form of Java. Java is less closely integrated into HTML and the browser however it is musch more general purpose language than Javascript making it better for more complicated applications.

8.9.1 Javascript example

Apart from making the page display more interesting client side services can reduce network traffic. The following Javascript example checks the values entered into a form, this can reduce the need for a server to check and send back an error page from the server. Here is a form with Javascipt checking code:

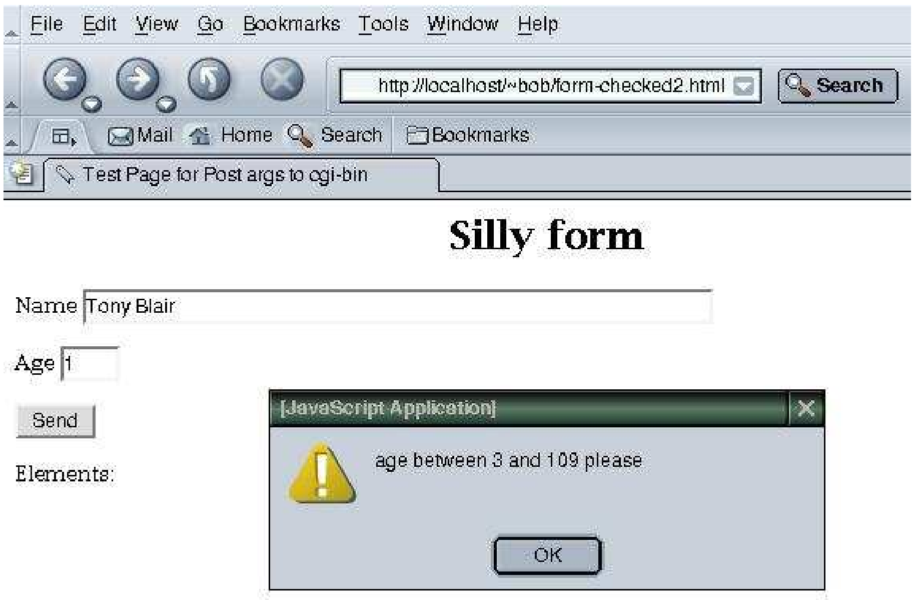
```
<!DOCTYPE html  PUBLIC "-//W3C//DTD HTML 4.0 transitional//EN">
<HTML>
 <HEAD>
  <TITLE>Test Page for Post args to cgi-bin</TITLE>
  <SCRIPT LANGUAGE = "JavaScript">

    function checkage() {
      var a;
      a = parseInt(document.okform.age.value);
```

```
        if (a<=2 || a>=110) {
           window.alert("age between 3 and 109 please");
           document.okform.name.value = "";
           document.okform.age.value = "";
           return false;
        } else {
           return true;
        }
     }
   </SCRIPT>
   </HEAD>
   <BODY>
    <H1 ALIGN="CENTER">Silly form</H1>

    <FORM NAME="okform" ONSUBMIT="return checkage()"
          ACTION="http://localhost/~bob/show-env.cgi" METHOD=POST>
       Name <INPUT TYPE="text" NAME="name" SIZE=64> <P>
       Age  <INPUT TYPE="text" NAME="age" SIZE=4> <P>
       <INPUT TYPE=SUBMIT VALUE="Send"><P>
    </FORM>
   </BODY>
  </HTML>
```

This is the output if the form is loaded into a browser, given unsuitable input and then the “send” button is pressed:


File Edit View Go Bookmarks Tools Window Help
http://localhost/~bob/form-checked2.html
Search
Mail Home Search Bookmarks
Test Page for Post args to cgi-bin

Silly form

Name Tony Blair

Age 1

Send

Elements:

[JavaScript Application]

age between 3 and 109 please

OK

Chapter 9

The Domain Name Service DNS

9.1 Domain names

The DNS (Domain Name Service) maps host names to addresses. At the level of TCP/IP connections on the Internet all addresses are the (IPV4) 32 numbers, there are no host names, the names are provided by the DNS. Once upon a time very large central tables were kept on the network, but now this has become impossible due to their size and rapidity of change. Now the Internet uses a protocol between systems called the DNS which queries remote systems about how to map a name to a number.

Names are read left-to-right from smallest domain (or unit) to widest: `slink.feis.herts.ac.uk` is a system, `slink` in the domain administered by the our faculty, `.feis.`, in the campus network domain administered by the University of Hertfordshire, `herts` in the UK academic community, `ac.uk`. Now although the University has a class B address, there is no structure, correspondence or mapping between parts of it and the `ac.uk` bit of the name.

Usually there is a domain for every separate *autonomous system* or network administrative authority, ie. 147.197 (a B address) is herts.ac.uk. But above that level the domains have a structure not related to IP addresses. The actual domains have grown up over time and the "top-level" domains are countries or the US names: `.com`, `.edu`, `.org` etc. Figure 9.1 is a picture of part of the domain name hierarchy.

