CHAPTER

Overview of the Internet

3

INTRODUCTION

In this chapter we explain the basic functioning of the Internet. We will use the example of the boid simulation, from the previous chapter, to discover what happens when messages are sent between two hosts. The structure of this chapter introduces the layers of the Internet in turn, from application level down to link and physical layers. We also discuss two cross-cutting issues that are particularly relevant to NVEs and NGs: multicast and quality of service (QoS). This chapter essentially corresponds to a network primer, which sets out many of the definitions for the remaining chapters, but if you are familiar with how Internet works, you might skip to Chapter 4.

Part III will contain several analyses of how the Internet performs in practice when under heavy load or when processing frequent messages.

3.1 THE INTERNET

In Chapter 1 we gave a brief overview of the history of the Internet. The term Internet was derived from the process of internetworking of networks, where each network was a local island of connectivity, a LAN, that uses a specific networking technology. There are many networks, and many internets, but the Internet is the world-wide public network.

A network, and thus the Internet, connects *host computers* or just *hosts*. Most of these are the desktops, laptops and mobile devices that we use every day, others are high-end workstations and rack-mounted server machines. Each network connects hosts together using some form of *communication technology*. A communication technology comprises a specific physical technology (wires, electromagnetic radiation, etc.) and protocols for using that technology (Ethernet, 802.11 WiFi, etc.). While small networks might use a single communication technology, for larger networks *routers* and *switches* connect different networks together.

Some terms that are often used are *clients*, *servers* and *peers*. These refer to application processes running on hosts. Typically a *server process* (or just *server*, the term is commonly used to refer to both host and process) is waiting to perform some service to several client processes (or just *clients*). The server might be a web server and the client, a web browser: the service provided is the retrieval of a web page corresponding to the uniform resource locator (URL) that the web browser sends the web server. Or the server might be a game server, where the service is a real-time communication of the state of the players of the game. A host could run several servers and clients. A *client process* connects to a server to retrieve data or perform some function. Clients thus initiate communication to servers. The term *peer* can mean several things, but generally, it implies a process that performs both the role of client and server. Thus collaborating peers might access similar data resources and perform similar functions on command. Thus peers in peer-to-peer games are all running the same code and cooperate with one another, rather than any one peer assuming the specialized role of a server.

We can already see that there are several layers to networking: there are physical connections of various types and abstractions for programming networks. In Chapter 2 when writing application code, we used sockets, which are the main programming APIs that are used to utilize networks.

If we peek under the hood, we would be able to see that there are actually five levels to networking: application, transport, network, link and physical layers. This is known as the *Internet Protocol Suite*, or the *TCP/IP stack*. This stack is often drawn as in Figure 3.1a. Against each layer of the stack, Figure 3.1b names some specific protocols and technologies that we discuss later in this chapter. It is worth mentioning that exactly what resides in the different layers is sometimes ill-determined, and different names are sometimes used for different layers. The naming we have chosen is based on the definition of Kurose and Ross (2008), and RFC 1122. Wikipedia has a page discussing the relation to other introductory networking books (Wikipedia Contributors, 2009). There is an alternative layering known as the Open Systems Interconnection Reference Model (OSI Model) (Zimmermann, 1980).

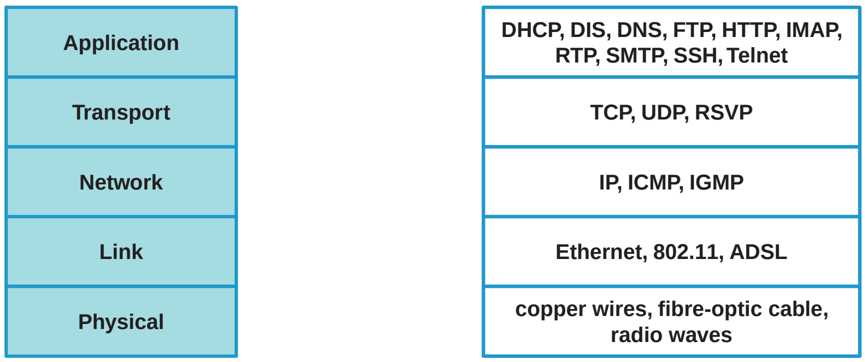

Application
Transport
Network
Link
Physical
DHCP, DIS, DNS, FTP, HTTP, IMAP, RTP, SMTP, SSH, Telnet
TCP, UDP, RSVP
IP, ICMP, IGMP
Ethernet, 802.11, ADSL
copper wires, fibre-optic cable, radio waves

FIGURE 3.1

(a) The layers of the IP suite. (b) Some of the protocols and technologies that we'll discuss

Rather than giving an overview starting at the top or the bottom layer in the stack, let's start in the middle at the *network layer*. This is the layer common to all Internet traffic, and the main protocol it supports is the Internet Protocol (IP). As discussed in Chapter 1, the IP is what binds the Internet together. The role of IP, along with some companion protocols we discuss in Section 3.3 is to *route* packets from a sending host to a receiving host. All hosts on the Internet thus need an *Internet Address* or *IP number.*[1] This is a 32-bit number comprising four bytes most commonly represented in a form such as 192.168.1.5. It is easy to look up the current IP address of a machine if it is online and one has local access: on a Windows machine, type "ipconfig" into a command shell, on Linux or Unix machines, type "ifconfig."

The IP is an unreliable protocol meaning that a packet sent on the network is not guaranteed to arrive. Thus moving up the stack, the transport layer provides more services such as reliability and ordering. Moving down from the network layer in the protocol stack we get more information that carries information for the specific link that is being traversed. This information is stored in *header information* or just *headers* that are appended at each layer. Thus applications create data and send them to the network. As it goes down the stack, at each layer transformed by the addition of new information in headers, that give more specific information. Right down at the base, information is in a form that can be transmitted in some physical medium.

Figure 3.2 shows this wrapping of information in the network stack, and to avoid confusion, we will use a specific terminology to refer to data at each level. At the application level, we start with application data or a *message*. This is sent to the transport layer, which adds a transport-layer header, to create a *segment*. This is sent to the network layer which adds a network layer header to create a *packet*. This packet is sent to the link layer, where a link-layer header and possibly a link-layer footer is added to create a *frame*. This frame is then sent over the physical layer.

Let's make this more concrete by looking at this layering for the transport of a specific type of application data: our boids from the example in Chapter 2.

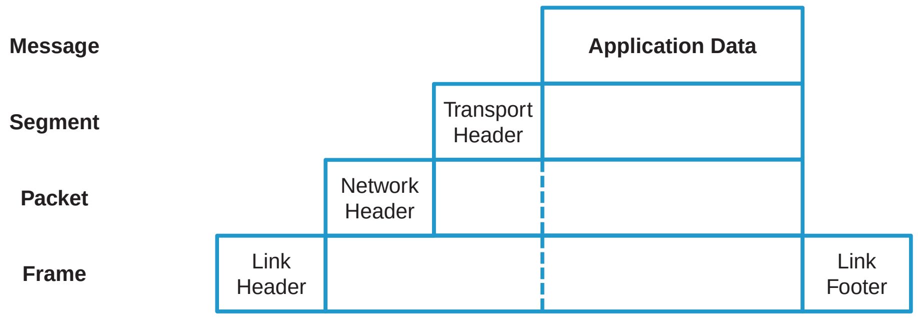


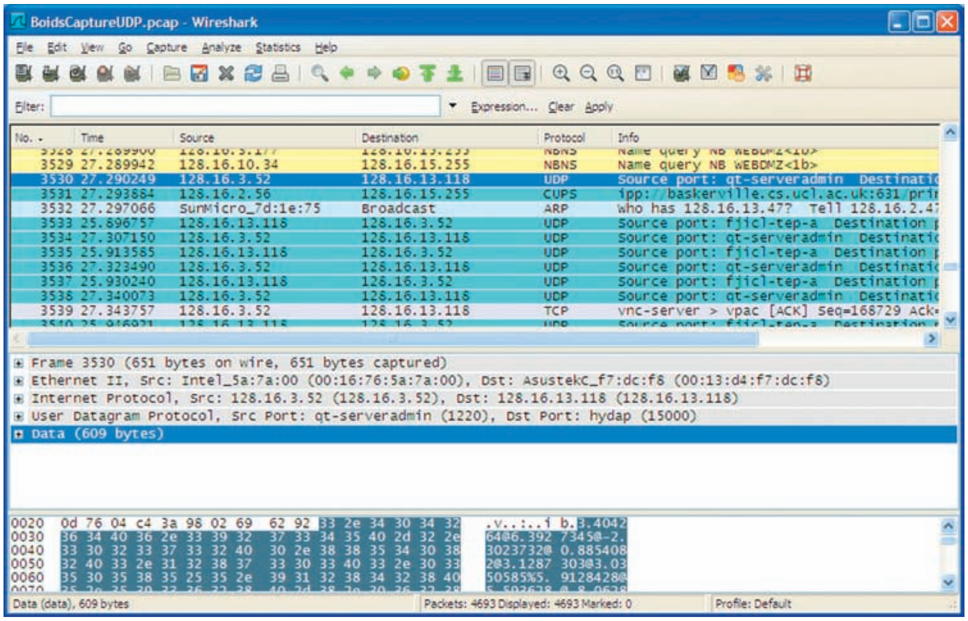
FIGURE 3.2

Encapsulation of application data within the lower layers of the protocol stack

[1] This is only partly true; we discuss network address translation and the issues of address spaces in Section 3.3.

We use a popular tool called Wireshark (www.wireshark.org) which can read the network information going to and from a host's interface to the network. Wireshark is an extremely useful tool for debugging network application or finding out how network applications work. Here we'll use it to see how information is added and removed as the boid application data travels down and up the TCP/IP protocol stack. Wireshark is available for many platforms including Windows, Mac, Linux and several variants of Unix. It is included by default in several Linux distributions. Figure 3.3 shows a snapshot of the main Wireshark screen.

Once Wireshark is started, the user starts capturing data from the network (see online material for detailed instructions). Because of the rate at which data might be captured, the typical mode of use is to capture a few seconds of traffic, then stop the capture to analyze what was captured. Figure 3.3 shows a small section of a capture of 30 seconds of network traffic. In this screen there are three important window panes below the toolbar. The top pane is a list of frames received from the network. The frames are numbered in the order received (left most column). All network traffic is recorded, and we are not interested in all the packets that are shown, some of them relate to other activities on the network. The fifth column lists the protocol that the packet corresponds to: this might be application, transport, network or link-layer protocol. We can see NBNS, which is the application-level protocol for NetBIOS over TCP/IP. NetBIOS is a legacy protocol which is supported for backwards compatibility. All of the packets using the UDP protocol are related to


BoidsCaptureUDP.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
No. -  Time       Source              Destination      Protocol  Info
3529 27.289942  128.16.10.34        128.16.15.255    NBNS      Name query NB WEBDMZ<1b>
3530 27.290249  128.16.3.52         128.16.13.118    UDP       Source port: qt-serveradmin  Destinati
3531 27.293884  128.16.2.56         128.16.15.255    CUPS      ipp://baskerville.cs.ucl.ac.uk:631/pri
3532 27.297066  SunMicro_7d:1e:75   Broadcast        ARP       Who has 128.16.13.47?  Tell 128.16.2.4
3533 25.896757  128.16.13.118       128.16.3.52      UDP       Source port: fjicl-tep-a  Destination p
3534 27.307150  128.16.3.52         128.16.13.118    UDP       Source port: qt-serveradmin  Destinati
3535 25.913565  128.16.13.118       128.16.3.52      UDP       Source port: fjicl-tep-a  Destination p
3536 27.323490  128.16.3.52         128.16.13.118    UDP       Source port: qt-serveradmin  Destinati
3537 25.930240  128.16.13.118       128.16.3.52      UDP       Source port: fjicl-tep-a  Destination p
3538 27.340073  128.16.3.52         128.16.13.118    UDP       Source port: qt-serveradmin  Destinati
3539 27.343757  128.16.3.52         128.16.13.118    TCP       vnc-server > vpac [ACK] Seq=168729 Ack=

⊞ Frame 3530 (651 bytes on wire, 651 bytes captured)
⊞ Ethernet II, Src: Intel_5a:7a:00 (00:16:76:5a:7a:00), Dst: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8)
⊞ Internet Protocol, Src: 128.16.3.52 (128.16.3.52), Dst: 128.16.13.118 (128.16.13.118)
⊞ User Datagram Protocol, Src Port: qt-serveradmin (1220), Dst Port: hydap (15000)
⊞ Data (609 bytes)

0020  0d 76 04 c4 3a 98 02 69  62 92 33 2e 34 30 34 32   .v..:..i b.3.4042
0030  36 34 40 36 2e 33 39 32  37 33 34 35 40 2d 32 2e   64@6.392 7345@-2.
0040  33 30 32 33 37 33 32 40  30 2e 38 38 35 34 30 38   3023732@ 0.885408
0050  32 40 33 2e 31 32 38 37  33 30 33 40 33 2e 30 33   2@3.1287 303@3.03
0060  35 30 35 38 35 25 35 2e  39 31 32 38 34 32 38 40   50585%5. 9128428@

