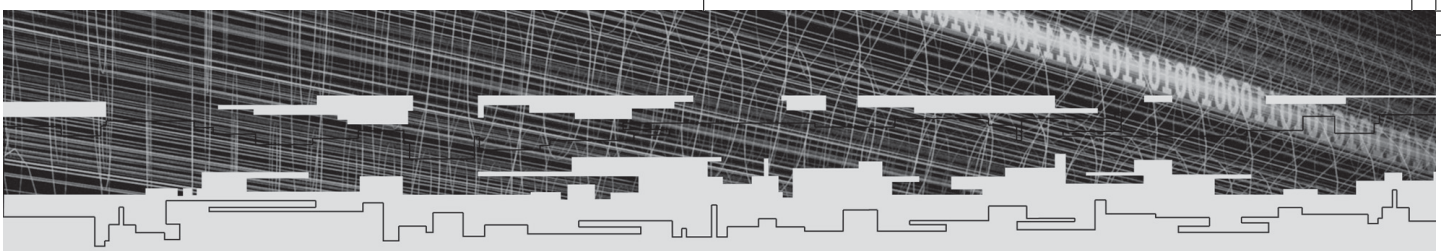
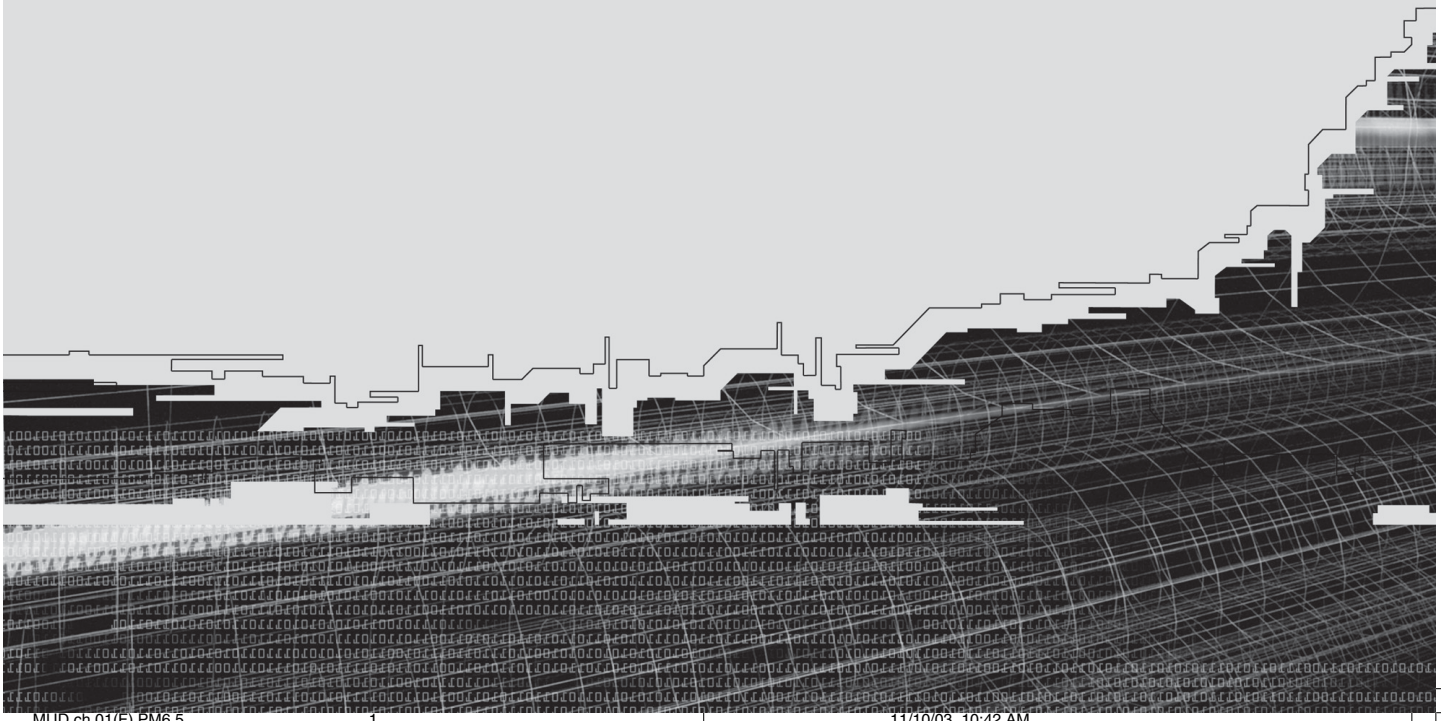
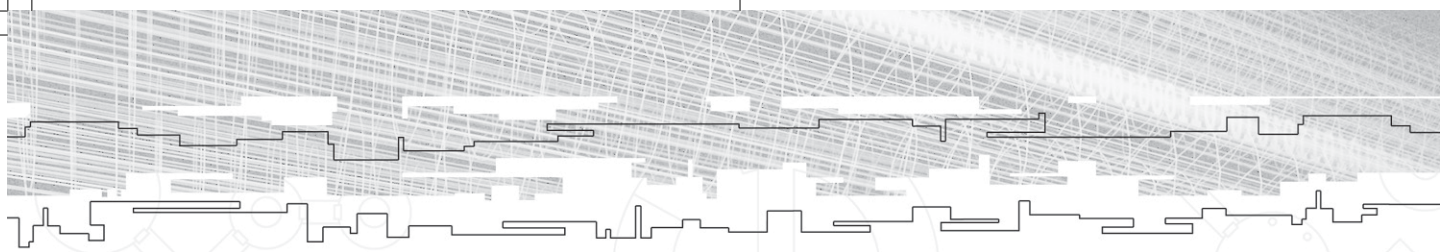
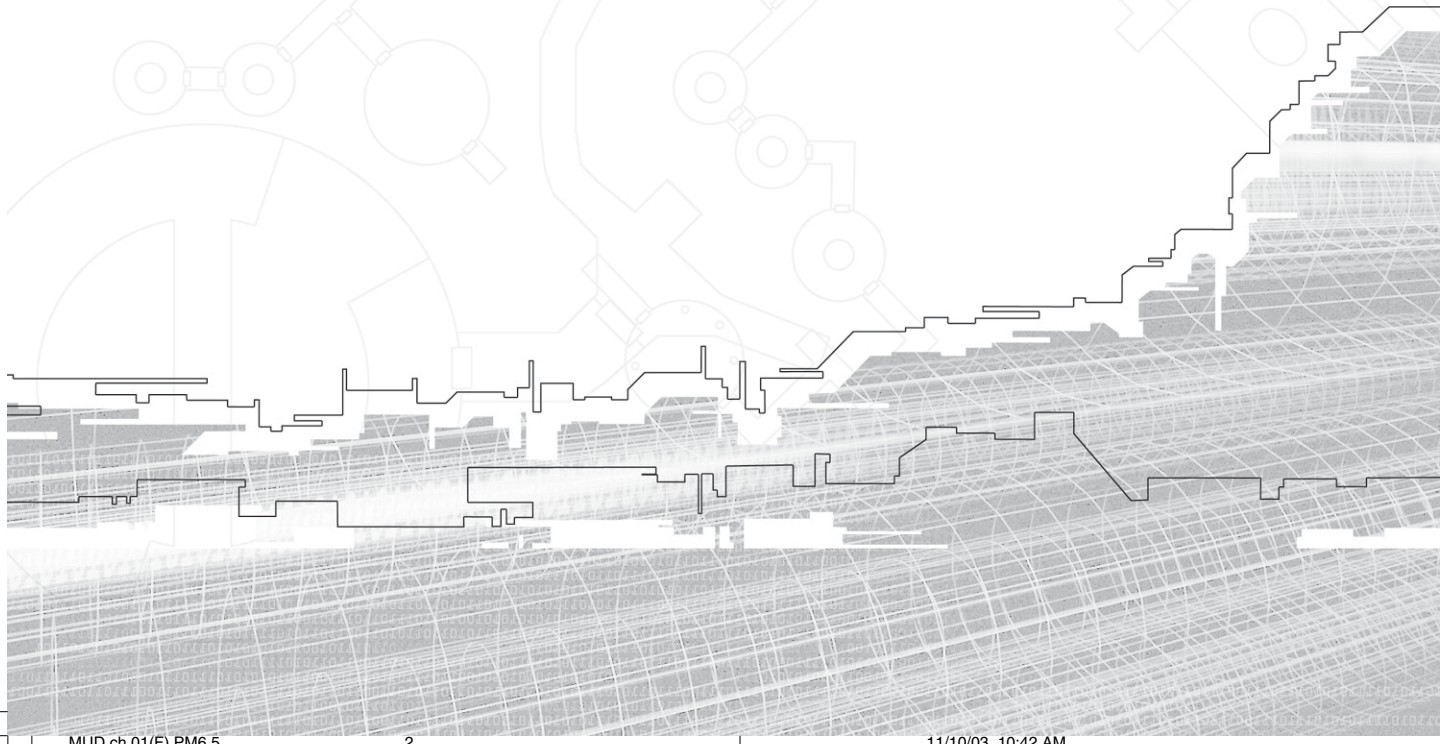