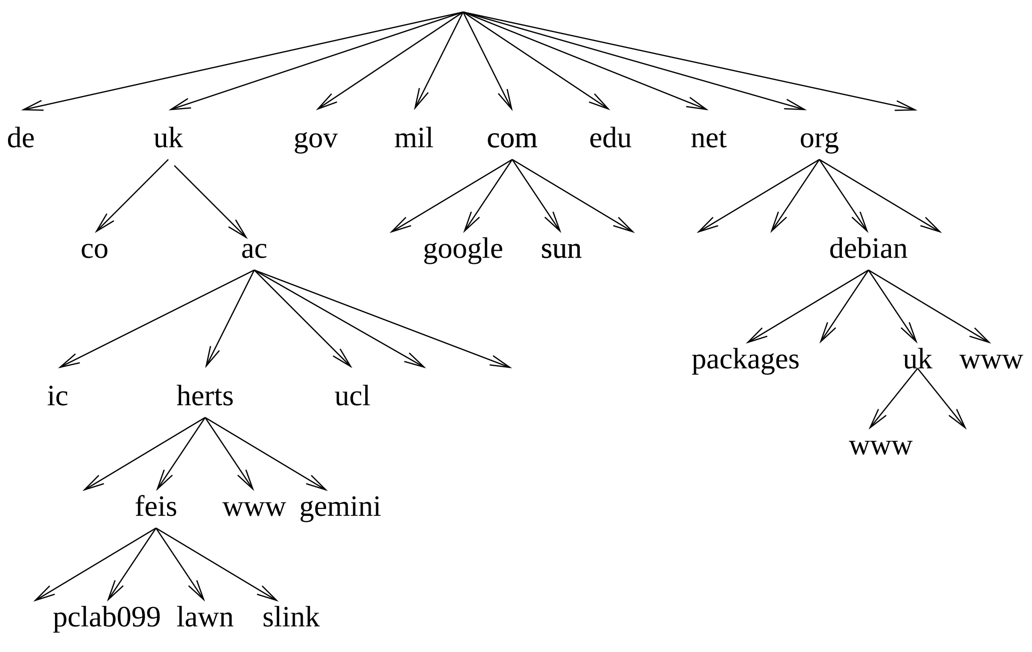


Figure 9.1: Domain name hierarchy

9.2 Zones and name servers

The hierarchy is divided into *zones*, each zone belongs to some administrative authority, either a company, university or network organisation. A zone is responsible for:

- allocating names and numbers to systems that belong in the zone, or pointing (delegating) to the name servers in sub-zones,
- maintaining two or more *name servers* to translate name requests to addresses of systems or of name servers for sub-zones.

This organisation can cope with the dynamic distributed nature of the network structure, the responsibility for translating names is passed down to the groups who allocate names and numbers to systems.

In order to enable end user zones to be found various network organisations provide intermediate zones, at the “top” there are about 20 name servers that know about how to find the next level name servers, `.com`, `.uk` etc. A zone doesn’t always correspond to one domain name level. It is possible for one zone to have two or more levels of name hierarchy supported by its name servers. In picture 9.2 there is one zone to

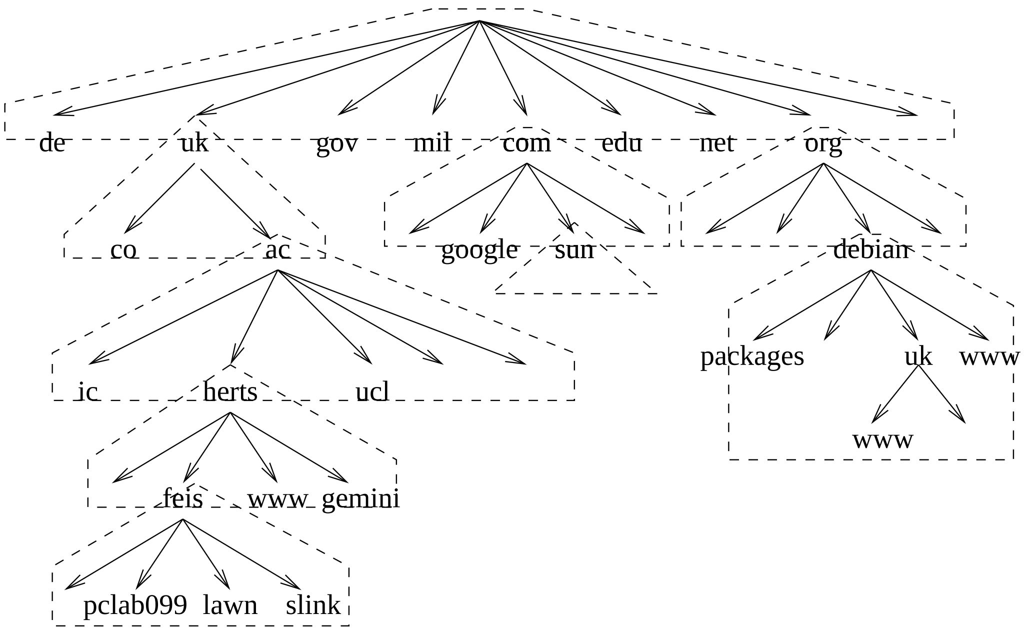


Figure 9.2: DNS zones

manage all the levels of the `debian` hierarchy.

The name servers in each zone hold a table mapping host names to numbers, or sub-domain names to their name servers. The responsibility of a name server is to deal with requests from two sources:

- local applications in that zone that need to begin resolve a local or remote name, the name server must, if necessary, contact other name servers on their behalf, or
- other name servers that need to find out about the names in the name servers domain.

9.3 Resolving a name

Every system connected to the internet has address(es) of one or more local name servers and software libraries to contact this server if any program wants to *resolve* (translate) a name. The server then deals with the request. There are alternative programs to provide the DNS but the basic operation of all is probably:

- if it is a local name in this system’s zone, lookup the table and return the number,
- search the cache to see if it has been recently requested and saved,
- contact a “top-level” server (all DNS programs know these numbers), and ask for the name,
- the top-level server will probably not know the full answer but it will know somebody who does know, in other words it will match the rightmost part of the domain name and provide the address of the name server for the next zone,
- the original name server then sends the same query to this next name server, and either get the answer or another name server address,
- this continues until it either fails or gets the answer.