Data (data), 609 bytes     Packets: 4693 Displayed: 4693 Marked: 0     Profile: Default
```

FIGURE 3.3

A screenshot of Wireshark

the UDP version of our boids application. We also see the CUPS protocol, which is the Common Unix Printing Service: essentially a printer status message. We see ARP, Address Resolution Protocol, which is a link-layer protocol for discovering how to connect to a host on the LAN. We discuss this later in Section 3.4. Finally, we also see VNC, or Virtual Network Computing protocol, which is a system for remotely accessing another desktop. This pane also shows the time (second column), source and destination IP addresses, and some information about the protocol (sixth column).

Below this pane is an overview of a frame received on the network. This shows the different layers of the network stack: we can see that we've selected a frame which is from Ethernet, that this is an IP packet, that this is a UDP segment, and that it contains application data. We'll be examining the different layers in much more depth later in the chapter, but from this view we can see that Frame 3533 had 662 bytes on the wire. This corresponds to 620 bytes of application sent data, and thus there are 42 bytes of header information in the different protocol layers. The bottom pane shows the data within the frame that we've chosen. If you select a particular layer in the middle pane, the bottom layer highlights all the data in that layer. In the figure, we've selected the application layer and highlighted in the bottom pane the data that would be read by the application. In the bottom pane, we see two views: on the left a hexadecimal dump of the data and on the right an interpretation of that data as ASCII. On the right we can recognize a string that starts "3.404564@6.3927345@2.3023732@0.8854082@3.1287303 …". This is the format of the strings that we were using in the boids example in the previous chapter.

We'll now examine the properties of different levels of the protocol stack by referring to this and other logs from Wireshark.

3.2 APPLICATION LAYER

The application layer is the layer at which applications access networking functions. Its role is to provide a simple API which abstracts much of the complexity of the lower levels. The application developer simply requests specific transport-layer protocols to be used and sets any options they need on those protocols.

On any particular machine many processes can interact with the Internet at once; thus in addition to the IP address of the machine, there is the concept of *ports*, which correspond to a 16 bit unsigned integer. At the application level, as a server you make a request to *listen* to a particular port for incoming applications, and as a client you request to connect to a port on a destination machine. When, as a client you request a connection, a port is created on the local machine which is connected to the port on the server application. Note that the outgoing port on the client may either be specified port number or chosen at random from a currently un-used port. The actual allocation of ports is slightly different for TCP and UDP, see Section 3.3 which also includes examples.

The server's port number will need to be well known to potential connecting clients. Some ports are commonly standardized across machines. For example, web servers usually use port 80. See Section 3.2.3 for other examples. In other cases the application client will need to be told which remote port to connect to. In our examples from Chapter 2, because our boids application is a new type of application where we wrote both client and server sides, the choice of ports was open for us to decide. Because in both TCP and UDP implementations each process acted as both client and server, both the port to listen to as server and the port to connect to on the peer were given as command line parameters.

When we open ports, either for listening or sending, we need to ask for a particular transport-layer protocol (eg: TCP, UDP or raw IP) at the point of opening. In Chapter 2 we gave example code of doing this using Java and the socket API. The choice depends on the basic Quality of Service (QoS) we require from the network. We discuss QoS as it relates to the choice between TCP and UDP in Section 3.2.2, but it's a large topic and we will return to it later in the book.

3.2.1 Application layer protocols

When two entities communicate, they need to use an agreed protocol to exchange data through the use of messages. An application layer protocol defines both a standard for representing data in messages, the *syntax*, and the expected meaning of those messages, the *semantics*.

In our boids application, the syntax of application-level protocol is very simple. ASCII strings are passed in both directions, and each string consists of a sequence of floating point numbers separated by punctuation marks. Thus, within the data in Figure 3.3, we can recognize a string of floats, separated by @ and % signs. Section 2.3.3 outlined the code for sending and receiving these strings. The semantics of these messages is that each set of six floats corresponds to the position and velocity of one boid, and that the receiving process should move the corresponding boid appropriately.

The syntax protocol is thus very simple, but there is a very important property missing: how many boids we might expect to read. As you might have found through experiment, the current code expects to receive the same number of boids as it sends. For both the UDP and TCP implementations, the clients will get confused if their peer sends the wrong number of boids and thus both clients must be started with the same configuration information for the number of boids. In Chapter 4, we'll be making several changes to the network protocol to make a more general system for boid animations. Amongst these changes are making a binary protocol to save space and making the number of boids we are sending explicit. In general, the network protocol must convey enough information, given the known constraints of the sender and receiver, so that the receiver can faithfully reproduce the semantics of the message sent. In our example, one could achieve a flexible number of boids in several ways. One way would be to declare the number of boids at the beginning of the message. Another approach would be to introduce a special character into the string which indicates the information for the last boid that had just been read and not to expect any more.

There are three main considerations in the design of such protocols: *compactness*, *robustness* and *efficiency*. Compactness refers to the number of bytes required by messages in the protocol. Especially for networked environments with many rapidly changing objects, we will need to be particularly careful about the amount of data sent on the wire. This means that the representation of data, such as floats as 16, 32 or 64 bits or ASCII, or flags as integers or bits, becomes very important.

Robustness is concerned with the ability to deal with corrupted, missing or hostile data. Although we can assume that individual packets from the network layer will not be corrupted, the sending application might make a mistake or the data might be corrupted in a way not detected by the lower levels of the stack. Additionally if we use an unreliable transport such as UDP, individual packets might be lost. For example, in the TCP implementation of the boids application, if the application data did get corrupted, for example by the other client sending a single bad data packet, it would be quite hard to re-find the start of the application data, and thus assuming the stream is good again, which data corresponds to the first boid? This leads to many application-level protocols having well-known key phrases, which the client can search the stream for. Our protocol does not currently have such a phrase, but then the code to parse the data is very simple.

Efficiency is related to compactness, but refers to the way in which the application relays only important data. A protocol is efficient when it sends the minimum amount of data to the network that would be necessary for the receiver to understand the semantics that was trying to be conveyed. Thus it is not only related to the compactness of the syntax, but also to the redundancy of data. Application layer protocols definitely want to reduce redundant data, that is information that is not needed to be sent because it is implied by the semantics of other parts of the message. In our simple example, there is no point in relaying both orientation and velocity of the boid, as we specifically said that boids are oriented along their direction of travel. In general, as an application protocol designer, you definitely want to make sure that sufficient information is conveyed. Making sure that only necessary information is conveyed is harder. This is especially true when you have to make the choice of choosing unreliable or reliable transport. There are overheads to reliable transport, which means that it isn't always a good fit to an application. This means that there is a tendency, especially when supporting a range of clients on heterogeneous implementations, to make the protocol somewhat inefficient, putting in some redundant data so that applications always have the up-to-date state.

3.2.2 Application QoS

From the argument about the contents of particular application layer messages, the main property that we might want the network to exhibit is reliability. In designing the content of our messages, we might make very different choices if we can assume that all our messages get through in the right order or that some of them are lost. At the application level our decision on this will usually dictate whether we select TCP or UDP as the transport layer. The most common reason for not needing reliability is when the data representation is redundant over time. In this case, the data being passed replaces previous information. So with our boid example, the position and velocity of the boid received completely replaces the position and velocity that was last received. We could, if we were really concerned about compactness of our data, encode only changes in position and velocity. The floating point values of differences would be smaller in magnitude than the absolute positions, so could be compressed more efficiently.

A fuller list of criteria that we might use would include:

- Reliability: that we know whether the messages are received.
- Timing: that we know how long the messages will take.
- Bandwidth: that we know how many messages we can send.

Unfortunately the latter two are not elements that we have much control over, but are factors that we need to know in order to manage how the application sends data. Many, if not most, Internet applications want to transfer data as fast as possible (e.g. data transfer such as email and http downloads), but it doesn't matter how long any particular piece of the data takes to download. Thus, typically, bandwidth is more important than timing. This contrasts with streaming media, which would include the functions of NVEs and NGs concerned with updates, where the data is very timely. For example, in audio and video streaming, we would need to estimate the likely timing and bandwidth in order to select the most appropriate version of the media to send; if there isn't bandwidth, then video can always be sent at a lower resolution.

3.2.3 Common applications and ports

There are many thousands of application servers and clients already available on the Internet. Many of these servers and clients support well-known *network services*, which are available on well-known ports. We already mentioned that a web server would normally be available on port 80. If it isn't then the web browser would need to be told the port as part of the URL.

Thus although whoever runs a specific server can choose the port number, there are well-known recommended defaults for many common services. The Internet Assigned Numbers Authority (IANA) maintains the table of recommended ports (IANA, 2009) amongst other important roles such as IP address allocation. Table 3.1 shows a few of the reserved port numbers for some services that you might have configured for your own machines. Table 3.1 also notes whether these services use UDP or TCP for transport.

FTP is the File Transfer Protocol, used for transferring files over the Internet. This protocol has a long history, though most users' experience of it now is through a web browser which handles both URLs starting ftp:// as well as http://. FTP is defined in RFC 959, see Section 3.2.4 for a discussion of what an RFC is. SMTP is the Simple Mail Transport Protocol, and it is a mechanism for the wide-area transmission of email, and also the sending of email. DNS is the Domain Name Service which we discuss in more detail in the next section. Finger isn't a common service anymore, but it supports awareness of whether a person is logged in or not. It is supported in most Unix systems. HTTP is the Hyper-Text Transfer Protocol that underpins the

Table 3.1 Some Example Services, Their Ports and Transport Protocols

Service Full Name	Short Name	Port	Transport
File Transfer Protocol	ftp	21	TCP
Simple Mail Transfer Protocol	smtp	25	TCP
Domain Name System	dns	53	UDP
Finger	finger	79	TCP
HyperText Transfer Protocol	http	80	TCP
Post Office Protocol (Version 3)	pop3	110	TCP
Internet Message Access Protocol	imap	143	TCP
Hypertext Transfer Protocol Secure	https	443	TCP
File Transfer Protocol Secure	ftps	990	TCP
Distributed Interactive Simulation	dis	3000	UDP
BZFlag Game Server	bzflag	5154	TCP
Quake Game Server	quake	26000	UDP

World Wide Web. For personal access to email, one might use one of POP, the Post-Office Protocol, or IMAP, the Internet Message Access Protocol. When a protocol has the letter S at the end, it commonly means a secure version of the same protocol, and thus a secure network service is combined with the original protocol. Thus HTTPS is HTTP run over a secure network service such as Secure Socket Layer (SSL) or Transport Layer Security (TLS) and FTPS is the same for FTP. Finally, we've included the same information for three NVE systems which appear in the IANA table, and which we mentioned in Chapter 1. Distributed Interactive Simulation (DIS) packets are sent by UDP by default to port 3000. The BZFlag Games server listens to incoming TCP connections on port 5154 by default. The Quake Game Server by default listens to incoming UDP packets on port 26000. We'll have more to say about DIS in Chapter 8 and we'll be discussing Quake at a couple of points in Part III. This is not to say that these applications only use these ports; FTP, for example sends control information on port 21, but the data travels via port 20; BZflag connections are made to port 5154, but the server then tells the client to reconnect to another port.

The well-known services (including FTP through to FTPS in Table 3.1) are allocated to port numbers between 0 and 1023, which are known as *reserved ports*. When you create a novel service, you would use a port number above 1024, but below 49151. If you try to create a port number below 1024, you will need privileged access to the machine, plus the IANA recommendation sets expectations that a specific protocol will be used when you use that port. For ports at or above 1024, the expectation is that application authors may select a port for use, unless they know a service that is commonly used on that port. The full IANA table lists several thousand applications and their typical ports.

3.2.4 RFCs

The concept of a Request for Comments (RFC) dates from the early days of the ARPANET. Originally these were actually requests for comments from researchers building key parts of the ARPANET. RFCs are numbered. RFC 1 was written by Steve Crocker of UCLA based on discussions of a working group. The title was "Host Software," and it discussed how hosts will connect to the IMPs that formed the ARPANET.

Originally simply working documents, over time, RFCs have evolved to be a more formal mechanism which Internet Engineering Task Force (IETF) describes the best practice and recommendations for common protocols. The RFC process itself is described in RFC 2026 "The Internet Standards Process, Revision 3".

Many common protocols are defined by a series of RFCs which record the evolution and refinement of the protocol. FTP is currently defined by RFC 959, but this supersedes RFC 765 and several other RFCs all the way back to RFC 114. Other protocols are complex and are discussed in several RFCs. DNS is an example: one RFC, RFC 920, is solely concerned with the names of the top-level domains (TLDs) and country codes. TCP/IP itself is discussed in RFC 1122.

Although RFCs are the standards for the Internet, they can also be light hearted, as clearly demonstrated by RFC 1149 "Standard for the transmission of IP datagrams on Avian Carriers". This RFC has since been updated by RFC 2549 "IP over Avian Carriers with Quality of Service".

3.2.5 DNS

The DNS is used by applications to map user memorable names (hostnames) to IP numbers. Thus looking up the name www.cs.ucl.ac.uk via DNS will retrieve the number 128.16.6.8. DNS is thus a database, and it's stored in a distributed manner. Applications query DNS to find the IP addresses of the services they need. DNS is one service whose configuration is practically necessary when you connect to the Internet. It's so important that if you manually configure your Internet connection, you can configure the IP addresses of two DNS servers. Of course you must know the IP address of the DNS server(s), otherwise how would you look it (them) up? See Figure 3.31 for an example of these settings in Windows XP.

The looking up of hostname is such a common action in network applications that access to DNS is typically handled by a single API function. Of course, look up requires a network query, so this function stalls and waits for a result. This network stall is the cause of the appearance of the slow start to loading web pages on a web browser: if the web browser hasn't visited the page recently, it must wait for DNS to give it the IP address of the web server. Windows and Unix have a command tool to look up DNS names—"nslookup".

The names in the DNS database are structured hierarchically. The rightmost element must be one of several approved Top-Level Domains (TLDs), such as .com, .net and .org, as well as country-specific TLDs such as .uk (United Kingdom), .fr (France), .tv (Tuvalu).[2] The management of TLDs is overseen by the Internet Corporation for Assigned Names and Numbers (ICANN), though country-specific codes are delegated to other authorities. At the time of writing ICANN was in the process of expanding the set of TLDs. Names must be registered with DNS, and this requires payment as the database needs to be kept reasonably small. There are many issues with who gets to register specific names, and fortunes have been made in registering and then selling on popular names. Names are registered with domain name registrars who are accredited with the ICANN. After the TLD comes other domains, such as google (for .google.com) and ac (for .ac.uk). There can be several lower-level domains, such as cs.ucl.ac.uk. The full list of hostname and all domain names is referred to as the fully qualified domain name (or FQDN). A FQDN is an unambiguous name and refers to a specific machine.[3]

Another important feature of DNS is the way in which a query is resolved. As we noted, DNS is structured as a hierarchical database. There could be many more names in the DNS database than there are IP addresses: the mapping is many to one. Thus no-one server is responsible for holding all the entries: not only are there a lot, but there are constant changes to the database, and every host on the Internet is potentially querying the database repeatedly. The main strategy to cope with the load is to cache DNS query lookups locally. Each host will keep a cache of recently looked-up names, and will keep these names for a short period, usually 24 hours. Then, the local DNS server or a server at the *internet service provider* (ISP) might cache the queries. If these caches fail, the query then resolves in a top–down manner, from the *root DNS servers* down. The root servers don't keep a list of all the names, but can always refer the query to a server which can answer the question or recursively delegate it to another server. Thus if no one on your local network has looked up narok.cs.ucl.ac.uk recently, this will be queried via a sequence such as: find the server for .uk, then .ac.uk, then ucl.ac.uk, then the DNS server ns1.cs.ucl.ac.uk, which is the *authoritative name server* for the cs.ucl.ac.uk domain.

DNS is an essential service so a reasonable expectation is that on the Internet your DNS lookups are secure and reliable. We thus expect that the DNS service we configure ourselves or is configured for us by our ISP has a certain level of integrity: it contains the canonical mapping from hostname to IP address. However, it is just a network service and thus it can be attacked.

Recall that DNS is both cached and hierarchical. Thus, if a domain has been looked up recently then the IP address remains in the cache for about 24 hours. DNS cache poisoning involves a malicious server pretending to be the authoritative DNS server for a domain, and injecting false records into the cache. Once a false record is in the cache, hosts who query against this cache can be directed to a host masquerading as the

[2] Tuvalu is a small Island in the Pacific Ocean. Its TLD of .tv has an obvious attraction to some media companies. Fortunately .tv domains can be registered with VeriSign, the income from which goes to Tuvalu (Salon, 2000).

[3] This is not strictly true: a DNS name can actually map to several IP addresses, and routers could route of packets with the same IP address to one of the several machines. These techniques provide scalability to certain very large web services which cannot be served from a single machine.

intended host. At the time of writing, although such vulnerabilities were theoretically possible, a demonstration of an exploit had been given by Dan Kaminsky (CERT, 2008).

Not all subversions of the DNS system are hostile. Let us reiterate that DNS is just a network service, and thus it's possible to create parallel or complementary DNS services. One example is OpenNIC which provides an alterative DNS without the restrictions imposed by hierarchical name registries of the "normal" Internet. You can access a wider variety of naming, with TLDs such as .geek and .eco. The IP addresses that are retrieved from OpenNIC are, of course, reachable on the Internet. If you look into such alternative DNS services, please be careful and understand that the risk of name poisoning on otherwise well-known hostnames (e.g. your bank) may be higher.

3.2.6 Telnet and HTTP

Telnet is an application-level protocol that actually does very little; however, it has been around for a very long time in Internet terms (i.e. since before TCP/IP) and its uses are many and various. It can be described as providing a "raw TCP" service. A *telnet client* uses telnet to connect to a TCP server port. As long as that TCP server port speaks in 7-bit ASCII strings, the telnet client can communicate with it. Telnet's original use was to log onto remote machines. One could "telnet in", log in and then issue command native to the remote machine by typing them on a console. This practice is uncommon with telnet now, except on local networks, as it is insecure: telnet passes ASCII strings back and forth, so over untrusted networks such remote login is typically done by SSH (see Chapter 13).

One of the most common uses of telnet is to test TCP services. For example, one can telnet directly to a web server and issue commands to retrieve data. An example is given below. On the command line (this machine being a SunOS machine, but the command telnet exists on most operating systems), we type:

```
telnet www.cs.ucl.ac.uk 80
the telnet program shows us this:
Trying 128.16.6.8...
Connected to haig.cs.ucl.ac.uk.
Escape character is "^]".
We then type the following, and press return twice:
GET /staff/A.Steed/ HTTP/1.1
Host: www.cs.ucl.ac.uk
```

and we get the following response, which we've truncated. It contains both a HTTP header and the HTML page corresponding to one of the author's web pages.

```
HTTP/1.0 200 Document follows
MIME-Version: 1.0
Server: CERN/3.0
Date: Sun, 08 Feb 2009 15:25:18 GMT
Content-Type: text/html
```

```
Content-Length: 16150
Last-Modified: Wed, 21 Jan 2009 17:42:00 GMT
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" dir="ltr">
<head>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
<meta name="keywords" content="A. Steed, Anthony Steed,
  Department of Computer Science, University College London,
  virtual environments, virtual reality, computer graphics"/>
<title>Anthony Steed, Department of Computer Science,
  UCL</title>
....
```

The header runs from the first line down to the line starting "Last Modified … ". The rest of the response is the actual HTML file. We don't have space to give a detailed overview of HTTP, but we can note some important lines. The line "HTTP/1.0 200 Document follows" contains a response code, 200, which indicates that the requested document is OK. If we had typed the following:

```
GET /staff/A.Steel/ HTTP/1.1
Host: www.cs.ucl.ac.uk
```

we would have got the response

```
HTTP/1.0 404 Not found-file doesn't exist or is read protected
  [even tried multi]
```

and the subsequent HTML might have, if we were lucky, given useful and perhaps witty help in finding the correct page we were looking for (there is no A.Steel on the staff, so their home page doesn't exist). There are several other codes that can be generated. We can see here that the application layer protocol for HTTP is fairly simple: there are several header lines, followed by data lines.

3.3 TRANSPORT LAYER

The transport layer is responsible for implementing the end-to-end communication between two processes. The transport layer thus implements the connectionless (UDP) or connection-oriented (TCP) communication between the two hosts. An important aspect of networking is that the transport protocol only operates between two hosts: nothing in the lower layers of the stack knows anything about the content of the application messages, they only read the header information for that layer. This means that the infrastructure of the Internet, the routers, switches and so on, that provide the path for any packet to move from do not look at the contents of the packets.[4] The most important impact this has is that TCP must implement its reliability without any help from the network infrastructure. Reliability is done at the transport level, and only the source and receiver perform any actions. The transport layer does perform another important function: it must decode the incoming IP packets, which correspond to either UDP or TCP segments, check the port numbers and deliver them to the correct socket in the appropriate application process.

3.3.1 Implementing UDP

UDP which is defined in RFC 768 provides very little functionality over the underlying IP network layer, thus providing maximum flexibility. Applications which create UDP sockets to send data should not rely on the data they send being received. However, UDP is attractive for a few of reasons:

- There is no delay in sending information. As can be seen in the UDP code for our boid application, we can see the point at which the UDP segment is sent. This dispatches the message immediately.
- There is no connection setup. UDP clients just send messages to servers, and the server need not be expecting any particular client to send a message. This contrasts with TCP, see the following section.
- There is no congestion control, ordering or reliability. In real-time NVEs, we might not want the standard TCP reliability, which causes data to be resent, because our data are changing so fast.
- The UDP header is very small.

A UDP segment is set out as shown in Figure 3.4. Each UDP segment includes the source port and the destination port. These are 16 bits numbers, so must lie in the range 0–65535. As we have noted, destination port must be known to the client (sending) process. However, the source port isn't that important; it is only needed for returning packets to the sender. Thus, at the application level, when a UDP socket is created to send a message, one can either request a specific port or get a random one. The UDP segment also includes the length of the data and a checksum for the data, again both 16 bits. The length includes the header, so the total length of a UDP segment must be less than 65,536 bytes. The checksum allows for some

Bits	0 15	16 31
0–31	Source Port	Destination Port
32–63	Length	Checksum
64+	Data	

FIGURE 3.4

Layout of a UDP segment

[4] This is not strictly true in all situations. Some ISPs prioritize traffic depending on its content, a controversy that is referred to as net neutrality.

error checking at destination that the UDP packet is intact; there are other such checks at other layers in the protocol stack.

Figure 3.5 shows the relationship between the source and destination ports. The transport layer must multiplex and demultiplex network layer packets to the correct application process. On the right, at the lower levels, there is a sequence of incoming frames, which becomes a sequence of incoming packets. The network layer passes these up to the transport layer, which then distributes them to the correct application process depending on their destination port. In this figure, a UDP segment has been received for port2, but another UDP segment has been received for port3. Segments for port2 and port3 are distributed to different applications.

Let's take a detailed look at the UDP protocol in our boids application, which uses a separate socket to send and receive messages. We've started the process on two machines: seychelles.cs.ucl.ac.uk (with IP address 128.16.3.52) and narok.cs.ucl.ac.uk (128.16.13.118). We started the Java UDP boids process on narok first with the following command line:

```
java -jar boidsUDP.jar 10 128.16.3.52 15000 15001
```

The meaning of this is that we'll simulate 10 boids. Our collaborating process is on 128.16.3.52 (seychelles). The receiving port on that machine is 15001. We'll also create a server port on our machine on port 15000. A few seconds later we started the process on seychelles with