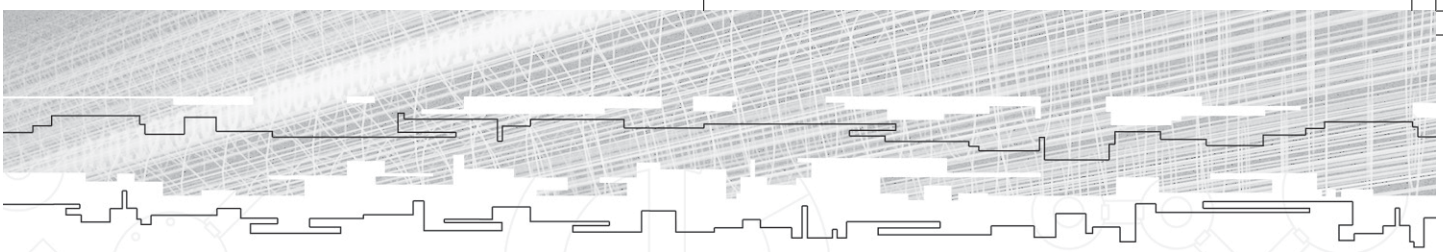
PART ONE

The Basics



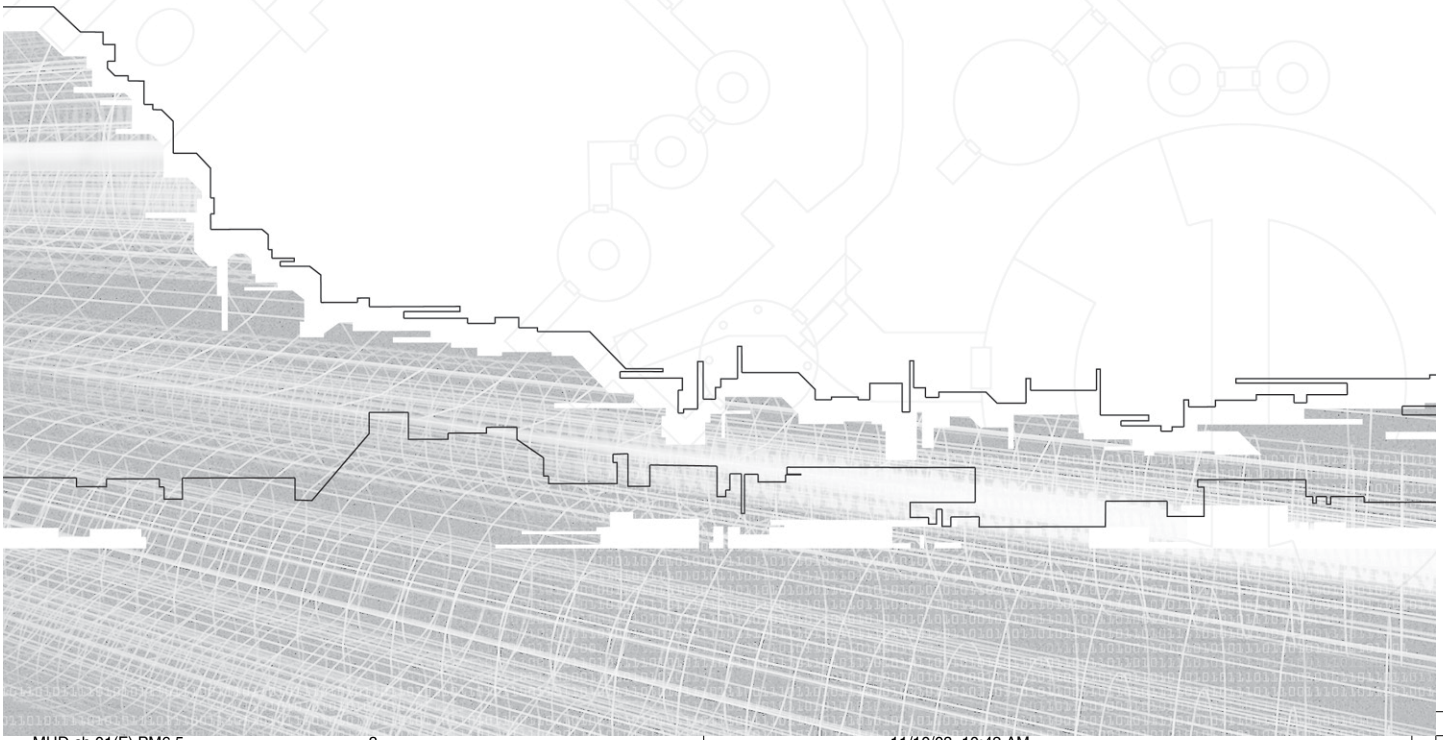






CHAPTER 1

Introduction to Network Programming



Unless you've been living under a rock for the past 20 years, you've probably heard about something called the *Internet*. To most people, that word is associated with ominous things like e-mail, the *World Wide Web* (*WWW*), and naughty pictures. To you, the game programmer, the Internet is so much more—a universe of its own where you can create games to play with people who live across town as well as those who live thousands of miles away.

The Internet is a grand thing for game programmers. It adds community interaction to games and allows players to match wits and reflexes with anyone, instead of being required to play against typically dumb and repetitive artificial intelligences. To learn how to efficiently program MUDs, however, you must first have a solid understanding of network programming. This chapter supplies that foundation. If you already have a good grasp of network programming, you may safely skip this chapter.

In this chapter, you will learn to:

- Relate the history of communication networks to game programming
- Understand the philosophy and layered hierarchy of Internet Protocols (IPs)
- Understand the basics of common transport protocols
- Find additional information on networking protocols

Why Learn the Basics?