For example consider the picture 9.3.

- Some system on the internet has an application that asks its local name server for the address of `slink.feis.herts.ac.uk`.

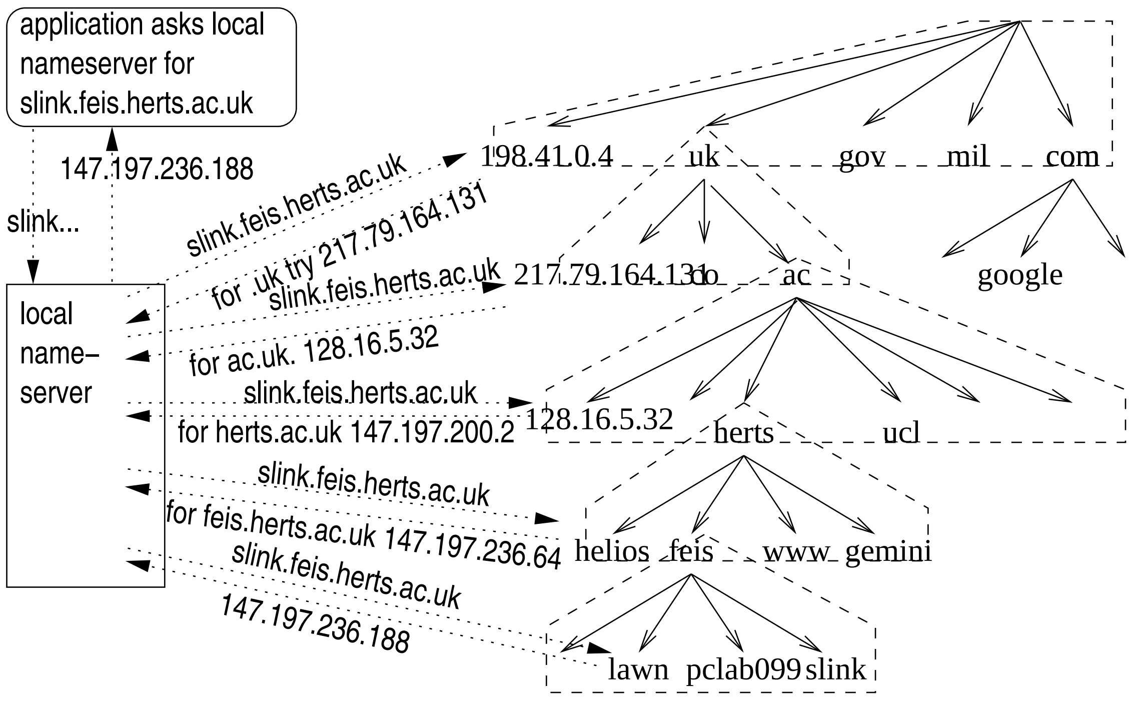


Figure 9.3: DNS query

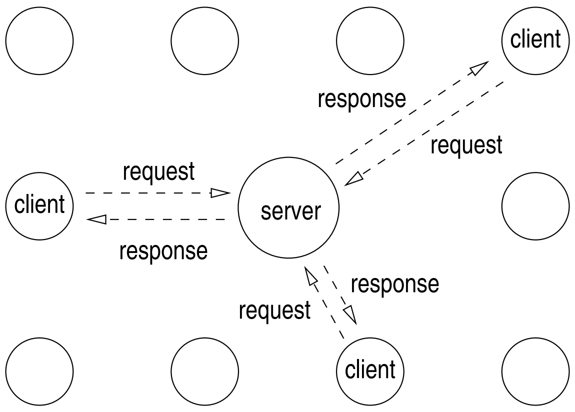
- it isn't a local name and the name is not cached so
- the name server contacts a top-level server, in this case `198.41.0.4`. The top-level server knows the zone servers for `.uk` so returns one of the addresses `217.79.164.131`
- the local name server then sends the full request to `217.79.164.131` which doesn't know the answer but does know the name servers for `.ac.uk` one of which is: `128.16.5.32`,
- the local name server again sends the full name and gets the address of `helios` on our campus, `147.197.200.2`,
- it contacts `helios` which returns the address of the server for feis.herts.ac.uk, this is `lawn` in computer science and its address is `147.197.236.64`,
- the poor tired local server then sends its request again, this time to `lawn`, now `lawn` does know the answer, it is in its zone. It replies with `147.197.236.188`, the number for `slink.feis.herts.ac.uk`
- the server passes this address to the program that asked, (it then collapses from exhaustion).