```
java -jar boidsUDP.jar 10 128.16.13.118 15001 15000
```

Note the reversal of the two server ports. Figure 3.6 shows a view of the messages between two hosts running the boids application. Within Wireshark, we have filtered

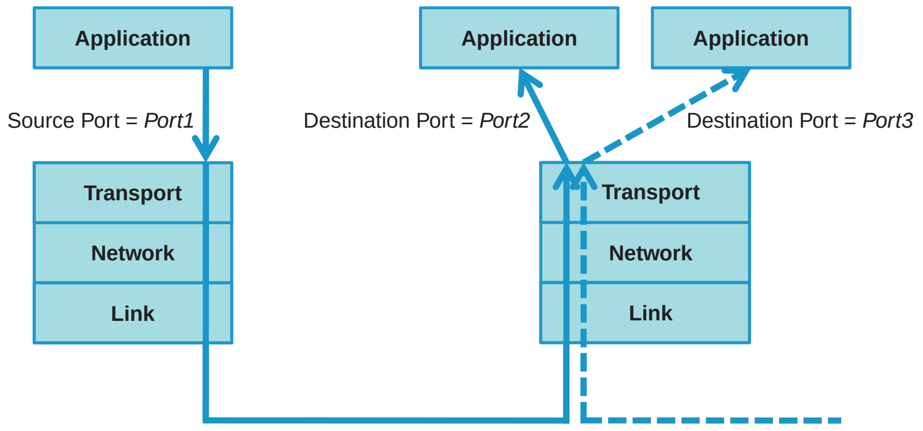
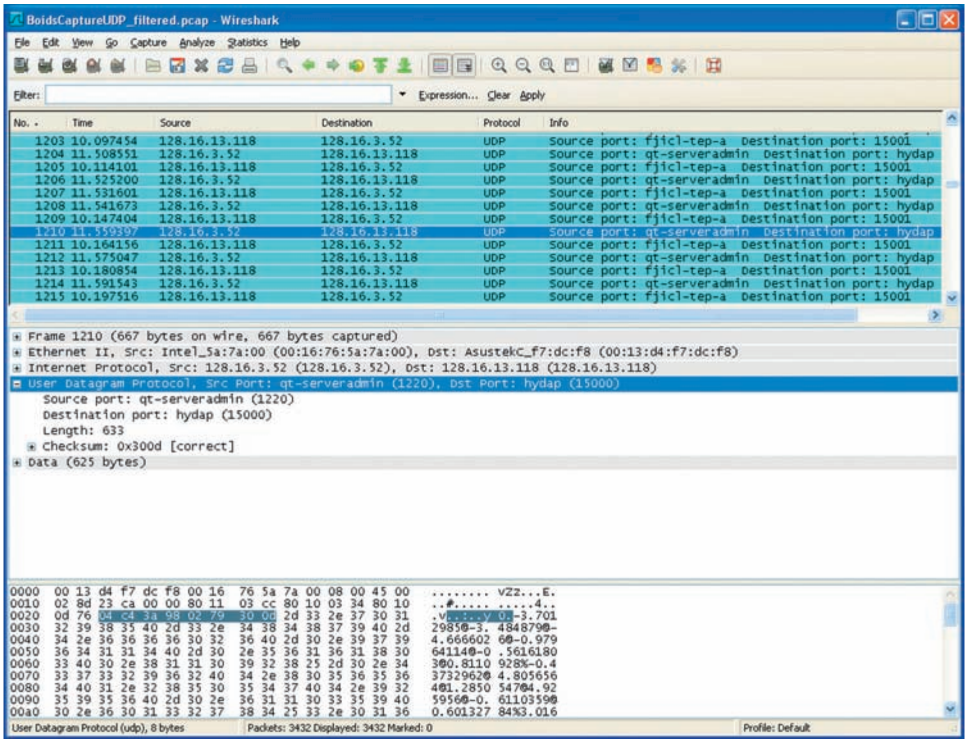


FIGURE 3.5

UDP connections from source to destination. The transport layer must multiplex and demultiplex incoming network layer packets to the correct application processes. It uses the destination port to do this. The solid line represents a single message's source and destination port. The dotted line shows another arriving UDP packet, destined for a different application process


BoidsCaptureUDP_filtered.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
No. .  Time       Source          Destination     Protocol  Info
  1203 10.097454  128.16.13.118   128.16.3.52     UDP       Source port: fjicl-tep-a  Destination port: 15001
  1204 11.508551  128.16.3.52     128.16.13.118   UDP       Source port: qt-serveradmin  Destination port: hydap
  1205 10.114101  128.16.13.118   128.16.3.52     UDP       Source port: fjicl-tep-a  Destination port: 15001
  1206 11.525200  128.16.3.52     128.16.13.118   UDP       Source port: qt-serveradmin  Destination port: hydap
  1207 11.531601  128.16.13.118   128.16.3.52     UDP       Source port: fjicl-tep-a  Destination port: 15001
  1208 11.541673  128.16.3.52     128.16.13.118   UDP       Source port: qt-serveradmin  Destination port: hydap
  1209 10.147404  128.16.13.118   128.16.3.52     UDP       Source port: fjicl-tep-a  Destination port: 15001
  1210 11.559397  128.16.3.52     128.16.13.118   UDP       Source port: qt-serveradmin  Destination port: hydap
  1211 10.164156  128.16.13.118   128.16.3.52     UDP       Source port: fjicl-tep-a  Destination port: 15001
  1212 11.575047  128.16.3.52     128.16.13.118   UDP       Source port: qt-serveradmin  Destination port: hydap
  1213 10.180854  128.16.13.118   128.16.3.52     UDP       Source port: fjicl-tep-a  Destination port: 15001
  1214 11.591543  128.16.3.52     128.16.13.118   UDP       Source port: qt-serveradmin  Destination port: hydap
  1215 10.197516  128.16.13.118   128.16.3.52     UDP       Source port: fjicl-tep-a  Destination port: 15001
```

```
⊞ Frame 1210 (667 bytes on wire, 667 bytes captured)
⊞ Ethernet II, Src: Intel_5a:7a:00 (00:16:76:5a:7a:00), Dst: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8)
⊞ Internet Protocol, Src: 128.16.3.52 (128.16.3.52), Dst: 128.16.13.118 (128.16.13.118)
⊟ User Datagram Protocol, Src Port: qt-serveradmin (1220), Dst Port: hydap (15000)
    Source port: qt-serveradmin (1220)
    Destination port: hydap (15000)
    Length: 633
  ⊞ Checksum: 0x300d [correct]
⊞ Data (625 bytes)
```

```
0000  00 13 d4 f7 dc f8 00 16  76 5a 7a 00 08 00 45 00   ........ vZz...E.
0010  02 8d 23 ca 00 00 80 11  03 cc 80 10 03 34 80 10   ..#..... .....4..
0020  0d 76 04 c4 3a 98 02 79  30 0d 2d 33 2e 37 30 31   .v..:..y 0.-3.701
0030  32 39 38 35 40 2d 33 2e  34 38 34 38 37 39 40 2d   2985@-3. 484879@-
0040  34 2e 36 36 36 36 30 32  36 40 2d 30 2e 39 37 39   4.666602 6@-0.979
0050  36 34 31 31 34 40 2d 30  2e 35 36 31 36 31 38 30   641140-0 .5616180
0060  33 40 30 2e 38 31 31 30  39 32 38 25 2d 30 2e 34   3@0.8110 928%-0.4
0070  33 37 33 32 39 36 32 40  34 2e 38 30 35 36 35 36   3732962@ 4.805656
0080  34 40 31 2e 32 38 35 30  35 34 37 40 34 2e 39 32   4@1.2850 547@4.92
0090  35 39 35 36 40 2d 30 2e  36 31 31 30 33 35 39 40   5956@-0. 6110359@
00a0  30 2e 36 30 31 33 32 37  38 34 25 33 2e 30 31 36   0.601327 84%3.016
```

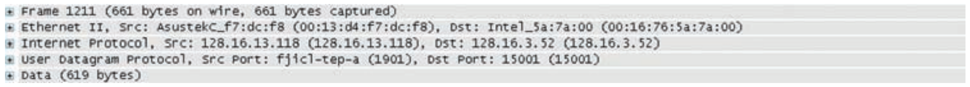
User Datagram Protocol (udp), 8 bytes | Packets: 3432 Displayed: 3432 Marked: 0 | Profile: Default

FIGURE 3.6

UDP segment information for frame 1210 of the UDP boids demo

out all the other traffic which is not related to the boids application. We see only UDP packets, half with the source IP address 128.16.13.118 and destination IP address 128.16.3.52 and half with the source and destination IP addresses exchanged. We have selected Frame 1210 from the log. The information on the right is a little confusing at first, it states the source port is *qt-serveradmin* and the destination port is *hydap*. But looking at the middle pane will clarify things for us.

In the middle pane, we've selected the UDP level. It tells us that the "Src Port" is qt-serveradmin (1220) and the "Dst Port" is hydap (15000). This is a boid message going from the randomly chosen (1220) source port on seychelles to the known server port (15000) on narok. The reason why we see qt-serveradmin and hydap is that these are services that are well known to use these ports. On the IANA website, which we have referred to earlier, these are listed against the ports 1220 and 15000. Wireshark is automatically using a very similar table to guess what protocol is being used but in our case these guesses are wrong. We can't avoid such confusions completely: the source UDP port is randomly chosen, but we could use a server port that isn't commonly used. If you look in Figure 3.6, at the next frame, 1211, you'll see that Wireshark hasn't guessed a service for port 15001, and the IANA website indicates that this port is unassigned. In the middle pane we can also see that the length is 633: this is the application message (625



```
⊞ Frame 1211 (661 bytes on wire, 661 bytes captured)
⊞ Ethernet II, Src: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8), Dst: Intel_5a:7a:00 (00:16:76:5a:7a:00)
⊞ Internet Protocol, Src: 128.16.13.118 (128.16.13.118), Dst: 128.16.3.52 (128.16.3.52)
⊞ User Datagram Protocol, Src Port: fjicl-tep-a (1901), Dst Port: 15001 (15001)
⊞ Data (619 bytes)
```

FIGURE 3.7

Summary information for frame 1211

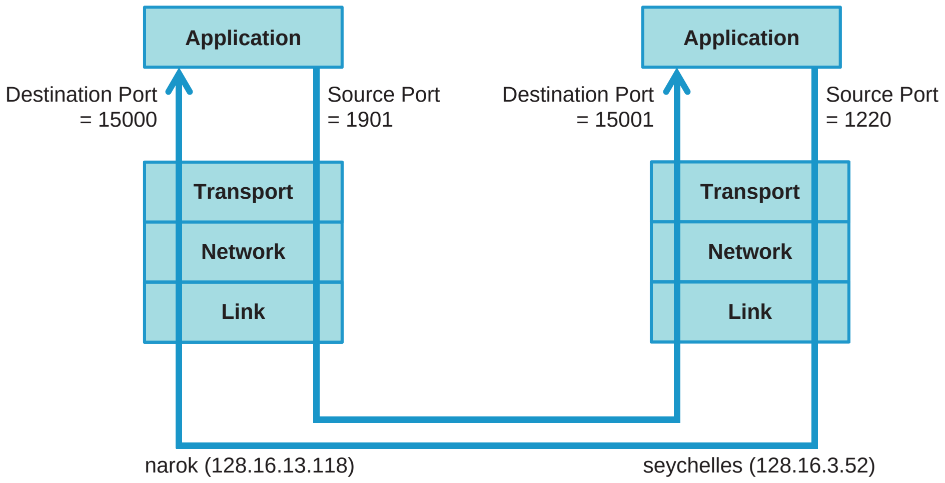


FIGURE 3.8

IP address and port numbers for the example of the UDP boids application

bytes) plus the 8 bytes for the UDP header. The checksum is 0x300d and Wireshark indicates in square brackets that this is correct.

In the bottom frame we can see highlighted the eight bytes which make up the UDP header. On the right of this bottom frame, we can see that immediately after this highlighted section, we can recognize a sequence of floating point numbers represented as ASCII strings.

Figure 3.7 shows the summary information for the subsequent frame, Frame 1211, which is a boid message in the return direction. Here we can see that the source port on narok was randomly chosen to be port 1220. Note that this frame is slightly smaller (661 bytes). The message size will change because the ASCII strings we use are not of constant length: e.g., if they are negative, there will be a "–" sign.

The full situation with source and destination addresses and ports is shown in Figure 3.8.

3.3.2 Basics of TCP

Compared to UDP, TCP offers four major services:

- Connection-oriented services with bi-directional (full-duplex) communication.
- Reliable transmission of messages in each direction.

- Congestion avoidance, using variable rate transmission.
- In order and nonduplicate delivery of information.

The transport layer has significantly more responsibility when implementing connection-oriented (TCP) communication. Because the underlying network is unreliable, reliability must be added at this layer. The first implication this has is that the transport layer must buffer information, because if any packet goes missing, it may need to regenerate information to send. The implementation of TCP and its behavior are not simple (Stevens, 1994; Wright & Stevens, 1995; Stevens, 1996), but we will pull out some important features that will affect our use of TCP in an NVE situation.

When sending data, the key difference to UDP is that with TCP, the application process pushes data down a pipe without concern for the actual contents of the packet. Thus, even at the transport layer, we no longer have a simple mapping of messages to likely segments to go to the network. What happens is that data sent by the application are buffered, and the transport layer takes chunks of this data and adds a header to make a TCP segment. This TCP segment is then passed to the network layer. This is visualized in Figure 3.9.

Note that when making a segment, a specific amount of data is extracted from the buffer; there is no knowledge of any meaning in the messages that have been sent. The amount of data that is taken out of the buffer to form the segment is actually chosen so that when the segment is sent to the network layer, it will not be fragmented. This size, known as the maximum segment size (MSS), depends on the particular link

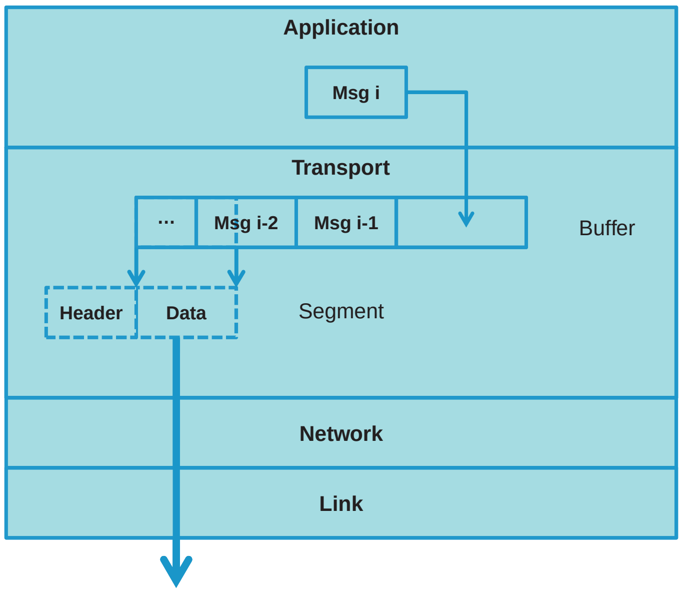


FIGURE 3.9

Outline of buffering of data in a TCP connection layer that will be used for all the connections. As we will see in Section 3.4.2, sending large segments to the network will cause them to be fragmented. TCP attempts to avoid this, by making sure that the segment sent is always within size bounds. Of course, there may be different link types on route from source to destination so there may be no one best segment size. At the network layer, the largest frame that can be transmitted is called the maximum transmission unit (MTU). For Ethernet, this is commonly 1500 bytes. The MSS would therefore be 1500 minus the size of the TCP header minus the size of the IP header, leaving 1460 bytes. This doesn't mean that the sender waits until there are 1460 bytes worth of messages to go out: the data sent also depends on the capability of the receiver to receive data. The policy of how to schedule the sending of data is actually quite open, and this is one aspect of TCP that means that it can be a bad fit for the NVE developer.

3.3.3 Reliability in TCP

To make a connection reliable, the buffer is not a like a queue: once the segment has been sent, it is not deleted, but the sender waits until it has been acknowledged by the remote host. If the data isn't acknowledged within a certain time, or the receiver specifically says that it is missing some data, then it is retransmitted. This is easier to explain by first introducing the layout of a TCP segment, as shown in Figure 3.10.

This is considerably more complicated than the UDP segment layout. The meaning of the source port and destination port are the same as in the UDP. The main new details are the *Sequence Number* and *Acknowledgement Number*. These are 32 bits numbers and they are used to provide reliability. The *Data Offset* is an offset within this segment that the data can be found if options are used. There are eight *Flags* which we explain in more detail below. The *Receive Window* is used for flow

Bits	0		15	16 31
0–31	Source Port			Destination Port
32–63	Sequence Number			
64–95	Acknowledgement Number			
96–127	Data Offset	Not Used	Flags	Receive Window
128–159	Checksum			Urgent Pointer
160–191	Options (Optional)			
160+ 192+, 224+, etc.	Data			

FIGURE 3.10

Layout of a TCP segment

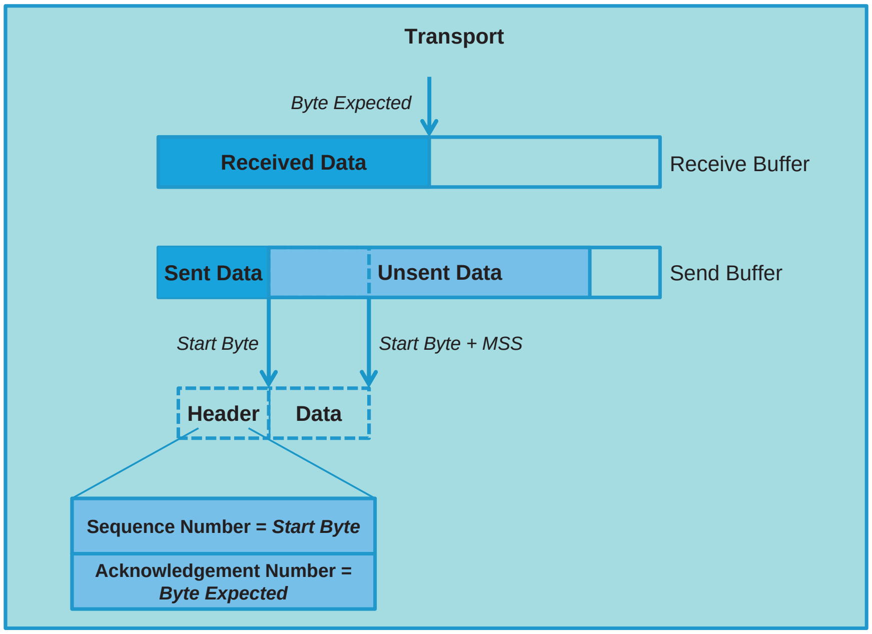


FIGURE 3.11

Sequence and acknowledgement numbers

control; see below. The *Checksum* performs the same role as in UDP and provides a simple check of the segment's integrity. The *Urgent Pointer* is used to indicate any urgent data if the segment is flagged as urgent; see below.

First, we'll discuss reliability. In the transport layer, for each connection there is a pair of buffers: both a send buffer and a receive buffer. Every time a segment is sent, the sender puts into the segment header, a sequence number for the outgoing data, plus the next expected sequence number, an *Acknowledgement Number*, from its corresponding process at the other end of the connection. The sequence numbers and acknowledgement numbers are counted in bytes. If one segment is sent with sequence number N and data size M then the next segment will have the sequence number $N + M$. This is illustrated in Figure 3.11. In practice, when a particular connection is set up the first acknowledgement number that is used is chosen to be random.

To implement reliability, the sender keeps track of the last acknowledgement number it has received from its peer. If it doesn't get an acknowledgement within a certain period of time, it resends the packet. Consider a simple example, where a single segment is sent, and confirmed by the recipient as illustrated in Figure 3.12. The data to be sent is packaged up and sent in a segment to the other host. The sender keeps track of the last acknowledged sequence number (M). The segment contains the sequence number which is M. Last acknowledged is an index on the send buffer *before* which all data have been acknowledged. As the segment is received, in

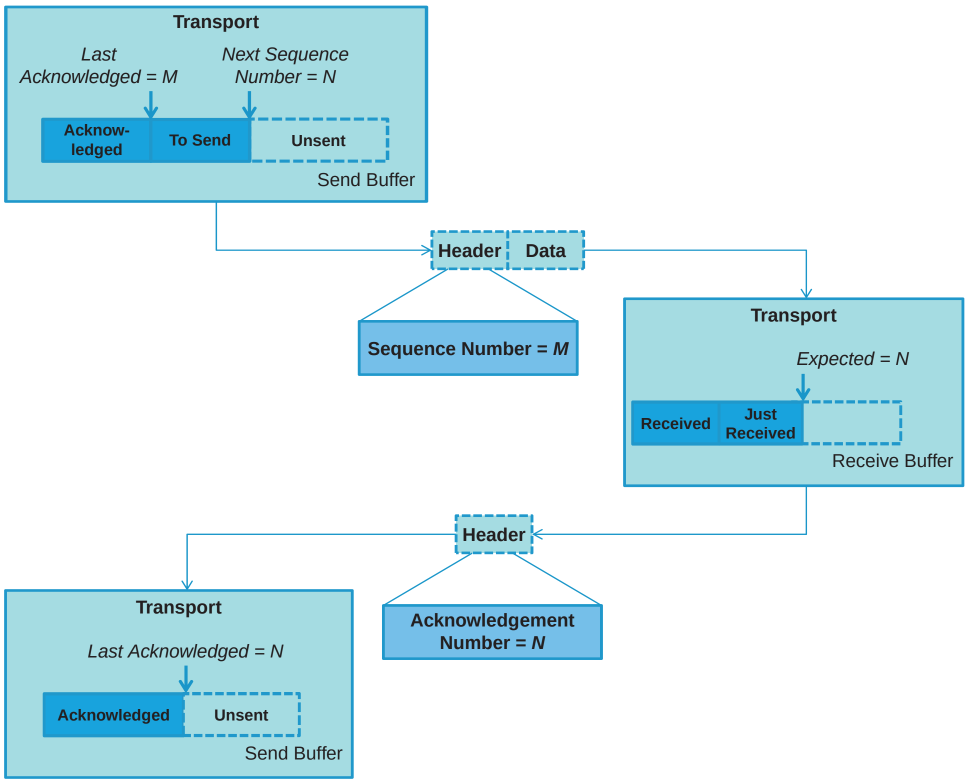


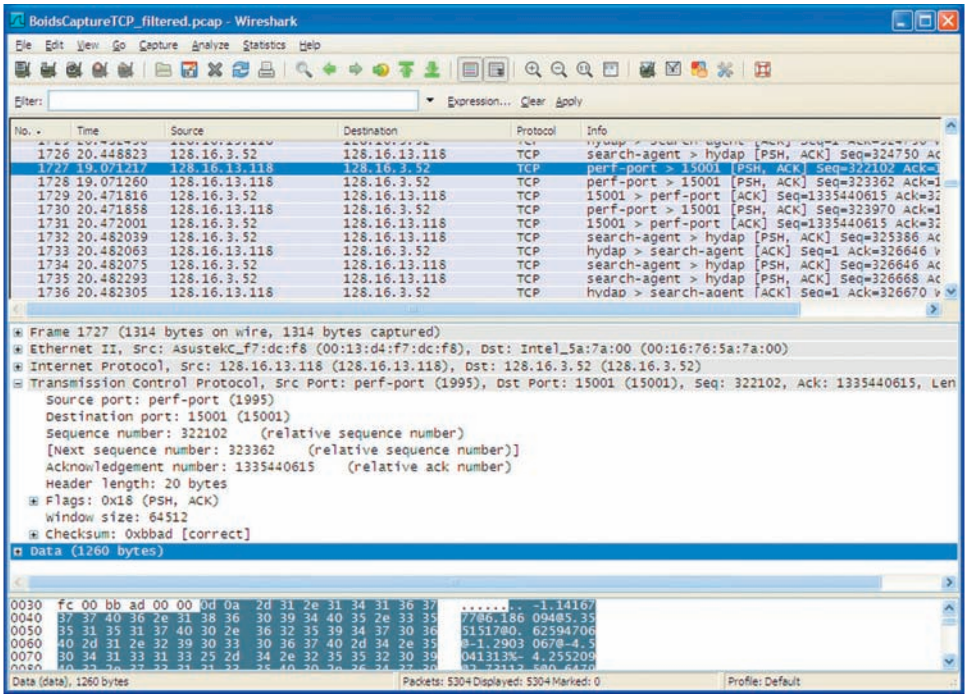
FIGURE 3.12

Acknowledgement of a segment's reception

the receive buffer, the new data are inserted into the buffer, and the next expected sequence number is set to *N*. An acknowledgement is then returned to the sender, containing the acknowledgement number *N*. The sender can then move its last acknowledged up to *N*, since all data before this are now acknowledged.