I have found that it is always a good idea to know the mechanics of anything I intend to work on. I disagree with computer professors and gurus who rant for hours about the beauty of abstracting the interface of a mechanism from its mechanics (how it works, in essence) to justify the concept that you shouldn't need to know how something works to use it.

Indeed, few people who drive actually know the physics of acceleration and energy usage or even how an internal combustion engine works. At first, this can seem like a good thing; anyone can jump into a car without knowing how the engine works. You press the gas, and the car goes; you press the brake, and the car stops.

It's not always that simple however. I can't count the number of times I've been at a stoplight and watched the car next to me

NOTE

Throughput **is a communications term that describes how much data can go through the network per unit of time. For example, the throughput of a 56 kilobits modem is roughly around 56 kbps (kilobits per second), and the upstream throughput of my cable modem is around 128 kbps.**

accelerate as fast as possible only to stop in a few hundred feet at the next stoplight. Whenever that happens, I know the person has no idea of how energy and acceleration work.

The person who accelerates to 50 MPH and then immediately brakes to a halt wastes far more energy than the person who accelerates to 30 MPH, coasts, and then brakes to a halt. The first car wasted energy accelerating 20 MPH faster, only to have that energy drained away as heat energy in the brakes.

So you can see that knowing how something works may not be necessary for operating a mechanism but is useful for operating it *efficiently*. And as you may know, game programming is all about using things efficiently and taking them to the limit.

History of Communication Networks in a Nutshell

From the beginning of history, communication has been an important part of human society. As important as communication has been, the mass distribution of communication through networks is only a recent development. Most early communication was accomplished through horseback riders carrying written messages.

The invention of railroads brought a major advancement in communications networks by facilitating the transfer of massive amounts of mail across the world. But communication was still inadequate. While the *throughput* of these railroad networks was large, the *latency* was also large.

NOTE

***Latency* is a communications term that describes how long it takes for one piece of data to reach its destination. For example, it takes less than one millisecond (msec) for data to go from one computer to another on my home network, and around 15 msec for data to reach my Internet Service Provider's (ISP's) routers.**

While masses of mail could be sent through railroad networks, it still took weeks for some pieces to reach their destinations, and this was unacceptable to many people.

Electric Communication—Telegraphs to Telephones

In 1835, something amazing happened: The telegraph was invented.

The telegraph was essentially a long wire with a speaker on one end and a battery at the other. Figure 1.1 shows a simple telegraph "network." Whenever the battery was engaged, it sent an electrical signal down the wire that would power the speaker and cause a small tone to be heard. Since there were only two states of the communication—the presence or absence of sound—a special messaging system called *Morse code* was invented, which varied

the number and length of tones to represent different characters. Short tones were called dots, and long tones were dashes.

Even though communication in this manner had a low latency (tones were transmitted almost instantly), you can imagine that the throughput of this method of communication was very low. There were no machines back then to convert signals from Morse code to English, so people had to do it by hand.

The next major innovation in communications occurred in 1876 with the invention of the telephone.

NOTE

It is a commonly held "fact" that Samuel Morse invented the telegraph, but there are conflicting reports about a person named C.M. Renfrew inventing it as well. You can read about this more on the Internet if you wish; there's good information about telegraphs at this site: http://www.worldwideschool.org/library/books/tech/engineering/HeroesoftheTelegraph/chap1.html.

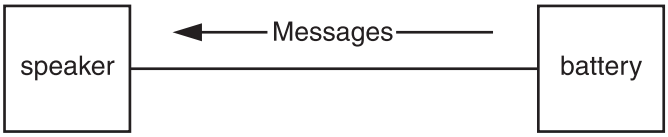


Figure 1.1

This simple telegraph network transmits electrical signals from one end to the other.

The telephone allowed people to encode sound data into an analog electrical pulse, which would then be sent down a wire, to the speaker on the other end (Figure 1.2). This method of communication was an incredible innovation, since with the direct interpretation of voice, communication could be accomplished without people encoding and decoding Morse code. This greatly improved the throughput of the communications, because voice data could now be transmitted in real time.

NOTE

Alexander Graham Bell is generally credited with inventing the telephone, but he was only lucky enough to get his patent approved first. Elisha Gray, working independently of Bell, simultaneously invented the telephone, but he didn't file his patent application fast enough. What have we learned today, class? Always file your patents immediately.

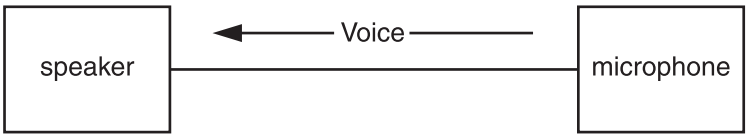


Figure 1.2

In this simple telephone network, voice data is turned into electricity by the microphone, and turned into sound on the other end by a speaker.

This method of communication only increased people's desire for faster and better communications, since only one person could use one telephone line at a time.

> **NOTE**
>
> **Both the telegraph and the telephone supported two-way communications, which Figures 1.1 and 1.2 do not show, for simplicity's sake.**

Switched Communication

Peer-to-peer networking connected many telephones to many other telephones. Alexander Graham Bell was a major proponent of this kind of network, and it worked well for small networks. Basically, every node in a peer-to-peer network is physically connected to every other node in the network through wires. This gets to be a major problem as the number of nodes grows, because, as you can probably see, the number of wires needed in this kind of network follows a geometric progression based on the number of nodes in the network. To add a third node to a network, you need two extra wires, making a total of three in the network. Table 1.1 shows a listing of the number of wires needed for networks with different numbers of nodes.

Table 1.1 Wires Needed for Peer-to-Peer Networks

Nodes	Wires
2	1
3	3
4	6
5	10
10	45
15	105
500	1,225

The number of wires needed in a peer-to-peer network follows this formula: (n * (n-1))/2. So you can see that any network that gets past a certain size is in the realm of being completely unmanageable.

Because of this, the concept of a centralized communications network was invented, and its implementation was called a *circuit-switched network*. This kind of network contains any number of *nodes* and one central *switching station*, arranged as shown in Figure 1.3.

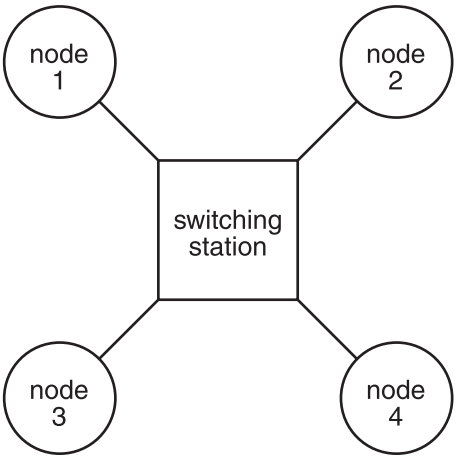


Figure 1.3

This simple switched communications network connects four nodes to a switching station.

Since only one conversation could be conducted at any given time on a telephone wire, the original networks had to use switching to enable multiple conversations to occur at the same time. Essentially, this is how it worked.

There was a human *operator* at the switching station, who monitored all the nodes for incoming activity. Whenever one of the nodes wanted to talk to any of the other nodes, a person called the operator from his node, and the operator asked whom he wanted to talk to. When the operator determined whom the caller wanted to talk to, he physically connected a wire from the caller's circuit to the destination circuit. For example, Figure 1.4 shows node 1 connected to node 4. When node 1 wants to talk to node 4, an operator physically connects the circuits with a wire.