Chapter 10

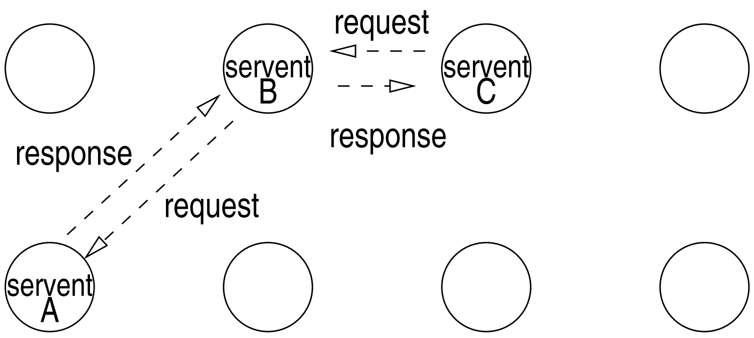
Peer to peer networks

10.1 Application architecture

There are two contrasting network application architectures: *client-server* and *peer-to-peer*. The definition of what actually constutures peer-to-peer can be a bit unclear. The important characteristic seems to be that a in client-server the client system always initiates the interact by sending a request, the server accepts the connection and sends a response:



With the peer-to-peer architecture any system can initiate requests or act as a server and receive requests:



in the above picture each participant is called a *servent*[1], and servent B is acting as a server for servent C, receiving a request and sending a response, but also behaving as a client and sending a request to servent A.

The definition concerns the way the parts of a network application interact, the nature of their protocol, it is not necessarily about how the user perceives the system. It is possible to have a person-to-person system such as a network message exchange where each participant seems both to send and receive messages, however the program implementation could involve a central server that routes the messages, the client programs initiate the connections to the central server, they don't receive incoming requests.

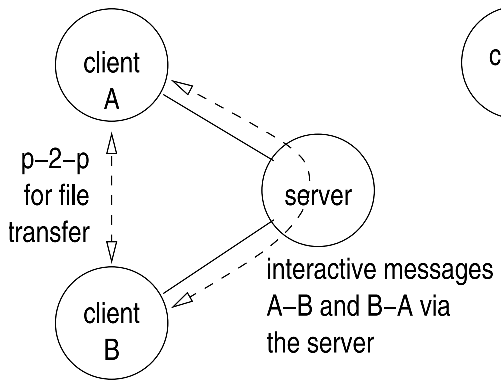
In addition the difference between client-server and peer-to-peer is not anything to do with the underlying network operation or topology below the application layer where all systems can be considered to to uniformly connected and all can open or receive connections.

10.2 Instant message systems

These are systems that allow people to hold remote conversations with each other using typed text messages, example are Micros**t Messenger, ICQ (bought out by AOL), AIM from AOL, Yahoo Messenger, and the open standard Jabber. In some ways most of these are not fully peer-to-peer systems as suggested above. However some have more peer-to-peer features than others. In most the conversations between client

[1]"servent" is a term used in the Gnutella file sharing system, the word seems to be a mixture of "server" and "client".

programs go through special purpose central servers but will support direct client to client connections for file transfers or video links.



Possible reasons for using a central server might be that:

- if extra clients (people) can be invited to join a conversation then the required number of inter-client links would rise very fast if peer-to-peer connections were used,
- there is less need to avoid legal attacks on a central system than with file sharing systems (see later),
- the actual data passing through the central server is not very high
- a central server is essential for notifying other when a new user logs in.

10.3 File sharing

These systems are quite recent but have spread and evolved quite fast. They enable users to search for and download files (usually music or film files) from other users' systems on a network. Examples are (or have been, because with fast evolution there seem to quite a few deaths): Napster, Gnutella, Freenet, Audiogalaxy and the Fastrack-Kazaa-(old)Morpheus family.

The earliest widely used system was Napster, it was used to provide access to mp3 music files. How it operated:

- a client program would login to one of several central servers and upload a list of files the client was prepared to make available,
- when a user wanted to search for a file they would send the search request to the central server and receive a list of client machine addresses,
- the user would choose one of the systems and the client program would download directly from the other system.

Initially because files were transfered from one individual to another it was hoped it would avoid copyright laws however the American music industry paid enough lawyers enough money for long enough that eventually the Napster site was forced to close. This encouraged more decentralisation in peer-to-peer application design, newer systems do not have a central server with a list of all available files, the search became peer-to-peer aswell as the file transfers. With a reduced role (or no role at all) for a central server it is hoped that the systems are less vulnerable to attack by lawyers.

10.4 Gnutella

Gnutella is an application network sitting on the internet, it has a continually changing topology as systems are turned on and join or are disconnected, in addition it seems to generate a lot of traffic. Each active node (servent, client or wahtever it is called) tries to maintain a small number of open TCP connections to other nodes, usually between 3 and 10, this produces the network structure, if connections break (systems turned off) a node establish new connections. There is no central server, and at the moment, no login procedure.

10.4.1 Distributed search

This section describes just the distributed search and file transfer, how the connections are found, set up and maintained will be summarised afterwards. So to search:

- a node transmits a search request to all its connected neighbours (3–10), the search request has a unique number, it also has TTL (time to live) count,

- the neighbours propogate or forward the message, each will:
  - record the unique message number in a table with the address from which it was received,
  - decrement the TTL count, and if it is not zero…
  - pass the request on to all their neighbours (except on the link they received it on),

  Note that if the same search request is received on another connection, which is highly likely because of the tangled, arbitrary structure of the net, it can easily be discarded because the search request's unique number has been recorded in the table.

- each node that receives the search request also performs the search on its files, and forms a search response with a variable length list of files satisfying the search. The response will include the search request's unique identifier and also the address of node forming the reply. The response will be sent back only on the connection from which it was received,

- any intermediate node will, in addition to forming its own search response, receive responses from other systems it propogated the original search to. It will then forward these responses back to the originator by using the unique number to look up its table to see which connection it got the original request on.