In practice, the sender doesn't have to wait for an acknowledgement to keep sending segments; it can send several segments and wait for one or more acknowledgements. The receiver doesn't need to confirm every single segment as a packet with acknowledge number *N* confirms *everything* before *N*. This is known as a *cumulative acknowledgement*. Let's look at the TCP version of the boids application with some of the Wireshark logs.

Recall from Chapter 2 that in the TCP implementation although each connection is bi-directional we choose to have each application open its own server port, and thus there are two TCP connections between the two hosts. Figure 3.13 shows a part of the Wireshark log that contains frames traveling in both directions. Frames 1727–1731 are related to one of the connections, Frames 1726 and 1732–1736 concern the other. We've started the process on the same two machines as before: seychelles.cs.ucl.ac.uk


BoidsCaptureTCP_filtered.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
No.  Time      Source         Destination    Protocol Info
1726 20.448823 128.16.3.52    128.16.13.118  TCP      search-agent > hydap [PSH, ACK] Seq=324750 Ac
1727 19.071217 128.16.13.118  128.16.3.52    TCP      perf-port > 15001 [PSH, ACK] Seq=322102 Ack=1
1728 19.071260 128.16.13.118  128.16.3.52    TCP      perf-port > 15001 [PSH, ACK] Seq=323362 Ack=1
1729 20.471816 128.16.3.52    128.16.13.118  TCP      15001 > perf-port [ACK] Seq=1335440615 Ack=32
1730 20.471858 128.16.13.118  128.16.3.52    TCP      perf-port > 15001 [PSH, ACK] Seq=323970 Ack=1
1731 20.472001 128.16.3.52    128.16.13.118  TCP      15001 > perf-port [ACK] Seq=1335440615 Ack=32
1732 20.482039 128.16.3.52    128.16.13.118  TCP      search-agent > hydap [PSH, ACK] Seq=325386 Ac
1733 20.482063 128.16.13.118  128.16.3.52    TCP      hydap > search-agent [ACK] Seq=1 Ack=326646 W
1734 20.482075 128.16.3.52    128.16.13.118  TCP      search-agent > hydap [PSH, ACK] Seq=326646 Ac
1735 20.482293 128.16.3.52    128.16.13.118  TCP      search-agent > hydap [PSH, ACK] Seq=326668 Ac
1736 20.482305 128.16.13.118  128.16.3.52    TCP      hydap > search-agent [ACK] Seq=1 Ack=326670 W

⊞ Frame 1727 (1314 bytes on wire, 1314 bytes captured)
⊞ Ethernet II, Src: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8), Dst: Intel_5a:7a:00 (00:16:76:5a:7a:00)
⊞ Internet Protocol, Src: 128.16.13.118 (128.16.13.118), Dst: 128.16.3.52 (128.16.3.52)
⊟ Transmission Control Protocol, Src Port: perf-port (1995), Dst Port: 15001 (15001), Seq: 322102, Ack: 1335440615, Len
    Source port: perf-port (1995)
    Destination port: 15001 (15001)
    Sequence number: 322102    (relative sequence number)
    [Next sequence number: 323362    (relative sequence number)]
    Acknowledgement number: 1335440615    (relative ack number)
    Header length: 20 bytes
  ⊞ Flags: 0x18 (PSH, ACK)
    Window size: 64512
  ⊞ Checksum: 0xbbad [correct]
⊞ Data (1260 bytes)

0030  fc 00 bb ad 00 00 0d 0a  2d 31 2e 31 34 31 36 37   ........ -1.14167
0040  37 37 40 36 2e 31 38 36  30 39 34 40 35 2e 33 35   77@6.186 094@5.35
0050  35 31 35 31 37 40 30 2e  36 32 35 39 34 37 30 36   51517@0. 62594706
0060  40 2d 31 2e 32 39 30 33  30 36 37 40 2d 34 2e 35   @-1.2903 067@-4.5
0070  30 34 31 33 31 33 25 2d  34 2e 32 35 35 32 30 39   041313%- 4.255209

Data (data), 1260 bytes    Packets: 5304 Displayed: 5304 Marked: 0    Profile: Default
```

FIGURE 3.13

Wireshark log from the TCP version of the boids application

(with IP address 128.16.3.52) and narok.cs.ucl.ac.uk (128.16.13.11). We started the Java TCP boids process on narok first with the following command line:

```
java -jar boidsTCP.jar 10 128.16.3.52 15000 15001
```

and on seychelles:

```
java -jar boidsTCP.jar 10 128.16.13.118 15001 15000
```

The TCP source ports were chosen at random again, and the full situation is discussed in Figure 3.14. Compare this to Figure 3.8.

Now we look at the sequence of segments in frames 1727–1731. We've summarized the data in Table 3.2. We can see three segments with data going from narok to seychelles and two segments returning. The returning segments carry no data; they are pure acknowledgements as we are only using this connection in a unidirectional manner. This means that the acknowledgement numbers on all frames from narok to seychelles are the same (1335440615). Although there is no data, seychelles returns a sequence number (1335440615). The important part about the two return segments is that one of the flags is set, ACK, see below. This means that the sender (narok) can now move its last acknowledged marker up. We can also see

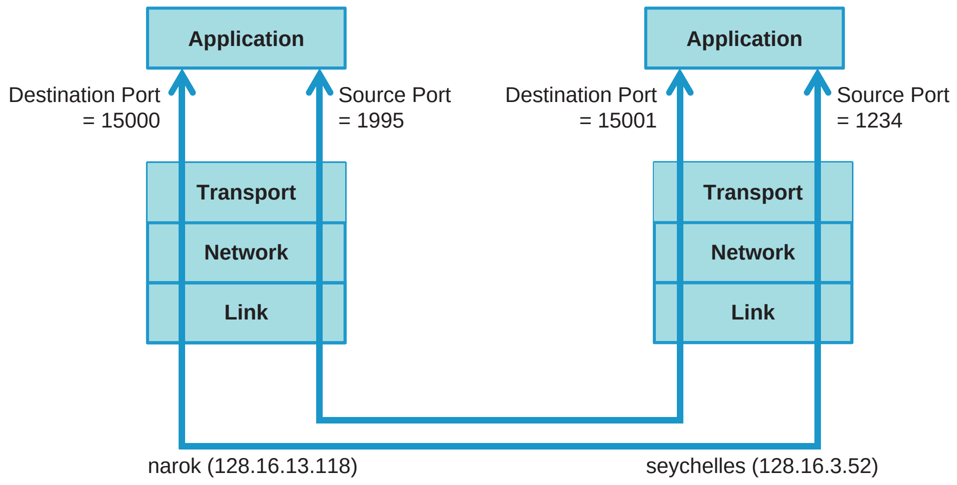


FIGURE 3.14

IP address and port numbers for the example of the TCP boids application

Table 3.2 Summary of Sequence of Frames 1727–1731

Frame	Source/Port	Destination/ Port	Sequence Number	Acknowledgement Number	Data Size
1727	128.16.13.118 1995	128.16.3.52 15001	322102	1335440615	1260
1728	128.16.13.118 1995	128.16.3.52 15001	323362	1335440615	608
1729	128.16.3.52 15001	128.16.13.118 1995	1335440615	323362	0
1730	128.16.13.118 1995	128.16.3.52 15001	323970	1335440615	2
1731	128.16.3.52 15001	128.16.13.118 1995	1335440615	323972	0

how the sequence numbers work, as well as the cumulative acknowledgement system. Frame 1727 has sequence number 322102 and data size 1260. This means that next sequence number would be 3233612. We can see this number as the sequence number of frame 1728 and the acknowledgement number in frame 1729. Frame 1729 acknowledges frame 1727, not frame 1728. Frame 1728 contained 608 bytes, so frame 1730, which only contains 2 bytes, has sequence number 323970. Frame 1731 has the acknowledgement number 323972 which is the sequence number of the segment which will come after 1730 (323970 plus 2). Frame 1731 acknowledges both frames 1728 and 1730.[5]

In practice, with data flowing in both directions along connections, acknowledgements are piggy-backed onto segments with data in. This means, at least at first glance, that if there is data flowing both ways, it is appropriate to use one bi-directional connection and not two uni-directional connections.

What would happen now if frames are lost? Either the segment with data would get lost on the way from sender to receiver or an acknowledgement segment would get lost. In either case, the sender doesn't receive the acknowledgement. The receiver can't know that it was expecting a segment nor that the acknowledgement it sent was lost. All that the sender needs to do is resend the segment after a timeout. To explain this, consider a situation where all data are sent and acknowledged. Thus the last acknowledged number (N) will equal the next possible sequence number. If application data is received, a segment is created with this sequence number (N). As this segment is dispatched, a timer is started with a period T_{timeout}. If no acknowledgement is received within T_{timeout}, then the segment with sequence number N is resent. If an acknowledgement is received, if further segments have been sent, then the timer is restarted.

Some scenarios for lost segments are shown in Figures 3.15–3.17. We've assumed that each segment contains 100 bytes of data.

The length of T_{timeout} needs to be chosen so that it is longer than the normal *round-trip time* (RTT) for communication for this connection. We will be investigating RTT in more detail in later chapters, especially when discussing latency (Chapter 10). An estimate of RTT is made independently by both hosts in the connection. Each takes the time between sending a segment and it is being acknowledged. Because RTT will change over time, the estimated RTT is an average. Given the average and the standard deviation, the T_{timeout} would be set to estimated RTT plus four times the standard deviation.

Retransmission, as described, works fine for single segments, but we have already suggested that multiple segments might be sent before awaiting confirmation. If only one segment is lost in a sequence, then the source doesn't have to rewind to the lost segment and retransmit all the subsequent segments, although this would work. Most TCP implementations thus support *selective acknowledgements* (SACK), whereby only certain segments are retransmitted. Selective acknowledgements are one use of the optional fields in the TCP header.

In summary, reliability is achieved, but at the expensive of retransmits after a certain period of time. One can thus expect that over long distances, RTT may vary quite significantly, and thus if segments are lost, retransmission might take a significant amount of time.

[5] Note that a segment has not been lost somewhere here: there almost certainly was no acknowledgement sent by the receiver in response to frame 1728. This is because TCP implementations are recommended to be a little lazy in sending acknowledgements: they wait 500 ms, and if segments are received within this time, they confirm every other segment. See the recommendations in RFC 2581.

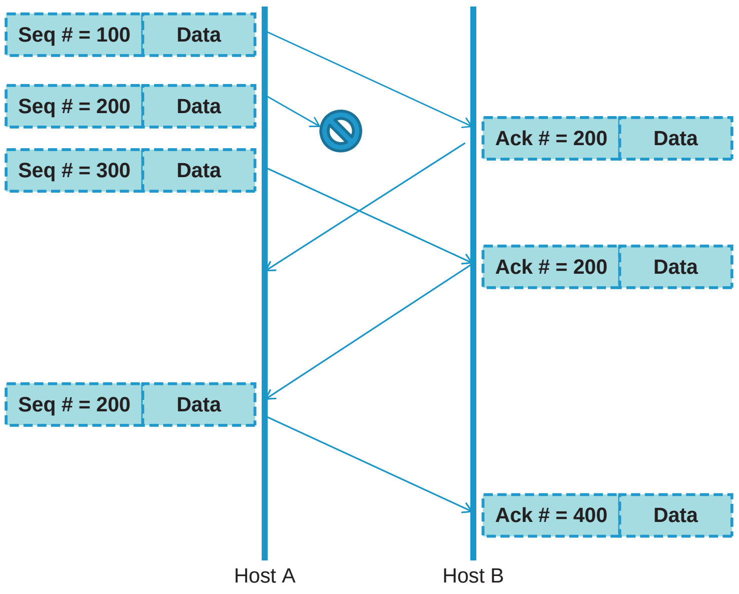


FIGURE 3.15

A lost data segment can be noticed because a duplicate ACK is received with acknowledgement number 200

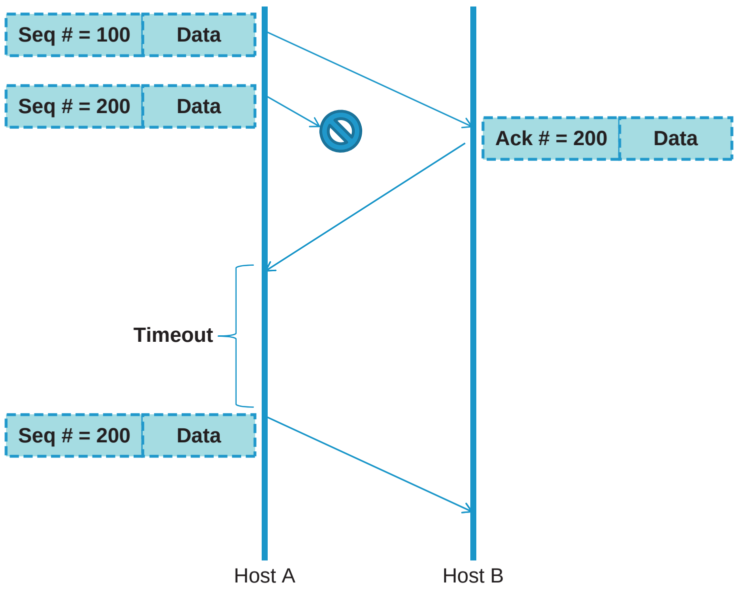


FIGURE 3.16

A lost segment on the tail-end of a series of segments means that the timeout will be triggered and the segment resent

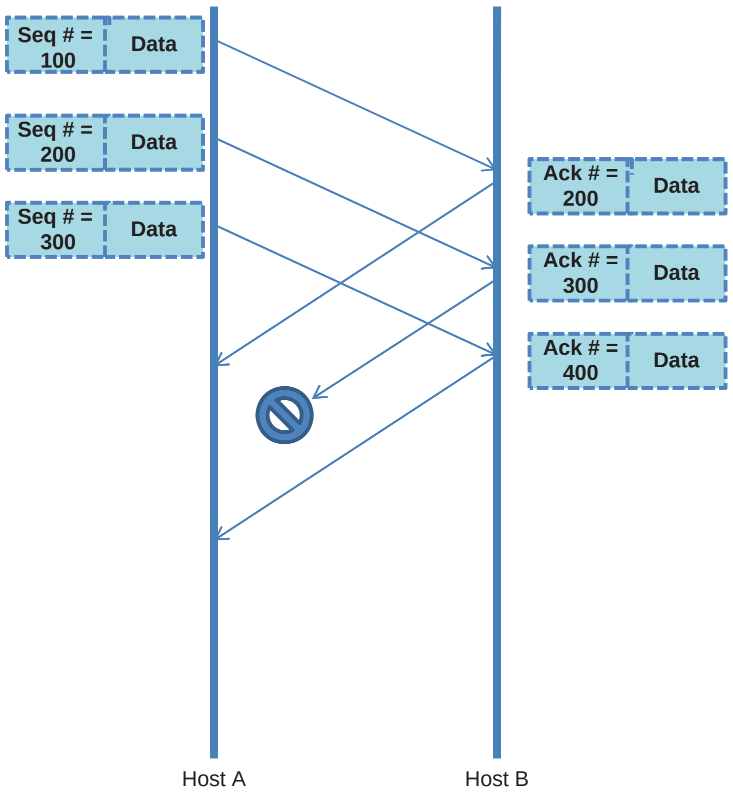


FIGURE 3.17

A lost acknowledgement packet doesn't trigger a resend because of the cumulative acknowledgement of the subsequent segment which arrives within the timeout period

3.3.4 Opening and closing TCP connections

Another key difference between UDP and TCP is that TCP requires some time to set up. There is a three-way handshake between the client and the server when setting up. Before explaining this, let's delve in more depth into the flags in the TCP header as shown in Figure 3.10. Bits 104–111comprise eight flags:

- CWR: Congestion Window Reduced.
- ECE: indicates that the TCP peer is Explicit Congestion Notification (ECN)-capable.
- URG: indicates that the URGent pointer field is significant.
- ACK: indicates that the ACKnowledgement field is significant.
- PSH: push function.
- RST: reset the connection.
- SYN: synchronize sequence numbers.
- FIN: no more data from sender.

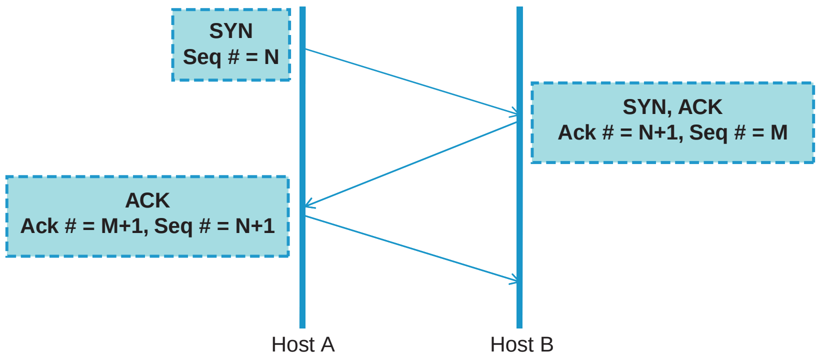


FIGURE 3.18

Three-way handshake to set up a TCP connection

The meanings of CWR and ECE, which are concerned with notifications from the hosts about congestion, are beyond the scope of this introduction. However, they are not compulsory in implementations. URG is not commonly used, but indicates that the receiver should pass some data (marked by the Urgent Pointer field) to the application immediately. Similarly, PSH indicates that the whole data should be handled immediately. If we look back to Figure 3.13, we can see that every segment with data in it has been flagged with PSH. ACK would be set in any segment where the acknowledgement field is relevant. In Figure 3.13, all the segments have the ACK flag set. The flags RST, SYN and FIN are used in the setting up and tearing-down of a connection.

Setup is done with a three-way handshake of SYN,[6] SYN/ACK and then ACK. Figure 3.18 shows this sequence. Note how the acknowledgement and sequence numbers are set up. Host A must choose a sequence number (N) to start, but Host B doesn't know this yet; thus the SYN flag means that the receiver buffer can be allocated with the expected next sequence of $N + 1$. SYN and ACK together on the return, acknowledge the sequence number of Host A, and set up the sequence number of Host B. The final segment ACK confirms that Host A has the correct acknowledge number ($M + 1$) to Host B's sequence number.

The closing of a TPC connection is done as shown in Figure 3.19. This is the normal four-way handshake to close a connection. A connection can be half-open, meaning that only one host can read information from it. This uses the RST flag and is commonly sent when one process terminates abruptly.

Let's look at setup in the TCP boids application. Figure 3.20 shows us that frame 62 contains a SYN segment from 128.16.3.52 (seychelles) to 128.15.13.118 (narok).

[6] You may have heard of SYN in the context of SYN-flooding, a type of network attack where a hostile entity tries to crash a server by sending lots of SYNs from different clients: each SYN requires the server to send a SYN,ACK but more importantly, it uses resources (buffers, etc.) on the server. See RFC 4987, TCP SYN Flooding Attacks and Common Mitigations.

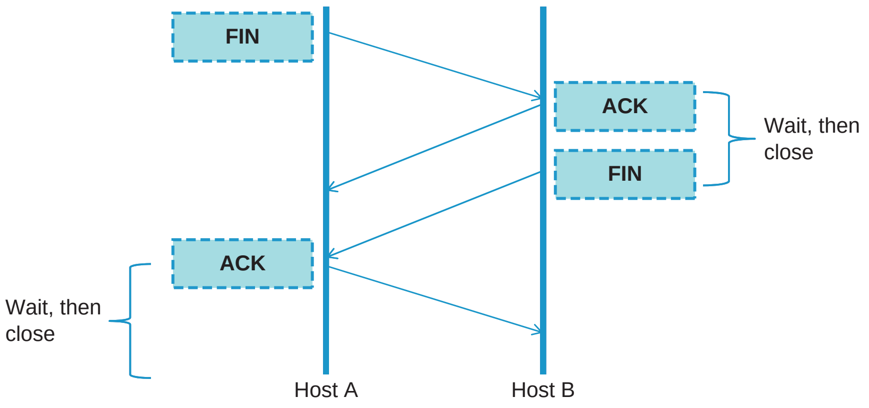
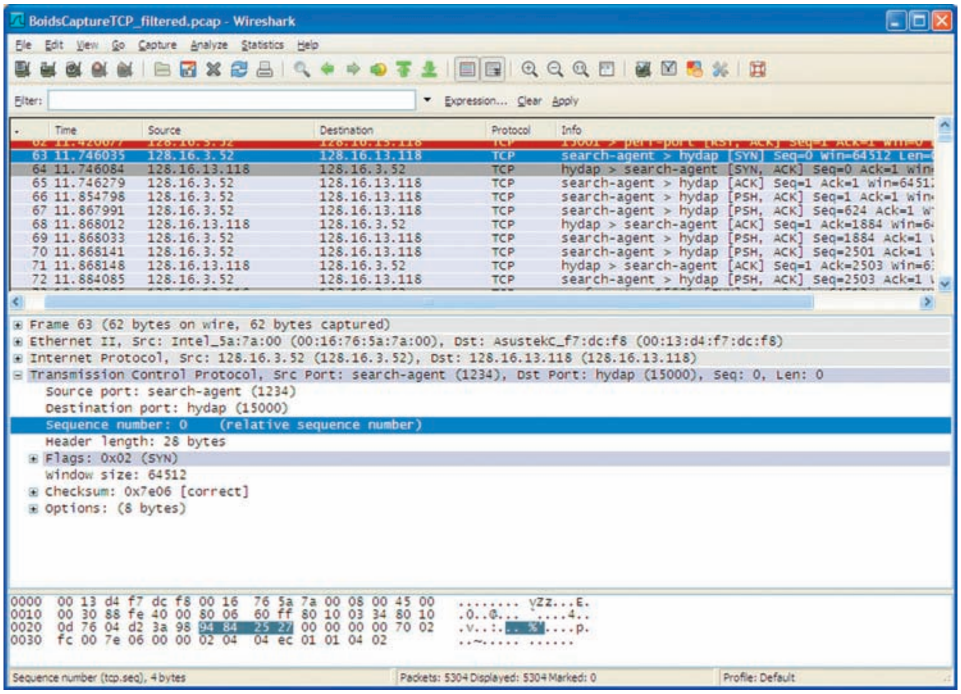


FIGURE 3.19

Closing a TCP connection

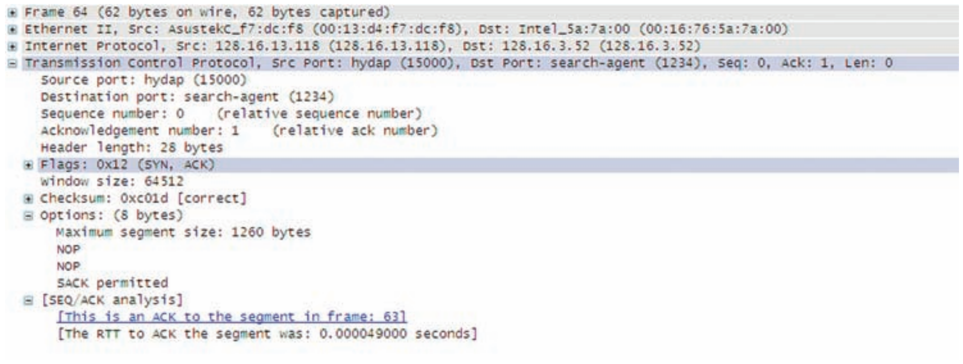

BoidsCaptureTCP_filtered.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
      Time       Source          Destination     Protocol  Info
   63 11.746035  128.16.3.52     128.16.13.118   TCP       search-agent > hydap [SYN] Seq=0 Win=64512 Len=
   64 11.746084  128.16.13.118   128.16.3.52     TCP       hydap > search-agent [SYN, ACK] Seq=0 Ack=1 Win
   65 11.746279  128.16.3.52     128.16.13.118   TCP       search-agent > hydap [ACK] Seq=1 Ack=1 Win=6451
   66 11.854798  128.16.3.52     128.16.13.118   TCP       search-agent > hydap [PSH, ACK] Seq=1 Ack=1 Win
   67 11.867991  128.16.3.52     128.16.13.118   TCP       search-agent > hydap [PSH, ACK] Seq=624 Ack=1 W
   68 11.868012  128.16.13.118   128.16.3.52     TCP       hydap > search-agent [ACK] Seq=1 Ack=1884 Win=6
   69 11.868033  128.16.3.52     128.16.13.118   TCP       search-agent > hydap [PSH, ACK] Seq=1884 Ack=1 
   70 11.868141  128.16.3.52     128.16.13.118   TCP       search-agent > hydap [PSH, ACK] Seq=2501 Ack=1 
   71 11.868148  128.16.13.118   128.16.3.52     TCP       hydap > search-agent [ACK] Seq=1 Ack=2503 Win=6
   72 11.884085  128.16.3.52     128.16.13.118   TCP       search-agent > hydap [PSH, ACK] Seq=2503 Ack=1 

⊞ Frame 63 (62 bytes on wire, 62 bytes captured)
⊞ Ethernet II, Src: Intel_5a:7a:00 (00:16:76:5a:7a:00), Dst: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8)
⊞ Internet Protocol, Src: 128.16.3.52 (128.16.3.52), Dst: 128.16.13.118 (128.16.13.118)
⊟ Transmission Control Protocol, Src Port: search-agent (1234), Dst Port: hydap (15000), Seq: 0, Len: 0
    Source port: search-agent (1234)
    Destination port: hydap (15000)
    Sequence number: 0    (relative sequence number)
    Header length: 28 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 64512
  ⊞ Checksum: 0x7e06 [correct]
  ⊞ Options: (8 bytes)

0000  00 13 d4 f7 dc f8 00 16  76 5a 7a 00 08 00 45 00   ........ vZz...E.
0010  00 30 88 fe 40 00 80 06  60 ff 80 10 03 34 80 10   .0..@... `....4..
0020  0d 76 04 d2 3a 98 94 84  25 27 00 00 00 00 70 02   .v..:... %'....p.
0030  fc 00 7e 06 00 00 02 04  04 ec 01 01 04 02         ..~..... ......

Sequence number (tcp.seq), 4 bytes    Packets: 5304 Displayed: 5304 Marked: 0    Profile: Default
```