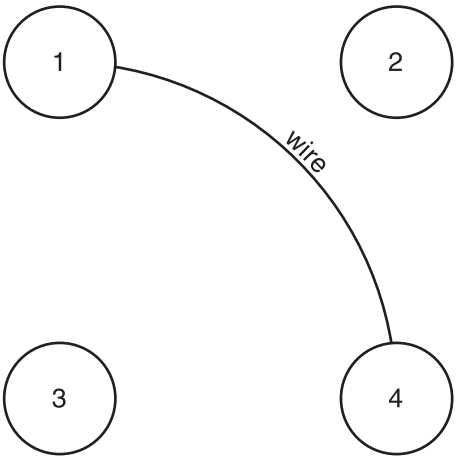


Figure 1.4

Four circuits at the circuit switching station from Figure 1.3.

So, with this network, a total of two conversations can be held at the same time, and any single node can talk to any other node, as long as the line is open. This spawned a major breakthrough in communications, but its service was still inadequate. Eventually these switching stations became too large for human operators to manage, so methods were developed to spread out the communications into many switches, as shown in Figure 1.5.

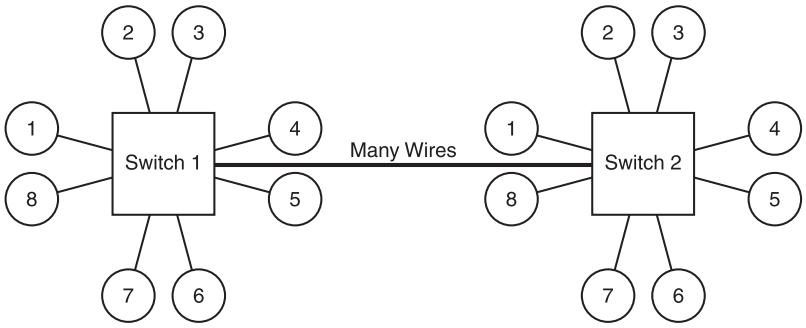


Figure 1.5

In this configuration, two switches are connected with many wires, allowing nodes from each local switch to connect to nodes on another switch.

When a person wanted to call someone at his local switch, the same procedure was followed. When a person wanted to call someone on another switch, the operator connected the person to the operator on the desired switch, and that operator connected the person to the right destination. Each switch had only a certain number of wires connecting it to other switches, and that limited the number of connections that could be made from switch to switch. For example, a switch may have 16 nodes, which allows up to eight *intra-switch* connections at once, but it may have only four wires connecting to an adjacent switch, which means that only four *inter-switch* connections can be made.

NOTE

The terms *intra* and *inter* refer to "internal" and "external" respectively. So *inter-switch* refers to connections between two nodes on one switch, but *intra-switch* refers to connections of nodes that are on different switches.

Eventually, each switch in the United States was numbered with its own area code, and this led to our current area code system.

It didn't take long for these networks to become such huge messes of wires that it was difficult to make connections. Therefore, an even more centralized system was created. The switches were given centralized switches, sometimes called *hubs*. Figure 1.6 shows one of these networks.

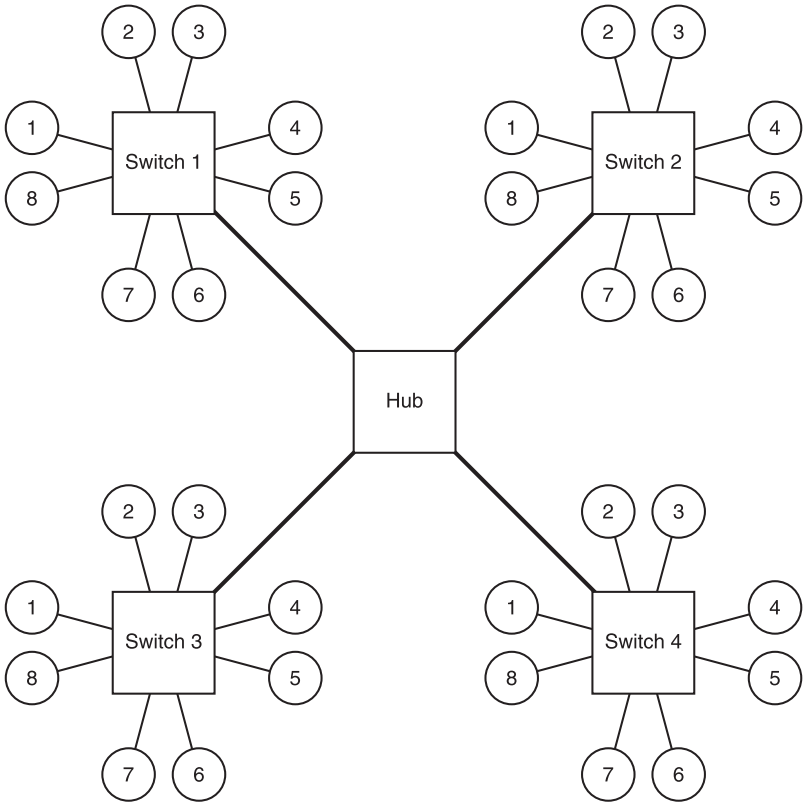


Figure 1.6

In this large switched network, the central switch controls connections among the intermediate switches.

Packet-Switched Networks

Traditional circuit switching was great, but it had too many limitations for our growing communications needs. Since traditional circuit-switched networks were so centralized, the main hubs could go down, and half of the communications in the country would instantly be halted. Only one line could be in use at any given time, limiting the number of concurrent connections drastically. There also came a time when it took about seven to eight minutes just to go through all the operators to connect to someone else on the network.

In the 1960s, the United States *Advanced Research Projects Agency* (*ARPA*) invented the first *packet-switched* network. The idea of such a network is to separate data into tiny chunks, called *packets*.

In this type of network, instead of only one connection per wire, special machines at the end of each wire accept discrete chunks of data (packets) and send each chunk one at a time down the wire, with the chucks arriving first sent first. These machines are called *switches*, but they are much more commonly known as *routers*. Figure 1.7 shows a simple network with two routers.

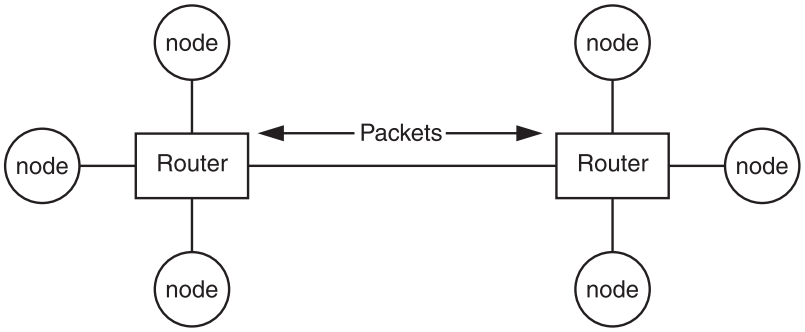


Figure 1.7

In a simple packet-switched network, the routers send packets of data down the single wire that connects them.

Whenever a node has data to send, it puts that data into a discrete-sized packet and then sends it to the router. The router decides where it goes and sends it to the right place. If the wire between the routers is busy, the router puts the packet in a queue and keeps it there until the wire opens up and is available for transmissions.