- When the responses arrive back at the initiator they will be shown to the user who will select which one to fetch. The file transfer uses the HTTP protocol's GET request; each node program contains its own code to act as a little HTTP server and client to deal with the file transfers. The HTTP connection will be a single new direct connection to the selected file's node–no viral propogation this time; this is possible because the necessary IP address was included in the search response.

10.4.2 Finding and maintaining connections

There is an unsettled question: how does a servent (node) get its connections? There is a special message: "GNUTELLA CONNECT" that is sent to any other existing node that can be accepted "GNUTELLA OK" or rejected. But how does a new node known what system to send this to? There has to be a handful of "well-known addresses" of systems that are always running and connected. These are the initial contact points. In some sense these are like special servers although there role is very limited; that is the problem of a very distributed system–how to contact it. So some "server" is still needed until some efficient broadcast method can be devised.

The whole problem is not solved, the new node only has one connection, where does it get the others? There is a special message called "PING" (not the ICMP ping) which works like a contentless search request, it:

- has a unique number
- has a TTL field
- is propogated like a search request, every node recording its incoming connection and number it the table,

The use is that recipients respond to it with "PONG" replies. A PONG reply contains:

- the unique number of the PING it's replying to,
- the IP address of the node that is replying, and
- the number, and total size of offered files on the replying system

These PONG messages get returned to the iniiator just like search replies. When PONGs get back the system that started the PING it will have loads of IP addresses, it can then use these to try to open connections using "GNUTELLA CONNECT".

Additionally PINGs can be sent out later to get more IP addresses if nodes that are used for connections are turned off.

10.4.3 Summary of protocol

- to open a new TCP connection there is the GNUTELLA CONNECT message, these is a before the real protocol can be used,
- once a connection is open fixed format binary messages can be sent, these constitute the real protocol. They all have a unique number, a TTL, a length field and a message type. The message types are:
  - PING, to discover more addresses, they are propogated,
  - PONG, the reply to PING containing the reponders IP address,
  - SEARCH, containing a file search string, propogated like PING,
  - SEARCH REPLY, that contains the names of files, and machine address, from each node responding to the search,
  - PUSH, used to start data transfers from systems that are behind firewalls, necessary but not a major part of the operation.

  These constitute the messages sent along the TCP connections.
- Lastly there are HTTP GET request and replies that will be sent directly between systems to fetch files once they have been found.

10.4.4 Issues in Gnutella

- It is very decentralised, it is very robust, connections and nodes come and go but the network is always there,
- it is an open published protocol and there are many client programs (servents) available,
- it is more secure against attacks from lawyers, the lack of a permanent central server containing all the search functions means it is harder to find anybody to take to court,
- at the moment it doesn't contain much internal security, anybody can connect (good) but anybody could write programs that flood the system with corrupt searches or pings (bad),
- additionally this basic version of Gnutella might not scale up very well as the number of users increases the traffic they produce rises exponentially. Each search spreads across the net like a virus (until the TTL gets to zero). Also each machine that runs a Gnutella client (servent) program is going to be used by other systems to search and pass on searches; you run the program, sit back, do nothing, but your machine and network connection are immediately very busy.
- there are already some improvements and suggestions for improvements in the protocol that might reduce the load on the internet,
- it is a very new idea and there is not yet enough experience to know exactly how things like this will evolve.

There are a couple or links for further information:
http://www.gnutelliums.com/, http://www.limewire.com/,
http://www.rixsoft.com/Knowbuddy/gnutellafaq.html,
http://www.gnutelladev.com/protocol/gnutella-protocol.html,
and the current home of the standard:
http://rfc-gnutella.sourceforge.net/.

Chapter 11

Network security

11.1 Some cryptographic concepts

A very important component in any secure system will be some form of *encryption*, the use of a *key* to "mangle" a message so that nobody else can read it except somebody else having a suitable decoding key. There are many different encryption schemes and algorithms with very different properties. The following brief notes summarise three schemes (no details of the actual algorithm, I'm not a mathematican).

11.1.1 Secret key encryption

This scheme uses one algorithm and key that can both encode and decode a message. So if Alice wants to send a message to Bob, she encrypts the message:

$$E = encrypt(K, M)$$

where M is the "plain-text" message, K is the key, E is the encrypted message, and *encrypt* is the secret key encryption algorithm, for example DES. The only way the message can be decrypted is with the same key K, Bob has the key aswell so he does:

$$M = decrypt(K, E)$$

and can read the message. Nobody else can read it, unless they know the secret key. Features of secret key:

- quite efficient and fast, can encode streams of data,
- has the problem of *key distribution*, how do you pass secret keys around safely?

11.1.2 Public/private key encryption

This scheme generates a complementary pair of keys, called the *public key* and the *private key*, with the property that anything encrypted with the private key can only be decrypted using the matching public key and vice versa. One of the most famous algorithms is RSA.

Public private key pairs belong to individuals, and they will publish, or make available, their public key but hide their private key.

$$E = encrypt(K_{priv}, M)$$

where M is the "plain-text" message, K_{priv} is the key, E is the encrypted message, to decrypt: aswell so he does:

$$M = decrypt(K_{pub}, E)$$

Also the converse holds:

$$M = decrypt(K_{priv}, encrypt(K_{pub}, M))$$

How can it be used? Firstly if Alice wants to send a message to Bob that only he will be able to read she encodes it using Bob's public key knowing that nobody but Bob (the owner of the matching private key) will be able to decode it. So Alice does:

$$E = encrypt(K_{pub-bob}, M)$$

and sends it to Bob, he decodes it:

$$M = decrypt(K_{priv-bob}, E)$$

Alternatively Alice might want to send a message to Bob in such a way that he will know she is the only one that could have sent it, this is *message authentication*. Also she will not be able to deny that she sent it, this is *non-repudiation*. (These are only the case so long as her private key is not disclosed.) So she will encrypt it with her private key:

$$E = encrypt(K_{priv-alice}, M)$$

and Bob (or anybody else) will be able to decode it:

$$M = decrypt(K_{pub-alice}, E)$$

The 2 can be put together. Alice will encrypt with her private key and then encrypt the result with Bob's public key:

$$E = encrypt(K_{pub-bob}, encrypt(K_{priv-alice}, M))$$

so that only Bob can decode it. Secret and authenticated.