FIGURE 3.20

First step in the setup of a connection with a SYN segment

Note in the middle pane that the sequence number is shown as 0. Wireshark helpfully counts from 0, no matter what the actual number in the segment is. We've highlighted the sequence number field in the middle pane, and in the bottom pane, we can see highlighted the actual sequence number, 0x94822527.



```
⊞ Frame 64 (62 bytes on wire, 62 bytes captured)
⊞ Ethernet II, Src: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8), Dst: Intel_5a:7a:00 (00:16:76:5a:7a:00)
⊞ Internet Protocol, Src: 128.16.13.118 (128.16.13.118), Dst: 128.16.3.52 (128.16.3.52)
⊟ Transmission Control Protocol, Src Port: hydap (15000), Dst Port: search-agent (1234), Seq: 0, Ack: 1, Len: 0
    Source port: hydap (15000)
    Destination port: search-agent (1234)
    Sequence number: 0    (relative sequence number)
    Acknowledgement number: 1    (relative ack number)
    Header length: 28 bytes
  ⊞ Flags: 0x12 (SYN, ACK)
    Window size: 64512
  ⊞ Checksum: 0xc01d [correct]
  ⊟ Options: (8 bytes)
      Maximum segment size: 1260 bytes
      NOP
      NOP
      SACK permitted
  ⊟ [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 63]
      [The RTT to ACK the segment was: 0.000049000 seconds]
```

FIGURE 3.21

SYNACK segment

The following frame, 64, contains the SYN/ACK flags. We show the middle pane details for this frame in Figure 3.21. Wireshark confirms this is the response to frame 63, and it contains the acknowledgement number 1. Out of interest, we note that frames 63 and 64 both also have the TCP optional field set. In both frames the header length is 28 bytes, not 24 bytes. Both happen to set the same options, that is the maximum segment will be 1260 and that both clients support selective acknowledgement (SACK permitted).

Frame 65 contains an ACK, with sequence number 1 and acknowledgement number 1. This completes the setup. Frame 66 contains the first data for this connection. This setup process thus takes an RTT to complete before the first data can be delivered.

3.3.5 Flow control and congestion avoidance in TCP

TCP provides two further advantages over UDP: flow control and congestion control. Both assist with ensuring that data transfer is reliable, and they do this by slowing down the sending process.

Flow control is a mechanism to attempt to ensure that the sending process doesn't send more data than the receiving client has space for in its receive buffer. Each TCP segment includes a *Receive Window*. This is the amount of data that the receiver is willing to accept. If an application process is stalled or busy, then this might decrease, and if it reaches zero, the sender must only send segments with one byte of data, until receive window rises above zero.

In our example of the boids application, the two machines were lightly loaded, and almost continuously showed that the receive window was 64,512 bytes. Very occasionally this is lower, but no lower than 63,800 bytes.

Despite the receiver being able to accept data up to receive window in size, the network might not be able to transport this amount of data at an appropriate rate. But what is an appropriate rate at which to send data? One of the key insights of TCP is that this can't be measured from the network, but that we can seek the correct rate by careful examination of failures to acknowledge packets. First we might guess that if we always receive an ACK for each segment we send, we can probably increase the rate at which we send. If we don't get the ACK, then we can suspect congestion and slow down. In TCP this is implemented with a *congestion window* on the sending side. This window, a multiple of the MSS, is the total size of unacknowledged segments that can be "in flight". If we assume that the RTT was constant, this would mean that (congestion window size/RTT) bytes/s data would be sent. Increasing and decreasing the size of the congestion windows effectively increases and decreases the bytes/s transmitted over the connection. At the core of the avoidance process is a scheme known as *additive-increase, multiplication decrease*. There are a few variations, but one common implementation is to increase congestion window by MSS bytes every RTT, but halve it if a loss is detected, see Figure 3.22. Thus the rate goes up linearly. The congestion window increases slowly, but decreases rapidly. Thus TCP connections show a noticeable saw-tooth pattern in their transmission rate; this in turn leads to the rather odd situation you may have experienced when you are downloading a large file over HTTP or FTP: whenever you are watching, the "Time Remaining" seems to be decreasing faster than the clock is ticking, but when you aren't looking, the "Time Remaining" goes up considerably!

This linear increase phase is undesirable in situations where the rate can actually be quite high, such as local networks. Thus TCP implementations frequently implement a *slow start* where the congestion window goes up by MSS for *every* acknowledgement (not for every RTT period). This gives an exponential increase, which will fail quickly: at this point the congestion window is reduced and the linear phase entered. There are many variants on this in different operating systems. The general

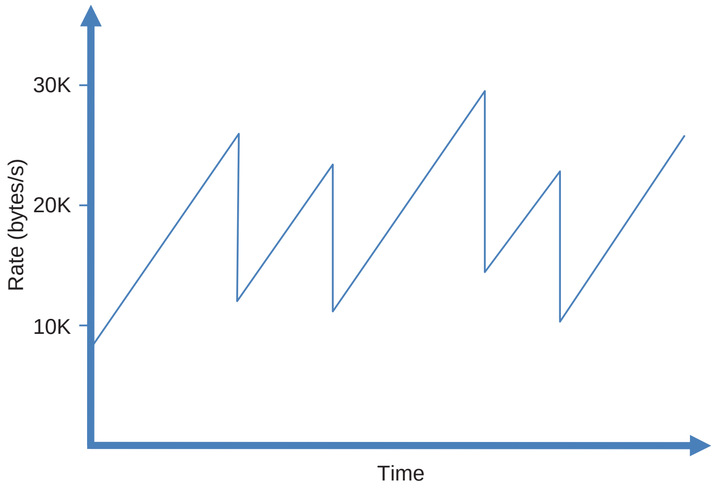


FIGURE 3.22

Saw-tooth behavior of the rate of transmission in a TCP connection principle that TCP implementation and other reliable protocols commonly adhere to is *TCP fairness*: that if there are N TCP connections over a link, they would all eventually achieve $1/N$ of the bandwidth of that connection.

3.4 NETWORK LAYER

The main responsibility of the network layer is to route packets from the sending host to receiving host. As discussed in Chapter 1, the Internet is a packet-switching network, and it interoperates between different link layers. We will be discussing IP version 4, though IP version 6 is already being deployed.

Thus the principle problem that must be solved, aside from finding the receiver, is the fragmentation of IP packets into multiple parts. We've noted that a common MTU is 1500 bytes for Ethernet; however other link layers have different sizes. 802.11 has a MTU of 2272, FDDI (fiber distributed data interface) has a MTU 4500.

A router *forwards* packets from one link to another, see Figure 3.23. An arriving IP packet is matched against a *route table* or *routing table* inside the router, and then forwarded to one or more output links. Typically each packet is sent on one link, but IP supports multicast and broadcast routing as well, see Section 3.6. *Routing* is the cumulative process of forwarding via multiple routers.

Routing is a complex process and configuration of routing is one of the key skills in network traffic engineering. The way in which routing occurs depends on the scale of the network. At the scale of a small office, or home network, routing is usually very simple. Every computer is on a LAN, and there is one *gateway* which connects this network to the outside world. Physically, this gateway is probably provided by the ISP that the home or office subscribes to. Each machine on the LAN configures this gateway in their IP options and this router routes all packets to the ISP connection. The box from the ISP might perform several other functions such as wireless networking, firewall and proxy, but routing is the most important function.

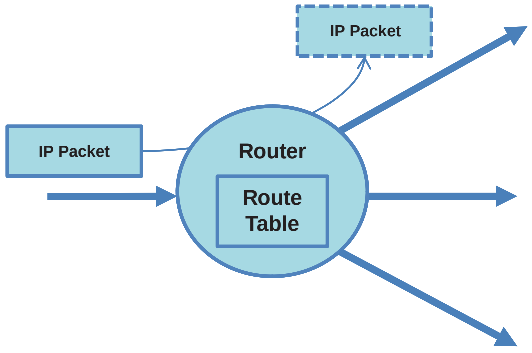


FIGURE 3.23

A router forwards IP packets from an incoming link to an outgoing link

Routing across larger scales, say a large office, will require a more complex network. This may be because the underlying link layer can't support the necessary number of machines or because there is so much traffic that the network must be subdivided to avoid congestion. In this case the network might have several routers, connecting together several *subnets*. Each router's router table needs to know, given an IP address, which subnet it is on and thus which link to send it. Each router could keep a list of the link for every IP address on the network, and every time a packet arrived, it could exhaustively search that list. However, with a little preplanning the routing can be made much more efficient with the use of subsequences of IP addresses.

Recall that IPv4 addresses are 32 bits in four octets. The machine narok at UCL has the IP address 128.16.13.118, see Figure 3.31 for the full settings for the IPv4 protocol. Amongst the settings for this machine will be a *netmask* 255.255.240.0. This netmask indicates which other machines are on the same subnet. In binary, narok's IP address is 10000000000100000000110101110110 (call this X), and the netmask is 11111111111111111111000000000000 (call this Y). If another machine, kivu's IP address is 128.16.68.182, then narok and kivu would be on the same network if X&Y (i.e. binary AND) is equal to X&Z. kivu's address in binary is 10000000 00010000010001001011 0110, and it isn't on the same subnet. Seychelles' IP address was 128.16.3.52 or 10000000000100000000001100110100 and it is on the same subnet as narok. Machines on the same subnet can communicate without sending packets to a router. If the machine isn't on the same subnet, then the default behavior for the link layer is to route the packet to the default gateway, or if the individual host has been configured with a different route, to a specific gateway. Another convention for describing subnets is the *classless interdomain routing* (CIDR). For the example of the subnet containing narok and seychelles, the CIDR is 128.16.0.0/20. The meaning of this is that on this subnet, all machines' IP addresses have the first 20 bits the same as 128.16.0.0. Thus there are 12 bits that are free to give a machine identifier on this subnet. The range of numbers would be 128.16.0.0–128.16.15.255. The machine kivu is on a different subnet (a subnet just for wireless connections), with CIDR 128.16.64.0/20 and thus this subnet could contain all hosts with IP addresses in the range 128.16.64.0–128.16.79.255.

Before discussing other aspects of routing, it is worth explaining how IP addresses are allocated. UCL Computer Science has the ability to assign IP addresses starting with the prefix 128.16 or with CIDR 128.16.0.0/16. This is somewhat unusual for a department of such size: there could be almost 65536 hosts on the network, whereas there are actually only a few thousand addresses allocated. Most of the addresses are thus unallocated. When a home user, a company or an institution wants to join the Internet they need to get at least one IP address. They get this from their ISP. The allocation of addresses is another process overseen by IANA and addresses are actually running out. Each home, office or lab needs one IP address in order to be able to be "visible" on the Internet, so that machines external can connect to the home, office or lab. Each ISP has a block of IP addresses, defined by one or more CIDR. If an ISP has a CIDR of the form X.X.X.X/21 bits, then it has

11 bits to allocate, and thus can provide 2^{11} or 2048 addresses. The ISP might sell these in chunks as *static IP addresses.* For a company running a web service, a static IP address is needed before a DNS entry can be placed, and the web service publicized.[7] For a home user needing only one address, this IP address will probably be a *dynamic IP*, allocated when the user connects. Many domestic services such as broadband are like this: an IP address is allocated when the broadband modem connects, and the user is not guaranteed to have the same IP address over long periods.

3.4.1 Network Address Translation

Another wrinkle is the discussion of IP addresses is the use of Network Address Translation (NAT) or Port Address Translation (PAT). Many readers will be aware of, or run a home network, with several machines, games consoles and devices connected to a gateway, but the ISP only allocates one IP address. Each machine on the home network has a *local IP address*. Often these have the form 192.168.*.* or 10.*.*.*.[8] Whenever a local machine connects or receives information from the larger Internet, the NAT rewrites the Internet packets. There are several ways it can do this depending on the NAT, and there are several variations. One example would be that if only one machine connects out, then the gateway rewrites the local IP on outgoing packets, replaces it with the gateway's global IP address and sends it to the ISP. As a packet comes in the router it replaces the global IP address with the local IP address and sends it to the local network. In this case, all the source and destination ports remain the same. This can work similarly if there is a small pool of local machines that are unlikely to connect at the same time, or even better, if there is a small pool of global IP addresses that the gateway can allocate. A more common situation is that the NAT will also perform PAT. In this case, as well as replacing the IP address it changes the port numbers.

If two machines (Host A with IP address L1 and Host B with IP address L2) on a local network with two different local IP addresses connect to the Internet, then the outside world only sees packets with one global IP address. Say Host A used ports P1, P2, P3, and Host B, P4, P5, P6. The gateway will change the port numbers on outgoing packets, so that all the packets that originated from Host A are different than those from Host B. Let the port number the gateway uses be Q1 … Q6. This means that even if the ports clash (e.g. P1 is equal to P4), the gateway on receiving one packet with a specific destination port will know which of Host A and Host B to send it to. Of course, there is a problem: the gateway can only know where to send an incoming packet if it first observed an outgoing packet and created the mapping.