This kind of network is a great improvement, because it drastically reduces the number of wires needed to connect two switches. One of the downsides, however, is that since many more communications are now occurring on the same line at the same time, each connection has less bandwidth. The original *Defense Advanced Research Projects Agency Network* (*DARPANet)* didn't have enough bandwidth to transmit a single voice communication, unless it was the *only* communication going on at the time.

Since data packets had to be in a form that the routers could understand, and the routers were *digital* computers, it made sense that they would send digital data. Unfortunately, data sent over a wire is *analog* by nature, so the digital data needed to be turned into an analog signal using a device called a *modulator-demodulator* (*modem* for short). Early modems didn't do a great job of converting data efficiently, and were limited to a bandwidth of about 300 *baud*.

NOTE

Bandwidth **is a networking term that generally describes how much data can be sent through a network. In the traditional sense, bandwidth refers to the size of a signal. For example, telephones have 3,000 Hz of bandwidth, from 400 Hz to 3.4 KHz. Telephone wires are not rated to send data above or below those thresholds. AM Radio broadcasts use about 10 KHz of bandwidth each, FM Radio broadcasts use about 200 KHz, and VHF/UHF TV broadcasts use 6 MHz of bandwidth. Most people, when dealing with packet-switched networks, refer to their throughput as bandwidth as well.**

NOTE

Baud **is an old term, dating back to the days of telegraphs. The term comes from the name of one of the engineers who first worked with telegraphs, Jean Maurice Emile Baudot. The speed at which an electronic circuit changed states was measured in *bauds*, and a baud was roughly equivalent to the number of bits per second that could be transmitted. So 300 baud is about 300 bits per second. Modems stopped using the term baud at around the time the 14,400 bps modem was invented. Does this sound like ancient history?**

Over the years, significant improvements have been made to methods of data transmission over traditional copper wires. New inventions such as fiber-optic wires and even wireless *radio-frequency* (*RF*) communication allow data to be transferred in a much more efficient manner. Eventually, everything will be transferred over packet-switched networks, since they are far more cost-efficient and useful than the circuit-switched or broadcast networks that your telephone and cable companies use. You won't need a specific cable line, phone line, or Internet line; everything will connect into one standard interface.

NOTE

Eventually, all land-based copper wires will be replaced with *fiber optics*. Fiber-optic communication is an incredible breakthrough in the realm of wired communications. Every electrical circuit has resistance, which slowly saps out the signal strength and causes the wires to become hot. Therefore, to maintain signal strength on copper wires, repeaters must be placed on the wire to boost the signal and send it further. Not only do these boosters require lots of energy, but they slow down transmission speed as well. Fiber-optic wires directly transmit light impulses with much less signal drop-off, and since they transmit light directly, they are faster than traditional electrical signals as well. In addition, fiber-optic wires require fewer repeaters.

Communications

Broadband communications originally referred to cables that carried more than one type of data at the same time. The first types of broadband included Digital Subscriber Line (DSL) and cable modem technologies. However, the term now generally applies to any Internet communications that are faster than traditional modems, which are limited to 56 kilobits per second.

DSL lines are essentially an extension of the standard telephone lines to your house. While telephone wires are not *officially* supposed to handle data above 3.4 KHz bandwidth, most new telephone wires actually can handle that kind of data. Therefore, digital data can be encoded into an analog electrical signal above 4 KHz and transmitted to the phone company without disrupting the normal phone conversation. The most popular DSL variant is *ADSL* (the "A" stands for *asynchronous*, because it allocates more bandwidth for downloading than uploading), which uses the band of 25 KHz to 160 KHz for its upstream, and 240 KHz to 1,500 KHz for its downstream.

Unfortunately for me, DSL technology came too late. The year before DSL was standardized, my telephone company installed a digital switch in my neighborhood that encoded the telephone data into a digital stream of data and sent it to the phone company via a fiber-optic cable, ignoring any data outside of the standard telephone range of 400 Hz to 3.4 KHz. Therefore, DSL cannot be used in my neighborhood because the phone technology is too advanced for DSL. Talk about irony!

Cable modems work in a similar way to DSL, except they use the unused bands of the coaxial cable that goes into your house, instead of your telephone line. Cable modems typically use the band of 5 MHz to 65 MHz for upstream data and 850 MHz to 1000 MHz for downstream data. Cable modems use much more bandwidth for their signals, because coaxial cables are typically a lot longer than telephone cables, and signals on them are weaker.

Mechanics of Packets and the Internet Wonderland

The Internet is a very cool thing, but I'm sure you already knew that. I want to show you how packets actually work, so you can appreciate even more how wonderful the Internet is.The basis of the Internet lies in the *Internet Protocol* (*IP*). This protocol was invented in 1981, which really wasn't that long ago in the grand scheme of things.

The whole idea of the IP protocol was to define a standard method of communication among routers, switches, and nodes on a network. Basically, every chunk of data that is sent is prefixed with a *header*, also known as the *IP header*. Two versions of IP exist today: IPv4 and IPv6.

All About IPv4

Figure 1.8 shows the standard layout of an IPv4 header.

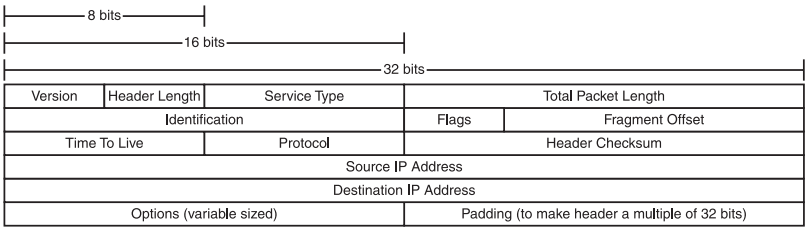


Figure 1.8

This is a standard IPv4 header.

The v4 means *version 4*. IPv4 is the current IP standard across the world, but there is a newer version called *IPv6*. (What happened to IPv5? Who knows? It's probably having a party along with DirectX 4.) I'll get into IPv6 later on, since it's not used too much—yet.

You don't have to understand all the little details of the header; they're really important only to network engineers. I'll go over the important points, though.

The first thing you should know is that the length of the header is variable. Everything up to the Options parameter (bottom row of the figure) is set in stone, but the Options parameter is variable. That is why there is a parameter that holds the length of the header (the Header Length parameter); any data past the header is the data stored in the packet.

The Total Packet Length parameter describes the entire length of the packet, in bytes, including the header. Since it's 16 bits long, an IP packet can be at most 65,535 bytes long.

The Time To Live (TTL) parameter is particularly interesting; it determines how long, in jumps, the packet lives. This prevents packets from accidentally being routed around in circles forever. Every time a packet passes through a new router, the TTL field is reduced by 1, and when it reaches 0, the router completely discards the packet. The field is 8 bits, so there can be at most 255 hops between routers before a packet is completely discarded. The 255 hops is an incredibly large number, so it is reasonable protection.

The Protocol parameter determines which protocol is being used on top of the IP header. Only a few kinds of protocols operate on top of IP. As a games programmer, you should mainly pay attention to two:

- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)

I describe these protocols in more detail later on. The Header Checksum parameter is an important data integrity measure in IPv4. A *checksum* is a value that represents the data and is computed by a checksum algorithm. The checksum is a simple measure that verifies if data has been changed in the transmission. Whenever a router receives an IP packet, the packet's checksum is calculated and compared to the existing checksum value in the packet. If the numbers match, you can be reasonably certain that the data has not changed; if the numbers don't match, you know the data was somehow changed by an error or interference in the communication path. Whenever the checksums don't match, the router immediately discards the packet. You'll see why this is a good idea later on. Also, since the TTL parameter is changed at every router, the checksum is recalculated whenever a router passes a packet on.