Features of secret key:

- quite inefficient and slow, can only encode small amounts of data,
- provides a solution to the problem of *key distribution*,
- there still remains the problem of knowing that the person who claims to own a public key really does own it.

11.1.3 Message digests

A *message digest* is a a special *hash code* formed from a message, a sort of cryptographic checksum. One widely used digest algorithm is MD5. If:

$$D = MD(M)$$

where M is the message, the document, the file, MD is a message digest function and D is the computed message digest hash code. The digest D is usually at least 128 bits long, it is not possible to infer anything about M from D, it is almost impossible that any other document M' will produce the same D, any change to M, however small, will change D. You could almost say it is a unique fingerprint.

One use of message digest is to reassure users of the safety and authenticity of files and programs that are being distributed. If the file distributor, Alice, has a file F to distribute they calculate the digest D and "sign" it using their private key producing ED which they put on the server along with F.

$$ED = encrypt(K_{priv-alice}, MD(F)$$

Now Bob wants to download the program F and be confident nobody has altered it or added a virus, so he dowmloads F and ED. He first computes the D of F using the same algorithm MD, then decrypts ED using Alice's public key, and finally compares them.

$$MD(F) = decrypt(K_{pub-alice}, ED)$$

If they are the same he knows nobody has tampered with F since Alice calculate D, and nobody but Alice could have done it.

11.1.4 Certificates

There is a remaining problem: how to you know that a public key belongs to the person who presents it? The solution is to use a "well known authority" to verify that a public key belongs to a specific person. It uses a *certificate*. If Bob wants a certificate he:

- goes to a well known authority (there are many, including companies like Verisign)
- proves who he is using an ID card, a driving license or something else,
- has a public-private key pair generated for him
- pays some money, and receives a certificate consisting of his public key and a statement of his identity (name, email, address etc.) all hashed and signed with the private key of the authenticating company (the "well known authority").

Then Alice (or anybody else) can verify his public key belongs to him, they compute the hash key, and compare it with the "signature" decoded with the public key of the authenticator.

11.1.5 SSL

There are many protocols and applications of encryption, PEM and PGP can be used to encrypt e-mail, IPSec encrypts IP network connections, Kerberos deals with user authentication, and many others. One of the best known protocols is SSL (and its newer standardised version TLS), it is used for authenticating and encrypting program to program (transport) connections. It is nearly always used by Web servers that require a credit card number to be submitted.

The server system (being run by Bob) has its own certificate (yes computers can have certificates). Alice wants to buy a Linux palm computer from his site so she will initiate an HTTPS connection (one using SSL):

browser	message	server
$\rightarrow$	algo. preferences + R_c	$\rightarrow$
$\leftarrow$	algo. choice + R_s	$\leftarrow$ server chooses algorithm
check certificate $\leftarrow$	server certificate	$\leftarrow$
$\leftarrow$	request client cert. or done	$\leftarrow$
assume no req. $\rightarrow$	$encrypt(K_{pub-serv}, SK')$	$\rightarrow$
$SK = f(SK', R_c, R_s)$		$SK = f(SK', R_c, R_s)$
$\rightarrow$	use encryption with SK	$\rightarrow$
$\rightarrow$	done SSL handshake	$\rightarrow$
$\leftarrow$	acknowledge done SSL	$\leftarrow$
	exchange data encypted with SK	

- the client sends initial request and suggests some encryption preferences, also a random number R_c, the random number is used later,
- server responds with a choice from encryption preferences, and its random number R_s
- server sends certificate which is checked by the client, if the server wants the client to authenticate itself using its certificate it asks for it now, the process will be similar, otherwise it says “done” so they can move on to the next step,
- client sends a value to be used as a secret key (stage 1) for encrypting the whole session after the handshake is complete. This is encrypted with the server’s public key.
- now both ends can compute the final secret session key based on the random numbers exchanged earlier and the stage 1 session key sent by the client,
- client says switch to using session key, server acknowledges,
- all the transaction messages encrypted using the symmetric secret key just generated.

11.2 System security without networking

Without networking the problem of policing an operating system is relatively simple. If users can only access the system through local terminals then they are easier to physically protect (no link tapping). Users can *only* access the system through terminals (no network servers accepting connections from elsewhere), so good password security can stop unauthorised users. The main problems arise from enforcing the different access policies and authorisation within the system (often called “protection” in opsy textbooks).

11.3 System security with networking

With networking there are thousands of ways in.