[7] One can get around this with a *Dynamic DNS* service, but the use of such a service is often against the terms and conditions of the ISP.

[8] Along with the ranges 172.16.*.* –172.31.*.*, and 169.254.*.*, these ranges are can only be allocated to local, private networks and they can't be assigned as global IPs. If a host has one of these IP addresses, this indicates that the host is on a local network, and cannot access the Internet without the use of NAT.

Thus NAT is controversial in some regards because machines behind a NAT can't be the target of connection requests, at least not without extra configuration on the gateway. This causes complications in the configuration of all sorts of network applications: because clients behind NATs can only contact servers, they can't be the target of peer-to-peer requests. This makes everything from Skype sessions through certain peer–peer file sharing protocols to certain NVEs problematic to run, and they often have to run servers specifically to relay connection requests to those clients. NAT is engineered out in IPv6. The process of rewriting packets isn't fast; the UDP and TCP levels and the IP level include checksums which need to be rewritten. Thus NAT is a potential bottleneck.

The gateway can be configured with a table that maps which incoming ports should be forwarded to what machine. If one wanted to run a game server on a home network or LAN behind a NAT, it is thus possible. However, the configuration must be done on the gateway, and thus there isn't standard way of doing this. Many forums related to popular NGs contain such instructions. NAT is an important topic for NVEs as it can constrain the way that ports and connections are used, so we give a more detailed description in Chapter 10.

3.4.2 IP packets

Let's look at the actual format of an IP packet. The format is presented in Figure 3.24.

Some of the fields here we can guess: the *Source Address* and *Destination Address* are IP addresses, and these do not appear elsewhere in the protocol stack. The *Version* is a 4-bit number which for IPv4 would be set to 4. The *Header Length* is the number of 32-bit words in the header. This needs to be given because of the variable length options field. The *Type of Service* (TOS) had an original definition which gave the priority with which the packet should be dealt. This was rarely used in practice, so the bits have been used for *DiffServ*, see Section 3.6.2. The *Total*

Bits	0		15	16	31
0–31	Version	Header Length	Type of Service	Total Length	
32–63	Identification			Flags	Fragment Offset
64–95	Time to Live		Protocol	Header Checksum	
96–127	Source Address				
128–159	Destination Address				
160–191	Options (Optional)				
160+ 192+, 224+, etc.	Data				

FIGURE 3.24

Format of an IP packet

Length is the size of the packet in words. Being 16 bits, the maximum possible size is 65,536 bytes. The *Identification* field is used to identify fragments; see below. *Flags* is a 3-bit field also used with fragmentation; see below. *Fragment Offset* similarly is concerned with fragments. *Time to Live* (TTL) is an important field and is related to routing; we discuss it in more depth below. *Protocol* indicates which transport-layer protocol the packet represents; this is necessary because there is no other indication of which transport layer is used. The values in common use are:

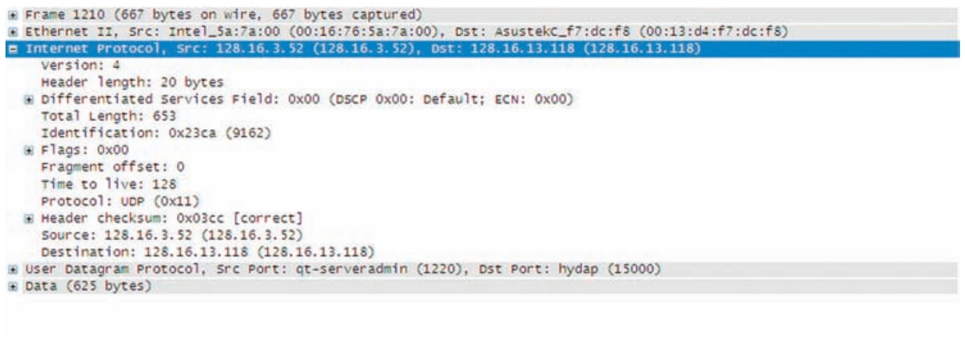
1: Internet Control Message Protocol (ICMP)
2: Internet Group Management Protocol (IGMP)
6: Transmission Control Protocol (TCP)
17: User Datagram Protocol (UDP)
89: Open Shortest Path First (OSPF)

We recognize the values 6 and 17 from earlier examples. We will discuss ICMP in Section 3.4.3 and IGMP in Section 3.6.1. OSPF is concerned with routing, and we discuss it in Section 3.4.4. The *Header Checksum* field only provides a degree of error detection for the headers on the IP packet. Each transport-layer protocol is expected to perform its own error detection on the data. Each router will check the checksum and discard the packet if it detects corruption.

There are two important concepts reflected in the headers for an IP packet: TTL and fragmentation. TTL is a mechanism which ensures that packets that “get lost” don’t clog up the network. It is possible for routing to give unreasonable routes, even circular routes for packets to a particular host. In the worst case, misconfigured routers might bounce a packet backwards and forwards between each other, taking up valuable resources on the routers and link. The TTL field is 8 bits. It is decremented by each router when the packet is forwarded. When the counter reaches 0, the packet is no longer forwarded, and the router will return an ICMP message (see below) to the sender. The number of routers that a packet must traverse is sometimes known as the *hop count*.

Fragmentation is necessary because individual packets may be too large for the MTU of the link layer that the packet is being sent on or being forwarded to. We already noted that the MTU of Ethernet is commonly 1,500 bytes. Internet routers must handle packets of 576 bytes, but commonly they handle much larger packets. Link layers might only support smaller packets but then the link layer must fragment and reassemble the packet, before it can be forwarded, which in turn might refragment it. Fragmentation makes sense because many application authors do not want to know the size of the packets that can traverse the network unfragmented. Doing it at the network layer is also appropriate because otherwise each transport layer protocol would have to implement a fragmentation system. However, we will note that in NVEs and other streaming applications, latency of packets is sometimes more important than reliability or throughput, and thus some sensitivity to likelihood of fragmentation is necessary.

When fragmentation is necessary all packets except that for the last fragment are sent with the flag *More Fragments* set. All fragment packets are sent with a *Fragment Offset* field, which is the byte offset in the original packet data that this



```
⊞ Frame 1210 (667 bytes on wire, 667 bytes captured)
⊞ Ethernet II, Src: Intel_5a:7a:00 (00:16:76:5a:7a:00), Dst: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8)
⊟ Internet Protocol, Src: 128.16.3.52 (128.16.3.52), Dst: 128.16.13.118 (128.16.13.118)
    Version: 4
    Header length: 20 bytes
  ⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 653
    Identification: 0x23ca (9162)
  ⊞ Flags: 0x00
    Fragment offset: 0
    Time to live: 128
    Protocol: UDP (0x11)
  ⊞ Header checksum: 0x03cc [correct]
    Source: 128.16.3.52 (128.16.3.52)
    Destination: 128.16.13.118 (128.16.13.118)
⊞ User Datagram Protocol, Src Port: qt-serveradmin (1220), Dst Port: hydap (15000)
⊞ Data (625 bytes)
```

FIGURE 3.25

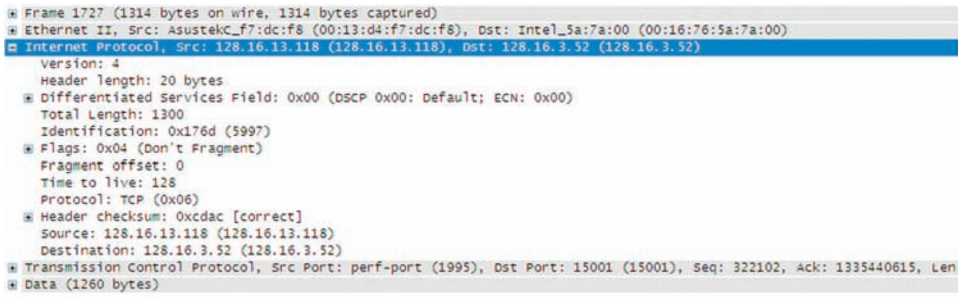
IP packet information for frame 1210 for the UDP boids demo

fragment contains. The fragments all have the same *Identification* field. On the receiving side, the original packet is reassembled, and is known to be complete when all the packets with their fragment offset "fill up" a continuous block of memory, where the last fragment didn't have the *More Fragments* flag set.

Let's now look at the IP packets from the boid application in Wireshark. Figure 3.25 shows the middle of the Wireshark window showing the same frame, 1210, of the UDP version of the boids application as shown in Figure 3.6 (UDP information) and Figure 3.8 (the addresses and ports used).

We can see that the IP version is 4. The header length is 20 bytes, and thus there are no options set. The diffserv fields are not set. The total length is 653 and the identification is 0x93cb. There are no flags set and no fragment offset. The TTL is 128, which means that it hasn't been decremented from its initial value. However this is consistent in this example where we now know that the two machines are on the same subnet. The protocol is 0x11 (17) and Wireshark helpfully indicates this is UDP. The header checksum is correct. The source and destination are 128.16.13.118 and 128.16.3.52. This all matches with our discussion in Section 3.2.1. The total length is 653 bytes, which correlates with the UDP segment length being 633 bytes (20 for the header).

Figure 3.26 shows the IP packet information for the TCP boids demo. This can be compared to Figure 3.13. The main difference we see is that the protocol is 0x06 (TCP). But we also see that the flag field is nonzero, in contrast to the UDP version. The flag 0x04 indicates that this packet should not be fragmented. This highlights an important difference between UDP and TCP. Recall that TCP will send information up to the MSS. This value is chosen specifically so that IP packets will not fragment: if the TCP connection was transporting large amounts of data, it is obviously more efficient to send packets that are as large as possible without causing fragmentation. Otherwise, for every TCP segment sent each would be split into two or more fragments, which would be unlikely to match the link layer's individual frame capacity. UDP has no such concept of connection and buffering, so each segment is handled independently.



```
⊞ Frame 1727 (1314 bytes on wire, 1314 bytes captured)
⊞ Ethernet II, Src: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8), Dst: Intel_5a:7a:00 (00:16:76:5a:7a:00)
⊟ Internet Protocol, Src: 128.16.13.118 (128.16.13.118), Dst: 128.16.3.52 (128.16.3.52)
    Version: 4
    Header length: 20 bytes
  ⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 1300
    Identification: 0x176d (5997)
  ⊞ Flags: 0x04 (Don't Fragment)
    Fragment offset: 0
    Time to live: 128
    Protocol: TCP (0x06)
  ⊞ Header checksum: 0xcdac [correct]
    Source: 128.16.13.118 (128.16.13.118)
    Destination: 128.16.3.52 (128.16.3.52)
⊞ Transmission Control Protocol, Src Port: perf-port (1995), Dst Port: 15001 (15001), Seq: 322102, Ack: 1335440615, Len
⊞ Data (1260 bytes)
```

FIGURE 3.26

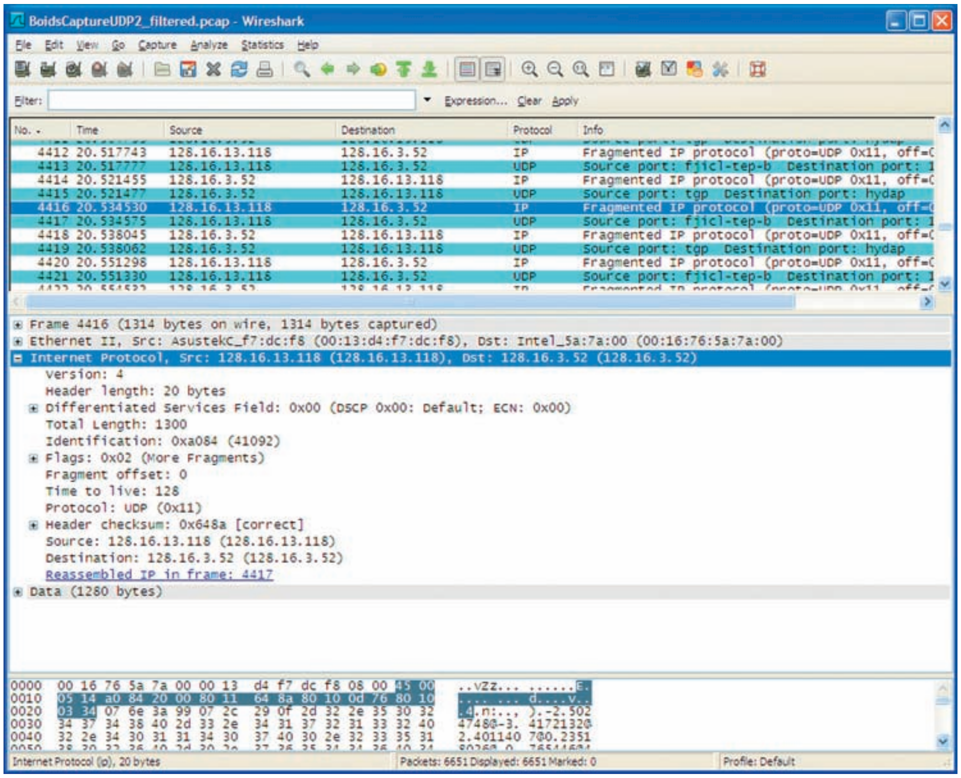
IP packet information for frame 1727 for the TCP boids demo

We can demonstrate this by looking at the difference in behavior when the UDP and TCP boids demonstrations start sending more information. Unlike the previous demonstrations in this chapter, we restart the processes with 30 boids on each side. This is still easily enough for real-time simulation and rendering, but now we need to send 180 (30x6) floating point numbers each frame.

Figure 3.27 shows a log from the UDP boids demo running with the same two machines as before. The source port numbers will be different as they are randomly chosen. Note that the main window shows interleaved IP packets and UDP segments. What is happening is that each UDP segment is fragmented into two IP packets. Frame 4416 is labeled as IP and note that in the middle pane, no application-level protocol is shown. The top frame labels frame 4417 as UDP. This is the frame that contains the last fragment of the original UDP segment. Wireshark has actually analyzed the IP fragment in frame 4416 and is suggesting in the middle pane information that this reassembled IP is in frame 4417. Note though that this information is a post-process, and the IP packet itself contains no information about when the next frame should be expected, it just indicates that there are more fragments. This indication is made with the flags field, which shows 0x02 (more fragments). We also need to note the identification field, 0xa084, which will be repeated any IP packets which contain the other fragments.

The middle pane information for frame 4417 is shown in Figure 3.28. Here we can see that the IP packet is a fragment of the same packet in frame 4416, as it has the same identification number, 0xa084. It also has a fragment offset, indicating that it isn't the first fragment. Note the offset, 1280, corresponds to the data length of frame 4416, which is also 1280. Because the offset is nonzero and the more fragments flag isn't set, this is the last fragment. Because it's the last fragment, the UDP segment can be reconstructed. The segment contains 1828 bytes of data, which would be too much for the link layer to transmit in a single frame.

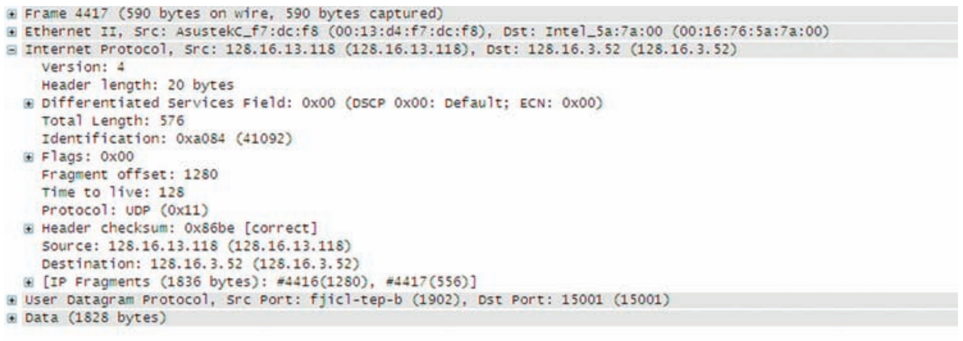
With a TCP connection, we don't see the packets fragmented in the same way. We see more packets being transmitted, with some of the packets having the maximum bye count allowed; see Figure 3.29.


BoidsCaptureUDP2_filtered.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
No. -   Time        Source          Destination     Protocol  Info
  4412 20.517743   128.16.13.118   128.16.3.52     IP        Fragmented IP protocol (proto=UDP 0x11, off=
  4413 20.517777   128.16.13.118   128.16.3.52     UDP       Source port: fjicl-tep-b  Destination port: 1
  4414 20.521455   128.16.3.52     128.16.13.118   IP        Fragmented IP protocol (proto=UDP 0x11, off=
  4415 20.521477   128.16.3.52     128.16.13.118   UDP       Source port: tgp  Destination port: hydap
  4416 20.534530   128.16.13.118   128.16.3.52     IP        Fragmented IP protocol (proto=UDP 0x11, off=
  4417 20.534575   128.16.13.118   128.16.3.52     UDP       Source port: fjicl-tep-b  Destination port: 1
  4418 20.538045   128.16.3.52     128.16.13.118   IP        Fragmented IP protocol (proto=UDP 0x11, off=
  4419 20.538062   128.16.3.52     128.16.13.118   UDP       Source port: tgp  Destination port: hydap
  4420 20.551298   128.16.13.118   128.16.3.52     IP        Fragmented IP protocol (proto=UDP 0x11, off=
  4421 20.551330   128.16.13.118   128.16.3.52     UDP       Source port: fjicl-tep-b  Destination port: 1

⊞ Frame 4416 (1314 bytes on wire, 1314 bytes captured)
⊞ Ethernet II, Src: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8), Dst: Intel_5a:7a:00 (00:16:76:5a:7a:00)
⊟ Internet Protocol, Src: 128.16.13.118 (128.16.13.118), Dst: 128.16.3.52 (128.16.3.52)
    Version: 4
    Header length: 20 bytes
  ⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 1300
    Identification: 0xa084 (41092)
  ⊞ Flags: 0x02 (More Fragments)
    Fragment offset: 0
    Time to live: 128
    Protocol: UDP (0x11)
  ⊞ Header checksum: 0x648a [correct]
    Source: 128.16.13.118 (128.16.13.118)
    Destination: 128.16.3.52 (128.16.3.52)
    Reassembled IP in frame: 4417
⊞ Data (1280 bytes)

0000  00 16 76 5a 7a 00 00 13  d4 f7 dc f8 08 00 45 00   ..vZz... ......E.
0010  05 14 a0 84 20 00 80 11  64 8a 80 10 0d 76 80 10   .... ... d....v..
0020  03 34 07 6e 3a 99 07 2c  29 0f 2d 32 2e 35 30 32   .4.n:.., ).-2.502
0030  34 37 34 38 40 2d 33 2e  34 31 37 32 31 33 32 40   4748@-3. 4172132@
0040  32 2e 34 30 31 31 34 30  37 40 30 2e 32 33 35 31   2.401140 7@0.2351