Finally, the two most interesting parts of an IP packet are the source IP address and the destination IP address. Every node on an IPv4 network is given a 32-bit IP address, typically represented as four numbers, separated by periods, like this: 192.168.100.5.

The original Internet addressing scheme classified all addresses into three groups: large, medium, and small networks. This system was wasteful and isn't used much anymore, so I won't waste time describing it.

NOTE

Under the old addressing system, organizations such as the University of California At Berkeley were given more IP addresses than the entire country of China. You can see how the old system just isn't going to work anymore, especially when you consider that Berkeley only has a few thousand people, and China has 1.2 billion.

Using 32-bit addresses limits the number of total nodes on a network to a little more than 4 billion, which used to seem like a large number, but it seems smaller and smaller every day. There are already many more than 4 billion people on this planet, so giving every person his own IP address is not even possible anymore. This was the major concern for upgrading the system to IPv6, which I will touch on next.

IPv6: Bigger and Better

IPv6 was created in 1995, when the Internet community realized that IPv4 was too constrained. The biggest problem, by far, was the small address space allocated to IPv4. As I've said before, 4 billion addresses is simply not enough to identify all the computers on the planet now or in the future. Most large ISPs dynamically assign an IP address to a customer when he logs on and reuse that number for another customer once he logs off. This method isn't useful anymore, as most people are starting to realize the importance of

permanent Internet addresses. How would you like a phone number that changed every day? In addition, as broadband connections are becoming the standard, more and more people are staying online longer and longer, making dynamic IP allocation less workable.

And finally, there are going to be thousands of devices in the future that will need their own IP addresses. IBM has even promised refrigerators that can connect to the Internet, and gosh darn it, I want them now!

Besides the larger address space, IPv6 has a host of new features making it more streamlined and functional, and even better security features have been added. However, if you're interested in those features, you should get a networking book, because this stuff isn't that important to game programming.

Figure 1.9 shows a diagram of an IPv6 header.

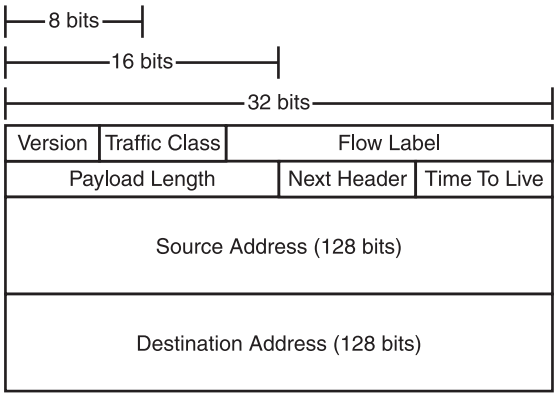


Figure 1.9

Notice that there are fewer fields in this IPv6 header than in an IPv4 header.

IPv6 has been simplified, and the rarely used portions of the IPv4 header have been removed. The other big change is the huge addresses; IPv6 addresses are 128 bits long. That means that there are a total of 3.4×10^{38} addresses available. That's 340 undecillion addresses, and when you haven't even heard of a number before, that means you've got enough addresses. But to further illustrate my point, I'm going to show you some more pointless calculations that illustrate the sheer size of the IPv6 address space.

At any rate, there should be enough addresses for at least the next few hundred

NOTE

The surface area of the earth, water included, is 5.1×10^{14} meters squared. If you divide 3.4×10^{38} by 5.1×10^{14}, you get 6.6×10^{23}. That means that there are around 660 sextillion IPv6 addresses available for *every square meter of space on the planet*. That ought to be enough for anybody—until we decide to give IP addresses to every molecule on the planet. Or until we provide free Internet to the multiverse.

years, so we'll let programmers find more addresses when the time comes. For now, there are places on the Internet where you can obtain a block of a few million—or even billion—IPv6 addresses.

The rest of the fields of an IPv6 header are pretty much the same as the important fields in IPv4, with updated names. They're not really important.

IP Philosophy and Layered Hierarchy

IP packets are not guaranteed. When you send an IP packet, you have absolutely no idea if it will be received. Even worse, you can't tell if a packet has reached its destination.

Doesn't that sound like an incredibly stupid way to design a network? Maybe so, but think about it for a moment. Imagine how much more complex the routing hardware would have to be to ensure that every IP packet arrived intact at its proper destination. Right now, the routers don't care; they take the data and keep relaying it on until it either gets to its destination, or they discard the packet as junk. Simple hardware is cheaper to build, and faster as well.

So, what is the point of unreliable communications? With digital data, even one byte missing out of a file can make the entire thing useless, so unreliable communications seems like the opposite of what you'd really want!

Instead of letting hardware control integrity and validation, the IP model puts software in charge. Before going into this topic, I want to explain the layered hierarchy of Internet communications.

Internet protocols are actually designed into four distinct layers:

- Network layer
- Internet layer
- Transport layer
- Application layer

Each time you send a packet of data over the Internet, it is encapsulated by a new header at each layer. For example, if you are sending *Hyper Text Transfer Protocol* (*HTTP*) data, which is basically web-page data, the data you send is first enclosed into an HTTP header at the Application layer. Then, the HTTP application sends the packet to the operating system, which adds a Transport layer packet header as well as an IP packet header for the Internet layer. (HTTP uses TCP as the Transport Layer Protocol. And I will get to TCP in a bit.) Finally, depending on what device you use for Internet access, a Network layer packet header is added to the packet, and finally it is sent on its way. Figure 1.10 is a pictorial representation of the layered hierarchy of Internet packets. This particular example demonstrates browsing the WWW with an Ethernet connection.

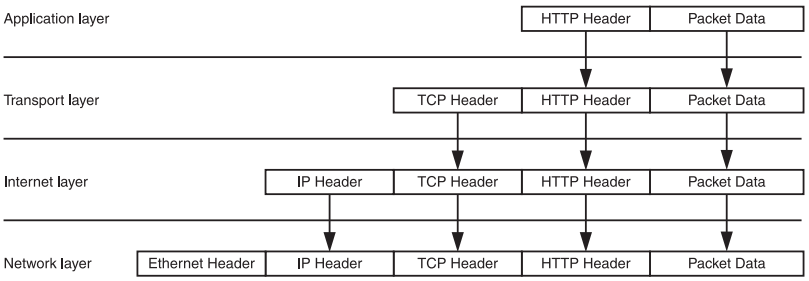


Figure 1.10

In this standard four-layer packet structure, each packet of data is prefixed with the header of the various protocols it uses.

Previously, I told you all about the IP protocol. This protocol makes its home in the second layer, the Internet layer. I started there first, because it is really the most important layer when dealing with Internet communications. Now I will go into more depth on the layers themselves.

Network Layer

The lowest layer is called both the *Network layer* and the *Physical layer*, because it is the layer that is added to a packet whenever a physical device sends the data. Examples of this include an Ethernet card (Ethernet Protocol), a modem (PPP Protocol), a wireless Internet card (802.11b protocol), or a cable modem (DOCSIS protocol). Each of these devices operates in a different way and has its own header format depending on its needs.

The great thing about the layered protocol system is that the physical devices *don't care* what kind of data you are sending over them, so you can send any kind of data, as long as the recipient of the data expects it and knows how to decode it.