- Use of stolen or unprotected user accounts via `telnet` and similar programs,
- At the data-link layer, for example Ethernet, packets can be observed and examined by any system attached to the Ethernet. These are called *packet sniffers*. Passwords, credit card numbers or confidential data are stolen.
- At the network layer people can install false routing systems to intercept and even change packets. This can be done by masquerading as DNS servers.
- Systems can be flooded with traffic at the application or the transport layer causing services to fail. These are “denial of service attacks”.
- At the application layer there are many types of attack.
  - CGI programs on WWW servers are often insecure,
  - network filesystems (NFS, SMB etc.) can be very insecure,
  - many server programs have known vulnerabilities that allow intruders in,

11.4 How can networking be more secure?

- Install *audit* programs so that attacks can be detected (and sometimes) repaired. They usually work by recording the state of important files and checking for unexpected changes,
- Use better authentication for passwords and remove old or unused accounts,
- Many systems have network servers that are not used or are badly configured: remove any unused services,
- check that all local network fileservers are secure (don't permit setuid programs from insecure file systems),
- Use authenticated and encrypted network connections, this means that the only people making or receiving connections to or from your systems are ones that can be *authenticated* and afterwards you are safe from sniffer attacks because of *encryption*.
- Use firewalls to filter and monitor all network traffic entering and leaving a local network. A firewall is a system between a local network and the rest of the Internet that can monitor all packet traffic. It can recognize attacks and reject packets.
- read regular network security reports about newly discovered weaknesses in any server programs you use and get new, fixed versions.

11.5 Firewalls, Proxies, and Masquerading

- Many related solutions depend on a "box" between the network to be protected and the rest of the internet.
- The "box" provides more functions than a simple gateway or router, it must provide some privacy or prevent some of the forms of attack from the outside,
- The sorts of protection it can give are:
  - to hide services and make it harder for port scanners,
  - to prevent some of datagram fragment attacks,
  - to prevent incorrect source address spoofing,
  - to hide machine and their identities
  - to prevent ICMP flooding,
- Sometimes fancy routers also provide firewall functions, sometimes they are separated.
- Very often firewalls are used to monitor and restrict outgoing security so that employers and owners of networks can spy on, or control what their employees or users are doing.

11.6 Position of firewall

```
                 __________     +--- ...
  _/\__/\_      | PPP gate/|    |     _______________
 |        |     | Firewall |  (LAN)  |               |
/ Internet \----|  System  |--(HUB)--| Workstation   |
\_  _  _  _/    |__________|         |_______________|
  \/ \/ \/                      |     _______________
                                |    |               |
                                +----| Workstation   |
                                     |_______________|
```

- Here is a simple ISDN, cable modem or phone line linking a small network to the internet.
- I've got one at home,

```
               _________         __________
  _/\__/\_    | Router  |       |          |        ____________
 |        |   |   or    | (DMZ) | Firewall | (LAN) |            |
/ Internet \--|Cable Mdm|-(HUB)-|  System  |-(HUB)-|Workstations|
\_  _  _  _/  |_________|   |   |__________|       |____________|
  \/ \/ \/                  |
                        (Outside)
                        (Server)
```

- Here is a more complicated system with a special router
- there is a separate firewall to do packet filtering
- this is suitable for a large net with legal addresses.

11.7 Encrypting network connections

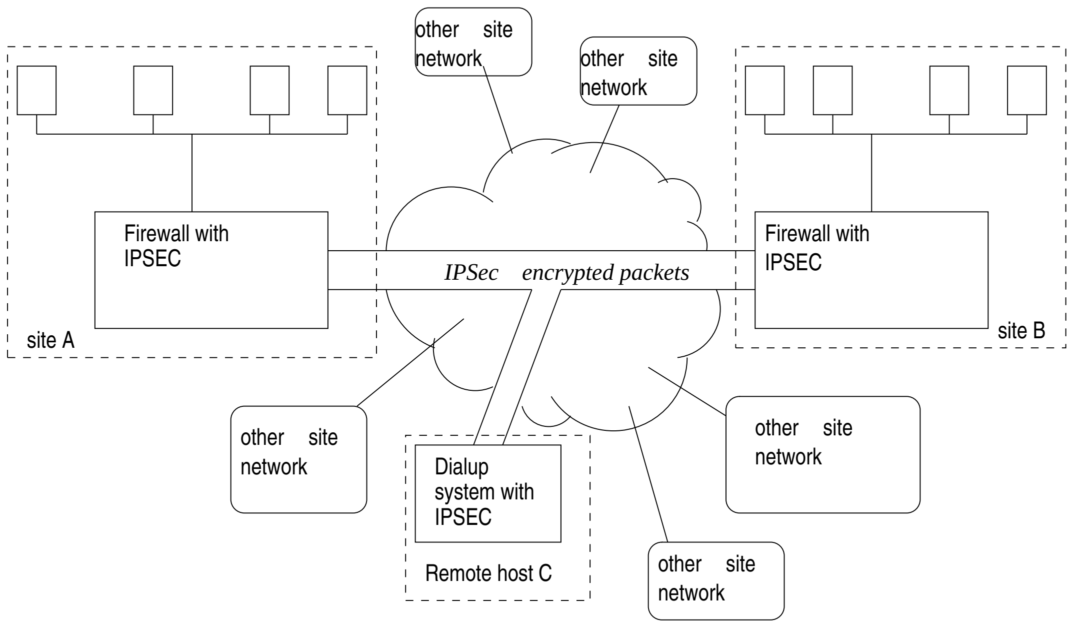
Use authenticated and encrypted network connections, this means that the only people making or receiving connections to or from your systems are ones that can be *authenticated* and afterwards you are safe from sniffers and man-in-the-middle attacks because of *encryption*. There are 2 levels:

- application level authentication and encryption of connections, such as SSL between WWW servers and browsers. The data is encrypted by the network applications.
  - these are between individual programs, not systems or sites,
  - it is used by secure servers and browsers for passing credit card numbers.
  - a system needs no special encryption or prior arrangement with another system.
- network level authentication and encryption, called IPSec (also called: Virtual Private Networks VPNs). All traffic leaving a site to one or more remote sites is encrypted.
  - typically done on a firewall system as traffic enters and leaves a site,
  - no extra work for applications, all traffic encrypted by firewall
  - IPSec must be arranged between sites so it cannot be used for arbitrary connections to single remote server programs,
  - traffic emerging from the firewall is vulnerable to attack inside the local network before it reaches the application

11.8 Encrypting network traffic: IPSec

- IPSec is also known as VPN virtual private networks,
- all IP packets to or from given destinations are encrypted and decrypted at a gateway or firewall system. Applications making connections to systems and programs on the remote destination site will have all their packets made secure as they leave the site.
- this only works between sites or dialup systems that have made prior arrangements, for example: different sites of a company of salesmen contacting their home site.
- It supports traffic encryption and authentication of the remote sites to establish the secure link. Key exchange and management is vital for links to be established safely.
- systems often change the public key used to encrypt the connection to reduce the risk of cracking.

11.9 Encrypting network traffic: IPSec

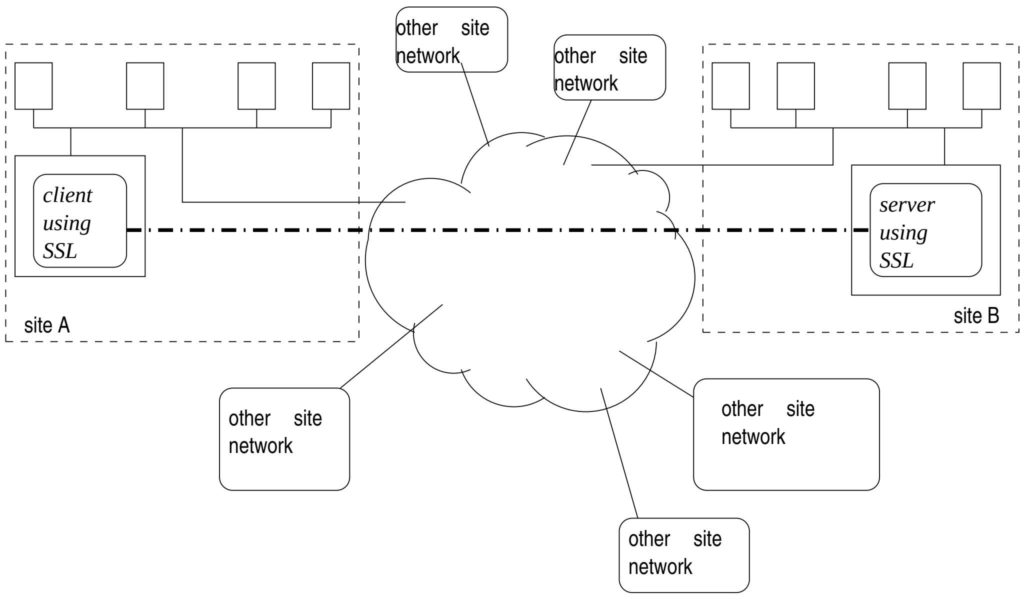


- Here sites A and B and the remote host C share a secure private network.
- no other systems on the network can spy on their traffic as it crosses the internet,
- any computer on site A contacting a computer on site B will have its traffic encrypted,
- connections can be made from computers on sites A or B to systems elsewhere on the internet but their traffic won't then be encrypted.

11.10 Application level encryption (SSL)

- SSL is a library of routines that applications can use to make secure connections,
- the best known example is "HTTPS", secure WWW connections,
- another example is OpenSSH (and the original SSH) that provides secure encrypted login sessions, it is a secure replacement for telnet,
- it uses *secret key* encryption for traffic and provides routines to support authentication using *public key* encryption,
- with WWW servers there are usually two main goals: encrypted traffic and authenication of the *server*, so you don't give your credit card number to the wrong system. The validation and authentication of the server is done using *certificates* recognised by browsers and issued by well known authorities. This is support by SSL but is really part of the application.

11.11 Using SSL



- the client program on a computer on site A connects to a program on a computer on site B,
- no other programs or systems on each site know about this or are needed to support it.

11.12 Openssh

- openssh is an end to end secure replacement for telnet, rlogin and rsh,
- it authenticates the human client and the remote server,
- it encrypts all the network traffic transmitted between the client and the server,
- openssh is an open source derivative of ssh that has become a commercial product,
- it supports 1024 bit user RSA public/private keys for authentication
- it has a choice of conventional cyphers for encrypting, currently 3DES and Blowfish,
- it is implemented on top of openssl the open source Secure socket layer, it is SSL that encrypts the data that is transmitted.
- (unfortunately it doesn't seem very easy to set up!).

11.13 Structure

There are two main programs:

- sshd the daemon that must be running on the server that receives connections. It must be run privileged (as root). This program is responsible for:
  - accepting connections
  - authenticating itself to clients
  - authenticating clients,
  - establishing the session: starting a shell etc.
- ssh the client program that makes the connection. It is not privileged. It does:
  - authenticating the remote server computer,
  - depending on various local files and the users configuration it selects and tries different user authentication methods on behalf of the user,
  - it requests other secure channels from the server, if required, for X display etc.