Internet Protocol (ip), 20 bytes    Packets: 6651 Displayed: 6651 Marked: 0    Profile: Default
```

FIGURE 3.27

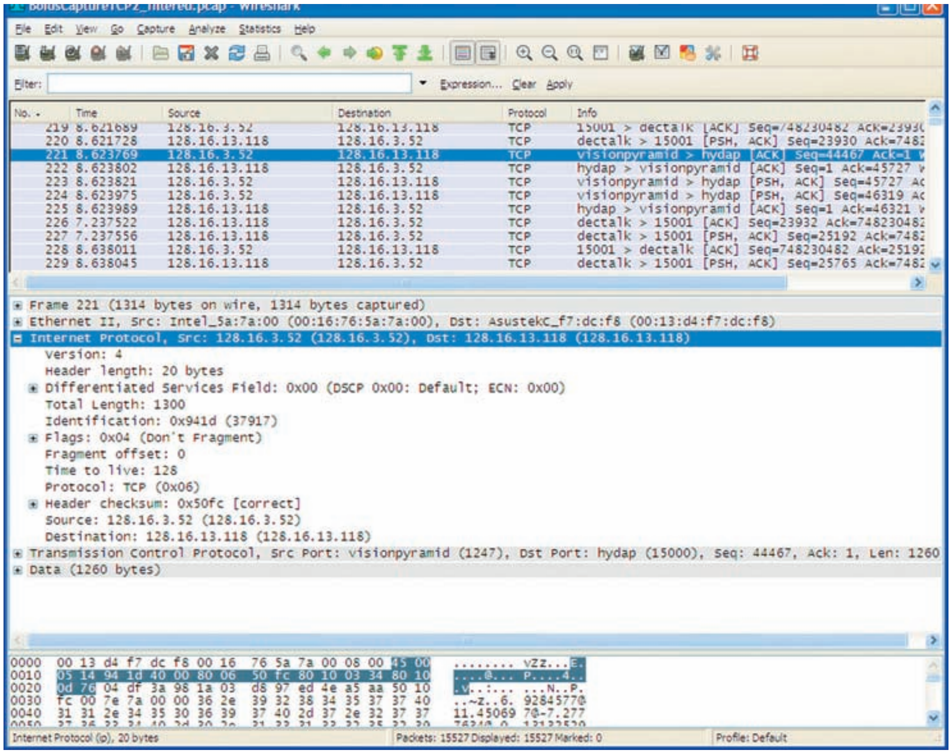
Fragmented IP packets for the UDP boids demo



```
⊞ Frame 4417 (590 bytes on wire, 590 bytes captured)
⊞ Ethernet II, Src: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8), Dst: Intel_5a:7a:00 (00:16:76:5a:7a:00)
⊟ Internet Protocol, Src: 128.16.13.118 (128.16.13.118), Dst: 128.16.3.52 (128.16.3.52)
    Version: 4
    Header length: 20 bytes
  ⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 576
    Identification: 0xa084 (41092)
  ⊞ Flags: 0x00
    Fragment offset: 1280
    Time to live: 128
    Protocol: UDP (0x11)
  ⊞ Header checksum: 0x86be [correct]
    Source: 128.16.13.118 (128.16.13.118)
    Destination: 128.16.3.52 (128.16.3.52)
  ⊞ [IP Fragments (1836 bytes): #4416(1280), #4417(556)]
⊞ User Datagram Protocol, Src Port: fjicl-tep-b (1902), Dst Port: 15001 (15001)
⊞ Data (1828 bytes)
```

FIGURE 3.28

Frame 4417 in the UDP boid demo



```
No.   Time      Source          Destination     Protocol  Info
  219 8.621689  128.16.3.52     128.16.13.118   TCP       15001 > dectalk [ACK] Seq=748230482 Ack=2393(
  220 8.621728  128.16.13.118   128.16.3.52     TCP       dectalk > 15001 [PSH, ACK] Seq=23930 Ack=7482
  221 8.623769  128.16.3.52     128.16.13.118   TCP       visionpyramid > hydap [ACK] Seq=44467 Ack=1 W
  222 8.623802  128.16.13.118   128.16.3.52     TCP       hydap > visionpyramid [ACK] Seq=1 Ack=45727 W
  223 8.623821  128.16.3.52     128.16.13.118   TCP       visionpyramid > hydap [PSH, ACK] Seq=45727 Ac
  224 8.623975  128.16.3.52     128.16.13.118   TCP       visionpyramid > hydap [PSH, ACK] Seq=46319 Ac
  225 8.623989  128.16.13.118   128.16.3.52     TCP       hydap > visionpyramid [ACK] Seq=1 Ack=46321 W
  226 7.237522  128.16.13.118   128.16.3.52     TCP       dectalk > 15001 [ACK] Seq=23932 Ack=748230482
  227 7.237556  128.16.13.118   128.16.3.52     TCP       dectalk > 15001 [PSH, ACK] Seq=25192 Ack=7482
  228 8.638011  128.16.3.52     128.16.13.118   TCP       15001 > dectalk [ACK] Seq=748230482 Ack=25192
  229 8.638045  128.16.13.118   128.16.3.52     TCP       dectalk > 15001 [PSH, ACK] Seq=25765 Ack=7482

+ Frame 221 (1314 bytes on wire, 1314 bytes captured)
+ Ethernet II, Src: Intel_5a:7a:00 (00:16:76:5a:7a:00), Dst: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8)
- Internet Protocol, Src: 128.16.3.52 (128.16.3.52), Dst: 128.16.13.118 (128.16.13.118)
    Version: 4
    Header length: 20 bytes
  + Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 1300
    Identification: 0x941d (37917)
  + Flags: 0x04 (Don't Fragment)
    Fragment offset: 0
    Time to live: 128
    Protocol: TCP (0x06)
  + Header checksum: 0x50fc [correct]
    Source: 128.16.3.52 (128.16.3.52)
    Destination: 128.16.13.118 (128.16.13.118)
+ Transmission Control Protocol, Src Port: visionpyramid (1247), Dst Port: hydap (15000), Seq: 44467, Ack: 1, Len: 1260
+ Data (1260 bytes)

0000  00 13 d4 f7 dc f8 00 16  76 5a 7a 00 08 00 45 00   ........ vZz...E.
0010  05 14 94 1d 40 00 80 06  50 fc 80 10 03 34 80 10   ....@... P....4..
0020  0d 76 04 df 3a 98 1a 03  d8 97 ed 4e a5 aa 50 10   .v..:... ...N..P.
0030  fc 00 7e 7a 00 00 36 2e  39 32 38 34 35 37 37 40   ..~z..6. 92845770
0040  31 31 2e 34 35 30 36 39  37 40 2d 37 2e 32 37 37   11.45069 7@-7.277

Internet Protocol (ip), 20 bytes    Packets: 15527 Displayed: 15527 Marked: 0    Profile: Default
```

File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

FIGURE 3.29

Frames in the TCP boid demo with larger data size

3.4.3 Ping, traceroute and ICMP

Internet Control Message Protocol (ICMP) is used by the network layer to relay information about its status. If an IP packet doesn't arrive, then ICMP can be used to return the IP packet to its source with an error message. ICMP was one of the possible protocols that could be indicated within the IP header, see the previous section. However, ICMP is commonly sent by a router, and thus it is considered to be at network level as hosts need not be involved. ICMP is unreliable as it provides no reliability.

The ICMP packet format is shown in Figure 3.30. The important fields are the type field and the code field. Together these give information about the status of the network. Type 3 corresponds to Destination Unreachable, and there can be several reasons given by the code. Some examples are:

0: Destination network unreachable
1: Destination host unreachable
4: Fragmentation required, and DF flag set
7: Destination host unknown

Bits	0	15	16 31
160–191	Type	Code	Checksum
192–223	ID		Sequence

FIGURE 3.30

ICMP packet format within an IP packet that has no options set

Codes 0 and 1 might be returned when a route can't be found, and code 7 when a route leads to a network where the host isn't known. Code 4 is useful in a few contexts: it indicates that the Do Not Fragment field was set in an IP packet, but the packet can't be forwarded without fragmenting: the source host can try to adapt to this by sending smaller IP packets.

Some other important types and codes are:

0/0: Echo Reply
8/0: Echo Request
11/0: TTL expired in transit

The Echo Reply and Echo Request packets are used to determine if a host is reachable. A common tool that uses these is the "ping" tool which is available on most operating systems. The following command issued from a machine at UCL computer science:

```
ping wikipedia.org
```

generates the response:

```
PING wikipedia.org (208.80.152.2) 56(84) bytes of data.
64 bytes from rr.pmtpa.wikimedia.org (208.80.152.2): icmp_seq=0
   ttl=47 time=116 ms
64 bytes from rr.pmtpa.wikimedia.org (208.80.152.2): icmp_seq=1
   ttl=47 time=115 ms
64 bytes from rr.pmtpa.wikimedia.org (208.80.152.2): icmp_seq=2
   ttl=47 time=115 ms
64 bytes from rr.pmtpa.wikimedia.org (208.80.152.2): icmp_seq=3
   ttl=47 time=114 ms
64 bytes from rr.pmtpa.wikimedia.org (208.80.152.2): icmp_seq=4
   ttl=47 time=115 ms
64 bytes from rr.pmtpa.wikimedia.org (208.80.152.2): icmp_seq=5
   ttl=47 time=114 ms
--- wikipedia.org ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5042ms
rtt min/avg/max/mdev=114.615/115.197/116.824/0.863 ms, pipe 2
```

The ping tool sends an ICMP packet with Echo Reply, and calculates the RTT for an Echo Reply to be received. We pressed Control-C to stop the ping after six packets were sent. We can see that the RTT is ~115 ms on average.

An associated tool is “traceroute” (on Unix) or “tracert” (on Windows). The two tools don’t work exactly the same way, but the basic idea is to send IP packets with increasing TTL values. As we noted in the previous section, routers decrement the TTL on an IP packet when they forward the packet. Packets are sent with an initial TTL, and if this reaches zero, an ICMP packet is returned with the 11/0 type and code. By systematically increasing the TTL we can discover the routers on the path between the source and the destination as each router will be expected to return ICMP packets. In practice, traceroute or tracert may not find the complete route: some routers don’t return ICMP packets, partly in response to certain types of network attacks, such as Denial of Service (DoS). Thus from the same machine, the command

```
traceroute wikipedia.org
```

generates the following response (from which we have removed some lines for space reasons):

```
traceroute to wikipedia.org (208.80.152.2), 30 hops max, 38 byte
   packets
1 cisco (128.16.6.150) 3.544 ms 0.817 ms 0.649 ms
2 128.40.255.29 (128.40.255.29) 0.323 ms 0.245 ms 0.229 ms
3 128.40.20.1 (128.40.20.1) 0.381 ms 0.283 ms 0.254 ms
4 128.40.20.62 (128.40.20.62) 0.311 ms 0.285 ms 0.276 ms
5 ic-gsr.lmn.net.uk (194.83.102.81) 0.351 ms 0.257 ms 0.256 ms
6 so-1-0-0.lond-sbr1.ja.net (146.97.42.61) 0.529 ms 0.511 ms
   0.527 ms
7 so-6-0-0.lond-sbr4.ja.net (146.97.33.154) 0.796 ms 0.812 ms
   0.815 ms
8 if-15-0-0.mcore3.LDN-London.as6453.net (195.219.195.85) 0.922
   ms 0.848 ms 0.800 ms
9 if-5-0-0.mcore3.L78-London.as6453.net (195.219.195.10) 1.125
   ms 1.137 ms 1.118 ms
MPLS Label=513 CoS=0 TTL=1 S=1
10 if-12-0-0-983.core2.NTO-NewYork.as6453.net (216.6.97.37)
   88.627 ms 88.717 ms 88.999 ms
MPLS Label=673 CoS=0 TTL=1 S=1
…
18 w006.z207088246.xo.cnc.net (207.88.246.6) 115.472 ms 117.700
   ms 114.693 ms
19 * * *
20 * *
```

Traceroute sends three packets with each TTL, and prints on each line: the hop count, the name of the router or host and its IP address, and then the RTT. After 18 hops, our traceroute stops returning responses. However, we can note some interesting aspects: our local gateway (128.16.6.15) is the first hop, and we configured this in our Internet settings. We can also see how UCL (128.40.*.*) is connected to the London Metropolitan Network (*.lmn.net), to the U.K. academic backbone SuperJANET (*.ja.net), then through a peering service (as6453.net) at hop count 9 to New York at hop count 10. We can also note that the RTT goes up from about 1 ms to 88 ms! By the time we reach hop count 18, the RTT is ~116 ms on average.

Despite us not being able to traceroute from this host to wikipedia.org, the practical use of traceroute is usually to find misconfigured routes or problematic firewalls. There are many web pages that provide traceroute and ping services so that the reachability can be tested from a variety of sources on the Internet.

3.4.4 Routing on the Internet

We've briefly outlined the purpose of routing and forwarding. Routing in general is outside the scope of this introduction as it is a complex topic. The specifics of how routing is done will mostly concern those who are building larger networks; as we've seen for small networks such as a small office or home, a single gateway is often used. However, it is worth knowing a few of the terms as routing is a very important part of how the Internet is structured.

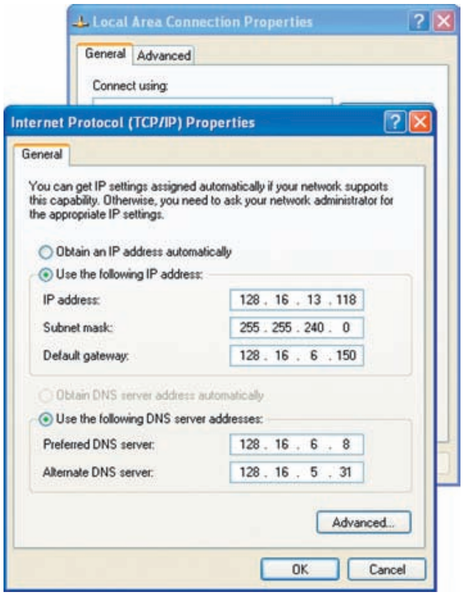
At a local level, we saw that with a single gateway, the only thing that is needed to be known is what is *upstream* from the gateway, that is the ISP. With a larger network, such as a larger company or university, there will necessarily be a number of routers. At this scale, a protocol such as Routing Information Protocol (RIP) or OSPF is appropriate. Both are types of Interior Gateway Protocol (IGP). IGPs calculate shortest routes across networks. RIP does this by having each router broadcast advertisements about local routes and its known distances to other routers. In OSPF, each route calculates a shortest path based on the knowledge of the whole network.

IGPs work well on a large network, but cannot scale to the full Internet. Routing over the larger Internet is done by the Border Gateway Protocol (BGP), which provides routing between networks of networks. BGP is deployed at the scale of large ISP or backbone networks in order to determine how networks are connected to each other.

3.4.5 DHCP

The final piece of network technology we need to discuss isn't really at the network layer. Dynamic Host Configuration Protocol (DHCP) is actually an application layer protocol. However, its role is to configure network layer parameters. Commonly, users do not configure their IP settings manually, but just indicate that DHCP is used. Figure 3.31 shows the Windows XP IP settings dialog, with the boxes to manually set the IP, gateway, etc. or just "Obtain an IP address automatically" and "Obtain DNS server address automatically" which invoke the use of DHCP.

Setting the DHCP flag uses a broadcast service on the network. Broadcast means that an IP packet is sent without a specific destination address, to anyone who is listening to broadcast messages. A DHCP server will then respond with an offer of IP settings that the host can use, which the host must then confirm. DHCP is extremely convenient, but obviously poses a security risk. This is commonly mitigated by


Local Area Connection Properties

General | Advanced

Connect using:

Internet Protocol (TCP/IP) Properties

General

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

○ Obtain an IP address automatically
[✓ Use the following IP address:]

IP address:	128 . 16 . 13 . 118
Subnet mask:	255 . 255 . 240 . 0
Default gateway:	128 . 16 . 6 . 150

○ Obtain DNS server address automatically
[✓ Use the following DNS server addresses:]

Preferred DNS server:	128 . 16 . 6 . 8
Alternate DNS server:	128 . 16 . 5 . 31

FIGURE 3.31

Windows XP dialog for configuring IP settings. (Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation)

having the DHCP server check that the MAC address (see Section 3.5.1) of the host is a well-known host before offering IP settings to it. IP settings obtained in such a way are usually on a lease, which means that the host must require new IP settings after a certain period. On a work or home network, the host might well get the same settings each time, but on a public service, the settings are likely to change, potentially interrupting the operation of certain network applications.

3.5 LINK AND PHYSICAL LAYER

Link-layer technologies are many and various, though a few technologies dominate deployment. The main constraint on the type of technology used is the bandwidth required. Today, if high bandwidth is required, the physical layer will likely be a fiber-optic cable, and over this will run a link layer protocol, which modulates light at different frequencies on to this fiber. Fiber Distributed Data Interface (FDDI) is such a protocol. For lower bandwidths, copper cable is suitable and twisted pairs of cables provide good resistance to interference. On top of this, there are many standards, common ones being the cables which are used for Ethernet, and the simple pairs of wires that form many telephone services. If the connection needs to be untethered, then the physical medium will be radio, and there are varieties of link-layer protocols here, such as 802.11 (a/b/g) which is common for laptops and PDAs, and various digital mobile phone protocols such as 3 G (HDSPA) or 2 G (GPRS). We discuss each of these technologies in a little more detail below.

The second constraint on the type of technology is the distance over which it can operate. Fiber can be run over long distances, but the signal needs boosting to cross continents or oceans. Copper wires attract interference over longer distances, so the quality of communication may vary with distance. Radio networks vary in range from tens of meters (802.11) to kilometers (2 G).

The main role of the link layer is to provide access to a transmission channel and effect communication. Link layers may or may not provide error correction, but since an IP is not reliable, there is no requirement that the link layer itself be reliable. Aside from bandwidth and range, transmission channels have two characteristics which are important: whether or not they are broadcast or point-to-point, and whether they are full-duplex or half-duplex. A point-to-point connection is relatively trivial to set up: both ends need to negotiate what speed, error correction and flow control will be used. Broadcast channels, such as wireless, are a single resource that multiple senders may wish to use. Because there can't be an explicit flow control between independent hosts, there is always the risk that two parties will send at once and their transmissions with clash. Full-duplex communication means that both or multiple hosts can communicate simultaneously. Half-duplex means that only one may.

At the time of writing, access to the Internet from home or a small business would probably be over dial-up modem, asynchronous digital subscriber line (ADSL) or hybrid fiber-coaxial cable (HFC). The first two run over the copper twisted pair provided by the phone service, the latter over the cables provided by cable TV companies. Each of these is a point-to-point technology. Bandwidth is defined by the quality of the connection and the adapters or *modems* at each end that modulate and demodulate the signal from the link layer. Dial-up might achieve 56 kbit/s, but ADSL might achieve 8 Mbit/s. Especially with ADSL, the quality of the wires determines the rate that can be achieved. Hence at the time of writing in the U.K., there was quite a bit of controversy over whether broadband advertised at 8 Mbit/s was really such: many customers would actually get much less as they were too far from the telephone exchange. Access for a larger organization might use an optical fiber, with speeds of 100 Mbit/s, 1 Gbit/s or more. Such lines would be leased directly from a provider and might need to be laid specifically.

Within a building or campus, the most common technology in use is Ethernet. At the time of writing, most common adapters run at 1 Gbit/s. 10 Gbit/s is available, but not common. Ethernet is cheap to deploy and the common cabling, unshielded twisted-pair category 5 (UTP CAT 5) can run up to 100 m. Each machine would typically be wired into a *network hub* or *switch*. A network hub is quite simple, and just copies frames on one link to all other links. They are relatively uncommon now because switches are very cheap. There are a lot of different types of switch, but the simplest type knows the MAC address (see below) of each link and then only forwards packets to the link(s) which they are destined. This would be referred to as a *link layer switch*, and is notably different from a router because it doesn't read any information in the IP packet header and can't forward frames from one link type to another.

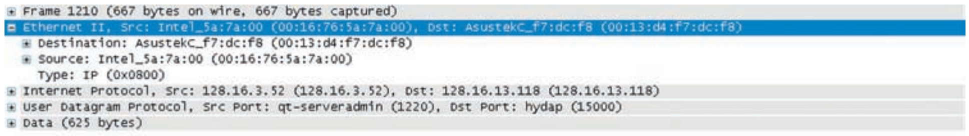
Also now very common are wireless networks based around the IEEE 802.11 standard, known as Wi-Fi. The most common implementations, 802.11 b and 802.11 g, both use radio frequencies at 2.4 GHz. 802.11 b provides 11 Mbit/s, 802.11 g provides 54 Mbit/s. Because it is a broadcast technology, this is shared bandwidth. A single wireless local-area network (WLAN) base station might cover an area of a few tens of meters. A key feature of the technology is that there are several channels, thus overlapping base stations need not contend for the same channel, but even if they do, transmission will be possible to both. Wi-Fi is now commonly available in public areas, and many cafes, libraries and hotels offer it as a free or paid service.

Mobile phones provide a variety of data services over which an IP can run. Again there are a variety of standards, but commonly deployed services include General Packet Radio Service (GPRS) (which uses GSM) and High Speed Downlink Packet Access (HSDPA) which uses Universal Mobile Telecommunications Service (UMTS). The former, also known as 2.5 G, commonly achieves up to 114 Kbit/s sending and receiving. The latter, also known as 3 G or 3.5 G, can provide up to 14.4 Mbit/s, but many handsets support only 3.6 Mbit/s. These are cellular services, so as the users move, they will be communicating with different cell towers. Thus, there needs to be a hand-over between towers. This does impact the network quality. Like Wi-Fi, bandwidth is shared, so the more users on a particular cell, the lower the overall bandwidth. Because users might be moving, bandwidth changes are quite common; even if the user is stationary other users might move into the same cell.

3.5.1 Ethernet

To complete our exploration of the boids application, let's look one final time at the Wireshark logs. The two machines narok and seychelles were both connected to an Ethernet-based LAN. Specifically they were connected through a small switch, a Netgear GS105. This is a common situation solution for a large building. The office containing the two machines has a UTP CAT 5 connection direct to a larger switch in a routing cupboard about 20 m away which serves the whole floor. This larger switch has a fiber connection to the department's single router. There is one port in the room and four machines. The switch has one link to the port out of the room, and four links to the four machines. Figure 3.32 shows the Ethernet frame information for the same frame in the UDP boids application demonstrations as was previously shown in Figures 3.6 and 3.25.

The format of one of an Ethernet Type II frame is given in Figure 3.33. The key features are the *Destination MAC Address* and *Source MAC Address*. These are six bytes each. Ethernet adapters have unique MAC addresses: the first three bytes give a manufacturer identifier, and the second three are assigned by that manufacturer and are usually unique for each device shipped. Thus, in Figure 3.32, Wireshark has been able to identify that 00:13:d4 is the manufacturer Asustek and 00:16:76 is Intel. The field *EtherType* identifies the protocol that this frame contains. 0x0800 means IP. As in an IP packet, the protocol at the next level up has to be identified, so that the receiver can handle it properly. The *CRC Checksum* isn't shown by Wireshark,