Internet Layer

The *Internet layer* is perhaps the most important layer, since every device in a single network must understand and recognize it. The primary purpose of the Internet layer is to provide routing and addressing services, so the routers know where to send packets. A network can use many different kinds of devices, such as modems and Ethernet cards, and as long as they all understand the Internet layer protocol in use, the network should operate perfectly.

Of the three major Internet layer protocols, I have described the two most commonly used: IPv4 and IPv6. In the past, a third protocol, called *IPX* or *Internetwork Packet Exchange,* was widely used as well. IPX is superior to IP in a few ways, but it never really caught on and is pretty much dead today. One of IPX's notable characteristics was that it had a segmented address space. It had 32-bit network addresses, and each network also had a 48-bit node address, essentially using 80-bit addresses.

Transport Layer

I haven't talked much about this layer yet, but it is important. The *Transport layer* accommodates protocols such as TCP, UDP (explained in a section that follows) and Internet Control

Message Protocol (ICMP). These protocols are primarily designed to handle connections, rate of data transmission, and data integrity verification.

For example, as I mentioned before, if you use the IP protocol, you have absolutely no idea if the packet you sent reached its destination. To solve this problem, you need to have the Transport layer protocol handle the transmission.

For example, when you wrap your data into a *TCP* (*Transmission Control Protocol*) packet, TCP calculates the checksum of all of the data, and then your operating system wraps the entire TCP packet into an IP header and sends that out.

When the recipient of your packet gets the data you sent, it sends an *acknowledgement* (ACK) packet over TCP, saying that it got the packet. It may, however, fail to receive your message for a variety of reasons. If the original TCP packet gets lost, for example, the ACK packet would never have been sent, or the ACK packet itself may have gotten lost. If the sender doesn't receive the ACK packet, the sender sends the data again, and keeps sending the data until he receives a confirmation that the data has been sent. Here's a simple listing of the process:

1. Send packet.
2. Wait for ACK.
3. If no ACK in given amount of time, go back to 1.
4. Send next packet.

The hardware costs for this method of data verification are far lower than making the IP protocol itself check the integrity of data and respond to the sender that there was an error. The routing hardware doesn't really care much about acknowledgements; instead, it just assumes the communications succeeded, and it lets the end computers figure out if something went wrong. The reason this works is because the number of times data transmission fails is far fewer than the number of times that the transmission is successful, so there's really no point in making *every node* along the transmission path check that it is successful, and send errors backward along the path.

There is a slight chance of data transmission error, and that slows things down a little bit, because the sender keeps trying to send the data; but, in the end, that is a far more desirable solution than having incredibly expensive routing hardware.

The *User Datagram Protocol (UDP)* elects to forego the data integrity issues and opts instead for speed over integrity; in other words, delivery is not guaranteed. For this reason, UDP is quite often preferred over TCP for very fast games, such as first person shooters. I won't cover UDP in detail in this book because it is not an important protocol for low-speed MUDs. I'll be sticking with TCP, which is a little slower but more robust and has guaranteed delivery.

Application Layer

The *Application layer* is theoretically the highest layer of a packet header, and it contains information about the specific application you are using with the packet. Examples of popular Application layer protocols include HTTP, File Transfer Protocol (FTP), Telnet,

Simple Mail Transfer Protocol (SMTP), and so on. The topics in this book focus almost entirely on creating and using application layer protocols for MUDs.

Other Layers

The four-layer model is really just a recommendation for networking; it's not a necessity. In the past, some crazy people have demonstrated this fact using completely useless technologies. For example, there is an IP over SMTP protocol, which defines how to send IP packets over SMTP. Of course, SMTP is built on top of TCP, which in turn is built on top of IP, so what is the point? Who knows? Never underestimate what a nerd and some free time can accomplish. After all, as game programmers, who are we to judge?

It is usually accepted to use the slash notation (/) to show protocol layering. For example, you may have heard of TCP/IP. This means that you are using TCP over the IP protocol. It is literally pronounced "TCP over IP." Slash notation, however, is not common for other combinations, because the entire idea of networked communications is to keep the layers as independent as possible. That way, you can easily use higher protocols over different lower protocols. This is why you don't see people saying "HTTP over TCP over IP." Not only is it a mouthful to say, but you aren't really required to send HTTP over TCP anyway (though I've really never seen anyone who doesn't).

Common Transport Protocols

As a game programmer, you usually won't pay attention to the IP protocol; the operating system should take care of that for you automatically. You'll only be slightly more interested in the TCP and UDP protocols, since most compilers have built-in libraries to handle these protocols. I'll start with UDP first, since it's simpler.

UDP

UDP, as I've said before, is the User Datagram Protocol. A datagram is basically just a single packet of data. The UDP protocol is simple and doesn't offer the reliability of more complex protocols, such as TCP. Essentially, you just send the packet out and hope it gets there. This is a "fire and forget" protocol. Figure 1.11 shows the UDP header format.

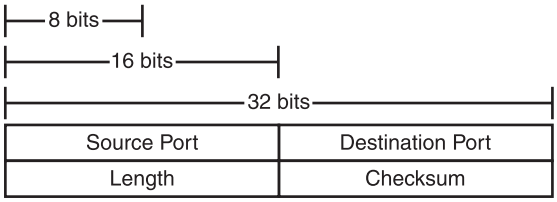


Figure 1.11

The header format for a UDP packet.

NOTE

The port fields for UDP (and as you'll see shortly, TCP as well) are 16 bits long. This means that a total of 65,536 ports are available for use. Ports below 1,024 are reserved for specific application-level protocols assigned by the *Internet Assigned Numbers Authority (IANA)*. To see a list of these ports, you can visit their Web site at http://www.iana.org. IANA covers things such as HTTP (80), FTP (21), Telnet (23), SMTP (25), as well as hundreds of other protocols that no one has ever heard of. You should generally try to keep your programs' port numbers above 1,024. If you are not running as root, UNIX-based systems won't even allow you to open ports below 1,024 (for servers). Table 1.2 shows a listing of common port numbers.

The first thing you should notice in Figure 1.11 is that the header is only 64 bits long, or 8 bytes. That's pretty small for a packet header, at least compared to other protocols.

Next, notice the two *port* fields. You see, once a packet gets to its destination, there really is no way for the receiving machine to figure out what program the packet is trying to get to. Therefore, the idea of ports was invented. When a port receives a packet, the operating system is supposed to read the port number off the packet and send the packet to the appropriate program. This way, you can have many different programs on the same machine, all using the network connection at the same time.

In Figure 1.11, you should also notice the length and the checksum fields. The length tells you the length of the packet data, including the header. The checksum field contains the checksum of the data in the packet, so that the receiving machine can figure out if the data is intact. If it isn't, the receiver just discards the packet altogether and acts as if it never got it.

UDP is a *connectionless* protocol. This means that UDP programs don't connect to each other; they just send the packet, and the server is supposed to accept it. Other protocols, especially TCP, will not accept incoming packets unless you explicitly connect to the other end first.

The fact that UDP does not guarantee delivery of packets can lead to problems. In a fast-paced game in which the server constantly sends the clients updates on the positions of other players, guaranteed delivery is not a great problem. If, for example, a position update packet is sent but never delivered, a reliable protocol like TCP will keep trying to send the packet; but by the time the protocol finally sends the original packet, the player's position may have changed. So in this case, UDP is a useful protocol.

Table 1.2 Common Ports

Port	Service	Purpose
17	QOTD	Quote of the day; sends a quote in text form
20	FTP Data	FTP data port
21	FTP Control	FTP control port
22	SSH	Secure Shell Terminal (a secure version of Telnet)
23	Telnet	Allows terminal control
25	SMTP	Simple Mail Transfer Protocol
37	Time	Sends the server time
53	DNS	DNS lookups
80	HTTP	World Wide Web pages
110	POP3	Post Office Protocol; more mail stuff
113	Ident	Identifies the name of a computer
119	NNTP	Newsgroups
143	IMAP	Another old mail protocol
6666	IRC	Internet Relay Chat
31415	PIE	Pieserver; it serves digits of pi*

* See my website at http://ronpenton.net/projects for more information on Pieserver.

But what happens with important data? What if something happens in a game, and the game is set up so that it won't retransmit that data later on? You could end up with your clients completely missing an important game event such as a gunshot and then getting out of sync. In this case, UDP isn't a very useful choice.

For MUDs and MMOGs, using UDP usually isn't a good idea, since most things that happen in these kinds of games are *event based*—that is, events occur once and the client absolutely needs to know they happened.

TCP to the Rescue!

TCP is probably the most highly used transport protocol, since it guarantees data delivery. If you tell your TCP library to send data, it *will* get there, barring any unusual events such as a nonexistent destination. Without TCP, file transfers and reliable communication over the Internet would be virtually impossible.

In contrast to UDP, TCP is a *connection-oriented* protocol. This means that the client must tell the server that he wants to connect, before the server will even listen to incoming data.

TCP is also a *streaming* protocol. This means that the protocol attempts to send streams of data, separated into packets for delivery over a packet-switched network. This is an important part of TCP, since it ensures not only that the data actually reaches its destination, but also that the data arrives there in the same order in which it was sent.

NOTE

Because of the decentralized nature of the Internet, two packets you send from one place to another may follow two completely different paths, which means that you can't be sure that sending one packet first will mean it will arrive first. This makes TCP a great way to make sure that the connections receive their data in order.

I've told you about TCP and the acknowledgement packet that it uses. Since this is important, here's a quick recap. Whenever a TCP port receives data, it sends an acknowledgement packet saying that the data was received. If the original sender never gets an acknowledgement, then the TCP port attempts to send the data again. Of course, this process is inefficient if the sender sends one packet and then waits for the acknowledgement before sending anything else.

That isn't how TCP actually works, though. TCP starts off sending all the packets it needs to in order, and just continues sending until there is nothing left to send. If TCP realizes that an acknowledgement packet hasn't come back for a packet it sent, it stops what it is doing and attempts to retransmit the packet that wasn't acknowledged.

On the receiving end, if the receiver detects that it is getting a packet out of order, it buffers the data in that packet until the packet or packets that are supposed to precede it arrive.

At the application level, all of these operations are transparent. Your TCP library handles all of this for you and makes sure you get the data in its intended order.

Unfortunately, all of these safeguards come with a price, as you see when you examine the TCP header shown in Figure 1.12. Note that the TCP packet header is much larger than a UDP header.

TCP is a feature-rich protocol. It includes not only the kitchen sink but the disposal too. The *minimum* size of a TCP header is 20 bytes, much larger than the 8 byte UDP header.

TCP uses the same port numbering scheme as UDP, 16-bit ports, which adds up to 65,536 ports. Like UDP, TCP has a checksum field, which is used for data integrity.

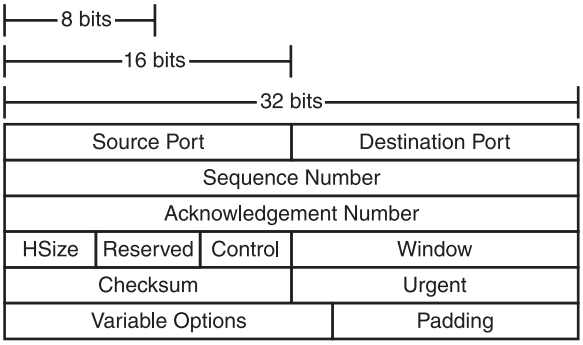


Figure 1.12

The standard for TCP packets.

The other fields you should at least note are the sequence, acknowledgement, window, and urgent fields. The *sequence field* denotes the position of the packet in the current stream at its transmission point; this is used so that the receiving end can piece together the packets if they arrive out of order. The acknowledgement field tells the receiver the acknowledgement number that the sender is expecting.

The window field is somewhat interesting. TCP implements flow-control mechanisms, which means that each side of a TCP connection can tell the other side how much data it is willing to accept. This is useful for preventing a connection from accidentally sending more information than the other side can handle.

Notifying the other side on acceptance limits is also particularly useful whenever there are dropped packets. Since TCP buffers data that is out of order, it may be useful for the receiver to tell the sender to stop sending data until it catches up. Buffered data can take up lots of room, since the TCP library can't do anything with that data until it gets all previous packets.

Finally, you should be aware that TCP supports a concept called urgent data, which the urgent field handles. *Urgent data* should not be used inside the data stream, but contains important connection and control information. The TCP library you are using should seamlessly strip this data out of the stream and take care of it automatically.

That sums up all the important things you as a programmer need to know about TCP. If you're interested in learning more about either UDP or TCP, networking books can do the trick. I just wanted you to know the basic mechanics of how these technologies work as they affect game programming.

Information on Networking Protocols

There is one important part of networking that I have neglected to mention: the standard documentation for all published networking protocols. Early on in the development of

NOTE

I'll let you in on an inside joke. There are many funny RFCs in the general RFC database. It is kind of an Internet tradition to submit one of these every April Fools' Day. For example, RFC 1149 is officially entitled "A Standard for the Transmission of IP Datagrams on Avian Carriers," which basically documents a method of transmitting IP packets using carrier pigeons. RFC 2324 is entitled "Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)" and defines a method of controlling coffee pots over HTTP. More recent is RFC 2795, "The Infinite Monkey Protocol Suite (IMPS)." I don't even want to know what that's about. We programmers have a strange sense of humor.

ARPANet, engineers recognized the need for a formal way to publish the standards and specifications of protocols. They eventually called the documents they created *RFCs*, which stands for *Request For Comments*.

You can easily look up RFCs by using their published identification numbers. For example, the current RFC describing IPv4 is RFC 791. You can search for that with any Internet search engine, and you'll get hundreds of links. RFCs are public documentation, and they're free to be published anywhere.

Once RFCs are submitted to the world, they can never be changed. If a protocol needs to be changed, the old RFC is *deprecated*, which means that it is no longer current, and a new RFC with updated information is published.

My favorite place for getting RFC information is at a website entitled *Connected: An Internet Encyclopedia*, which is located at http://www.freesoft.org/CIE/. If that site is down, you can probably find a mirror, as it is very popular. The site is fairly well updated with all the RFCs, and it's even got useful courses and background material for most of the networking topics I haven't covered here.

Summary

I hope you found this chapter interesting. I have a passion for history, and I feel that knowing your history is a good way of understanding why things are the way they are, what has succeeded and failed in the past, and where we can go in the future. Packet networking is a new development in the grand scheme of things, and we're still pioneering the field, so I think it is appropriate to know this material.

As I've said at the start of the chapter, you don't have to be an expert in something to use it, but it helps a lot if you at least know some of the mechanics. I hope you now understand the basics of how the IP, TCP, and UDP protocols work, and how networks work in general. This knowledge should pave the way to the next chapter, "Winsock/Berkely Sockets Programming."