```
⊞ Frame 1210 (667 bytes on wire, 667 bytes captured)
⊟ Ethernet II, Src: Intel_5a:7a:00 (00:16:76:5a:7a:00), Dst: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8)
  ⊞ Destination: AsustekC_f7:dc:f8 (00:13:d4:f7:dc:f8)
  ⊞ Source: Intel_5a:7a:00 (00:16:76:5a:7a:00)
    Type: IP (0x0800)
⊞ Internet Protocol, Src: 128.16.3.52 (128.16.3.52), Dst: 128.16.13.118 (128.16.13.118)
⊞ User Datagram Protocol, Src Port: qt-serveradmin (1220), Dst Port: hydap (15000)
⊞ Data (625 bytes)
```

FIGURE 3.32

Ethernet information for frame 1210 from the UDP boids demo

Bits	0 15	16 31
0–31	Destination MAC Address …	
32–63	… Destination MAC Address	Source MAC Address …
64–95	… Source MAC Address	
96–127	EtherType	Data
…	Data	
…	CRC Checksum	

FIGURE 3.33

Ethernet Type II frame

but it is probably removed at a low level by the Ethernet device itself. At the physical level, there are other bits that are sent as preamble, but we needn't be concerned with those.

Despite all the protocols we've discussed before, there is still a small piece missing. After all the encapsulation, how does a machine on an Ethernet know what the destination MAC address is? This is the role of the Address Resolution Protocol (ARP). We fleetingly saw an example in Figure 3.3. ARP is a link-layer protocol which allows hosts to identify which MAC address is mapped to a particular IP address.

3.5.2 Comparisons

To wrap up the section on link and physical layers, in Table 3.3 we summarize some key features of each technology: bandwidth, latency and range. There are many variations on these technologies, so this should be used as a rough guide.

3.6 FURTHER NETWORK FACILITIES

So far we've discussed the Internet as a point-to-point service supporting a best effort service. Of course the Internet has been extremely successful, but many applications don't map easily to these conditions. There have been many efforts to extend the

Table 3.3 Comparison of Link Technologies

Link Type	Bandwidth	Latency	Range
ADSL	Medium	Low	km
HFC	High	Low	km
Ethernet	High	Low	100 ms
Fiber	Very High	Very Low	100s–1000s km
2 G	Low	High	km per cell, but cells pervasive
3 G	Medium	Medium	km per cell, but cells fairly pervasive
802.11 b	Medium–High	Low	100 ms, sparse
802.11 g	High	Low	100 ms, sparse

services that the network provides. Although these very quickly become advanced topics and areas of networking research, we'll outline two main classes of facility: multicast and quality of service. These facilities bridge the layers of the TCP/IP stack, but they aren't provided natively in the IP layer. Thus, as you might expect, there will be difficulties using them because they aren't universally available.

3.6.1 Multicast

Multicast is the process of sending one IP packet and having it delivered to multiple receivers. Unlike broadcast where one IP packet is delivered to everyone on a LAN, multicast can work across wide areas, and the packet is only delivered to hosts that are interested in receiving the packet. The multicast model (Diot et al., 1997) is thus ideally suited for group communication, where all hosts that share common interest are grouped together by using a logical address. With multicast, the bandwidth requirements grow linearly with the number of participants (Fisher, 2002).

The data link layer might support multicast by default, such as in the case of the Ethernet, where the mapping is direct. However, the same cannot be said of the networking layer, as the initial IP protocol did not foresee the necessity for multicast until it was proposed as an extension by Deering and Cheriton (1990). To distinguish from normal unicast addressing, the first 4 bits (1110) of the 32-bit IP address identify a multicast address (class D) that ranges from 244.0.0.0 to 239.255.255.255. This address is entirely logical, giving the possibility of cross-data transmission due to a clash in the address space by different applications adopting identical multicast addresses. The traditional approach with networked multimedia is to resort to the session directory tool known as sdr (Handley, 1997) which announces existing multicast sessions.

The process of multicast routing is fundamentally different from the point-to-point routing of unicast communication, since the distribution model resembles a tree and the routing process consists of two distinct processes: the building of a distribution tree and the actual forwarding. A general overview of the building process and forwarding process is given in Figure 3.34.

The building of a distribution tree starts when a host requests to join a multicast group. They do this using IGMP (RFC 3376) to inform the local router(s) of their intent. The membership is dynamic, thus a receiver may join and leave anytime and any number of times. The local router then needs to build a distribution tree for this group if one does not exist. There are many ways to do this, each with its advantages and disadvantages. The protocols can be basically categorized into the following:

- Source-Based Tree: These protocols, such as Distance Vector Multicast Routing Protocol (DVMRP) (RFC 1075) and Protocol Independent Multicast Dense-Mode (PIM-DM) (RFC 3973), build the shortest path distribution tree from the source. They do this by flooding the network, and then pruning unnecessary links.
- Centered-Based Tree (CBT): These protocols, such as PIM Sparse Mode (PIM-SM) (RFC 2362) and CBT (Ballardie et al., 1993) build distribution trees with the shortest path from a well-known central root of the tree.

Independently of the routing protocol used, there is always a delay from the moment of joining a multicast group and beginning to receive data because this routing tree needs to be built. This delay may be sufficient to cause inconsistencies within the VE, thus particular care should be taken with management of group membership and its mechanisms (Levine et al., 2000).

In the next chapter, we show how to implement a peer-to-peer NVE using multicast. This is actually probably one of the simpler applications, as all the peers are actually identical code and the actual network protocol is simpler. Unfortunately, multicast is not widely deployed in the Internet. In particular, it isn't deployed for the backbone routers that support WANs. Partly this is because the Internet wasn't designed for multicast and thus changing any installed equipment would be risky. We come back and discuss multicast and its variants in Chapter 12.

3.6.2 Network QoS

The Internet is a best-effort service. As we will see later in the book, particularly in Part III, best effort can mean many things, but one thing that is almost certain is that the QoS (bandwidth, latency, etc.) will change over time. For most applications this doesn't matter, but for some, such as high-quality audio and video, and some applications of NVEs, it would be better to have some guarantees on services. Thus many proposals have been made to support *QoS* on the Internet (Xiao & Ni, 1999; Zhao et al., 2000).

QoS can mean various things including guaranteeing that a certain latency be kept or that a certain bandwidth be available. The two main ways of achieving such guarantees would be to reserve capacity on routers, or pick routes across the network which bore in mind the likely capacity.

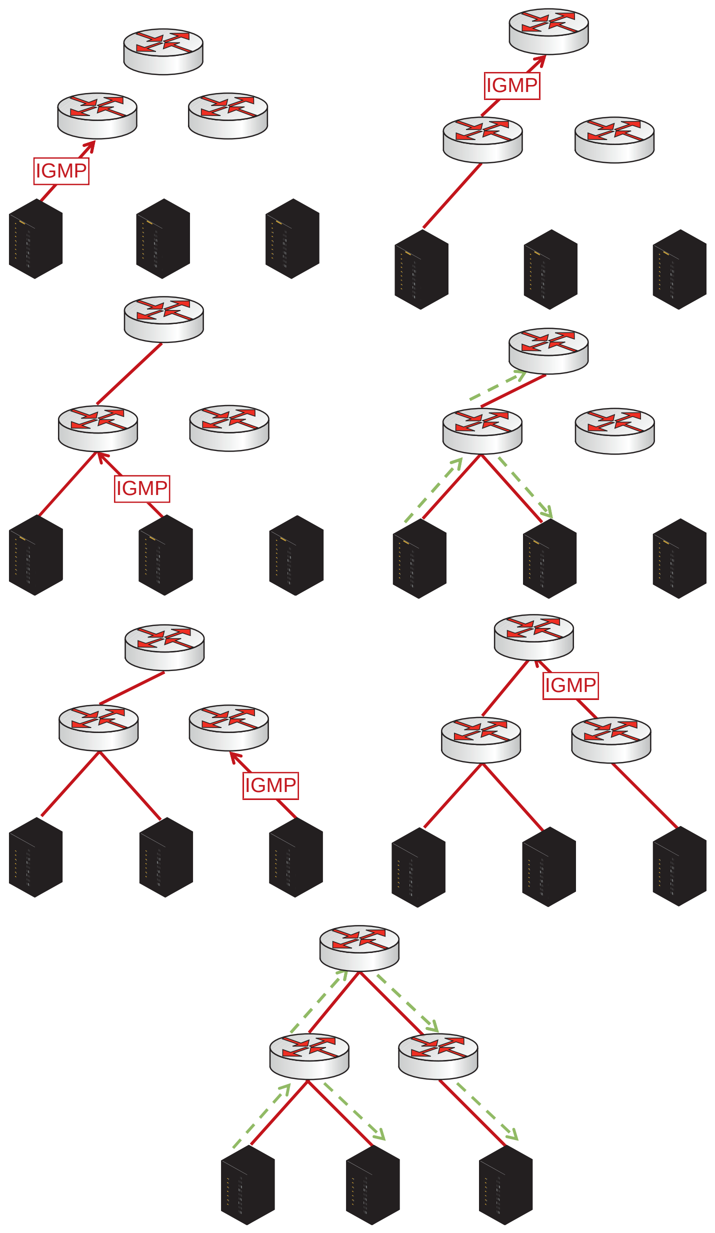


FIGURE 3.34

A multicast group is formed by sending IGMP messages. Top Row: The first host joins the group by sending an IGMP message. This message is relayed by the server a certain number of hops “upwards” on the network. Second Row: Another client joins the group by sending an IGMP message; now the router on the bottom left knows about the shared group. Thus when the first client sends a message, it is relayed to the peer. Third Row: Another client joins the group with an IGMP message, but the first router encountered doesn’t know about it. The higher-level router can however form a route. Final Row: A message sent by the first client now reaches all the other clients

The Resource Reservation Protocol (RSVP) (RFC 2209, RFC 2205) is a protocol aimed at supporting QoS on the Internet by making resource reservation along a particular path. The RSVP is common to both the hosts and the routers of the network. The hosts that are receivers initiate the protocol by sending a RESV message toward the source, requesting the allocation of the appropriate resources along the path. In turn, the source emits PATH messages downstream to maintain the state at the routers and to inform the hosts of the current state of the data flow, such as its traffic characteristics. The fact that RSVP is receiver-initiated allows the protocol to scale to large groups and support the receiver's heterogeneity.

A particular characteristic of the protocol is the fact that the request for resources corresponds to simplex data flows, from the receiver to the source. When an application needs to have reservation for a full-duplex data flow, it is required for both remote parties to send their corresponding RESV, making separate requests for resource reservation.

RSVP has been used with NVEs: in Chassot et al., (1999) an attempt is made to associate a RSVP architecture with applications based on DIS (see Section 7.3). However, RSVP by its natures is concerned with QoS between two points, and one of the fundamental characteristics of NVEs and NGs is that data will need to be routed between many points, and that any one host's interest in receiving data from another might change quite quickly (see Section 12.2).

The goal of Integrated Services (IntServ) (RFC 1633) is to provide end-to-end QoS by enhancing the best-effort service model with another model that provides QoS guarantees. This is again achieved by reserving in advance the necessary resources in all the routers along a particular path between a source and a receiver. IntServ thus uses a reservation protocol (commonly RSVP), but also provides a framework for checking whether resources are available, shaping traffic to fit the QoS that has actually been reserved and classifying traffic in to different queues on routers.

There are three main barriers to wide deployment of IntServ. The complexity of the necessary mechanism makes the framework incompatible with the existing Internet and incremental deployment is not feasible. The scalability of the framework is severely compromised by the amount of additional state required to maintain on a per flow basis. Finally, the amount of overhead traffic signaling is prohibitive.

The Differentiated Services (DiffServ) (RFC 2474) model was devised to address the concerns of scale of IntServ. The approach consists of categorizing the data traffic into several classes, each with distinctive QoS. Traffic classification is achieved by marking each packet with the corresponding QoS class and this process effectively aggregates flows together into Behavior Aggregate (BA). In addition, routers have Per-Hop Behavior (PHB) profiles for each type of QoS class. These profiles indicate to the router how to handle the packets belonging to each class.

The DiffServ architecture makes a distinction between the core routers and the edge routers of the network, pushing complexity toward the edge. The core routers continue to be very simple, focusing on fast forwarding mechanisms, but now taking into account the BA. The routers toward the edge must perform traffic conditioning, assuring that the traffic forwarded to the core fits the existing BA. Since the packet classification is based on the OS field of the IP packet, it is possible to have incremental deployment of DiffServ (we mentioned these fields in Section 3.4.2).

Should the VEs be of reduced dimensions with few participants, analogous to the scenario of a videoconference, it is possible to benefit from the allocation of differentiated services (Yu et al., 2001).

DiffServ is more scalable than IntServ, but still requires additional complexity in the network, in particular at the edge routers. There is a need to establish Service Level Agreement (SLA) between customers and service providers, and in the case of dynamic SLA, a signaling protocol is required. There are no QoS guarantees, but it is possible to have relative QoS between different aggregated flows.

In summary, although there have been many research proposals for incremental deployment of network QoS, there is no one technology that is widely deployed. As with multicast, this means that QoS may be available for a specific NVE or NG implementation, but otherwise, if the system needs QoS, it becomes an application-level problem.

3.7 SUMMARY

We've given a brief summary of the different layers of the IP suite. There are shelves full of books that cover this in more depth, from high-level application protocols through to router and switch configuration. We have focused on pulling out a few characteristics that will shape the way we implement NVEs.

- Application layer protocols need to balance efficiency and compactness against readability. Although it is tempting to design messages to be as small as possible, ASCII strings are commonly used for header-like information as they are human readable.
- At each layer of the TCP/IP stack, there is header information to add and to provide that layer with the necessary information to handle the data it contains. This encapsulation in layers is an important property of the stack, and means that very different host types and software types can interoperate.
- UDP is an unreliable connectionless service, whereas TCP is a reliable connection-oriented service. The choice between the two is not as simple as just deciding whether or not reliability is needed. Reliability can be added on top of UDP: after all, TCP uses IP just as UDP. However, we pointed out that retransmission is not desirable in situations where the sender is sending rapidly changing data such as positions.
- TCP provides flow control and congestion control so that bandwidth of the application can be managed. It also avoids fragmentation of packets, so it can be more efficient at transmitting large messages, utilizing the bandwidth available fully. Implementing this over UDP would be onerous.

- IP provides the packet-routing layer of the Internet. Routers provide the basic forwarding mechanism, but there is also a "back-channel", ICMP, that can be useful to get information about the reachability of hosts.
- There are various link technologies in use, from mobile through wireless to fiber. There is a balance to be struck between cost and availability, though latency will be a particularly important feature for NVEs.
- Multicast and QoS support might be available under certain situations.

We've also introduced a small suite of tools that can help us explore how the network is working. Wireshark, Ping, traceroute/tracert, nslookup and ipconfig/ifconfig are all essential tools to help understand and debug network applications.

REFERENCES

All RFCs can be accessed by number at <http://tools.ietf.org/html/>, accessed February 1, 2009.

Ballardie, A., Francis, P., & Crowcroft, J. (1993). Core based trees (CBT): An architecture for scalable inter-domain multicast routing. *Proceedings SIGCOMM'93* (pp. 85–95).

CERT. (2008). CERT Vulnerability Note VU#800113: Multiple DNS implementations vulnerable to cache poisoning. United States Computer Emergency Readiness Team. <http://www.kb.cert.org/vuls/id/800113>, accessed January 29, 2009.

Chassot, C., Lozes, A., Garcia, F., et al. (1999). Specification and realization of the QoS required by a distributed simulation application in a new generation Internet. *Proceedings Interactive Distributed Multimedia Systems and Telecommunication Services, Lecture Notes in Computer Science*, 1718, 75–91).

Deering, S., & Cheriton, D. (1990). Multicasting routing in datagram inter-networks and extended LANs. *ACM Transactions on Computer Systems*, *8*(2), 85–110.

Diot, C., Dabbous, W., & Crowcroft, J. (1997). Multipoint communication: A survey of protocols, functions and mechanisms. *IEEE Journal on Selected Area in Communication*, *15*(3), 277–290.

Fisher, H. (2002). Multicast issues for collaborative virtual environments. *IEEE Computer Graphics and Applications*, *22*(5), 68–75.

Handley, M. (1997). On Scalable Internet Multimedia Conferencing Systems, PhD Thesis, University College London.

IANA. (2009). Port Numbers, <http://www.iana.org/assignments/port-numbers>, accessed January 29, 2009.

Kurose, J. F., & Ross, K. W. (2008). *Computer networking: A top-down approach* (4th ed.). Addison Wesley.

Levine, B., Crowcroft, J., Diot, C., et al. (2000). Consideration of receiver interest in delivery of IP multicast. *Proceedings Infocom 2000,* IEEE 78–88.

Salon. (2000). tv, <http://archive.salon.com/tech/view/2000/07/24/dot_tv/index.html>, accessed January 29, 2009.

Stevens, R. (1994). *TCP/IP illustrated, Volume 1: The protocols*. Addison-Wesley, Indianapolis, IN.

Stevens, R. (1995). *TCP/IP illustrated, Volume 2: The implementation*. Addison-Wesley, Indianapolis, IN.

Stevens, R. (1996). *TCP/IP illustrated, Volume 3: TCP for transactions, HTTP, NNTP, and the UNIX domain protocols*. Addison-Wesley, Indianapolis, IN.

Wikipedia Contributors. (2009). Internet Protocol Suite. <http://en.wikipedia.org/wiki/TCP/IP>, accessed January 29, 2009.

Xiao, X., & Ni, L. (1999). Internet QoS: A big picture. *IEEE Network*, *13*(2), 8–18.

Yu, H., Zhou, Q., Makrakis, D., et al. (2001). Quality of service support of distributed interactive virtual environment applications in IP networks. *Proceedings Pacific Rim Conference on Communications, Computer and Signal Processing,* IEEE 196–199.

Zhao, W., Olshefski, D., & Schulzrinne, H. (2000). Internet Quality of Service: An Overview. Technical Report CUCS-003-00, Columbia University, New York, February 2000.

Zimmermann, H. (1980). OSI reference model—The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, *28*(4), 425–432.