Overview of Computer Networks

Norman Matloff
Dept. of Computer Science
University of California at Davis
©2001-2005, N. Matloff

April 11, 2005

Contents

1 Significance of Networks **3**

1.1 History 3

1.2 What Are Networks Used For? 3

1.3 Which Aspects of Networks Are Important to Know? 3

2 An Introductory Example **3**

3 The Famous, Overrated But Useful 7-Layer Model **5**

3.1 Overview of the Layers 5

3.1.1 Physical Layer 5

3.1.2 Data Link Layer 6

3.1.3 Network Layer 6

3.1.4 Transport Layer 6

3.1.5 Session Layer 7

3.1.6 Presentation Layer 7

3.1.7 Application Layer 7

3.2 How the Layers Interact 7

4 More on TCP/IP **8**

4.1 TCP/IP Overview 8

4.1.1 TCP . . . 9
4.1.2 UDP . . . 9
4.1.3 Stream Vs. Datagram Communication . . . 10
4.1.4 IP Addresses . . . 11
4.1.5 Peer Communication . . . 11
4.1.6 Viewing Current Socket Status . . . 11
4.1.7 What Makes a Connection Unique . . . 12
4.2 Sample TCP/IP Application: NFS . . . 12

5 Network Programming 13
5.1 TCP Socket Example . . . 13
5.1.1 Source Code . . . 14
5.1.2 Who Shall I Say Is Calling? . . . 19
5.2 UDP Socket Examples . . . 20
5.2.1 Basic Example . . . 20
5.2.2 Advanced Use of Sockets . . . 21
5.3 Nonblocking I/O . . . 23
5.4 Debugging Client/Server Programs . . . 23

6 Packet/Frame Formats 24
6.1 TCP . . . 24
6.2 IP . . . 24
6.3 Ethernet . . . 25

7 Putting It All Together 25

8 Application-Layer Protocols 27

9 Routing Issues 28

1 Significance of Networks

1.1 History

The key to the computer revolution of the 1980s was the invention of the microprocessor in the 1970s. This gave rise to the cheap computers which have become ubiquitous in homes, schools and offices, and to the **embedded computers** which serve as controllers inside cameras, cars, washing machines and so on.

The computer revolution of the late 1990s, now continuing int the 21st century, involves computer networks, whose existence is enriching our society in countless different ways.

1.2 What Are Networks Used For?

A simplified but worthwhile description of the uses of computer networks might be as follows:

- Sharing of hardware: For example, several PCs might be networked together in a wired or wireless **local area network** (LAN) to share a printer.
- Sharing of information: **Distributed databases**, e-mail, the World Wide Web and so on are examples of this. Here the sharing involves both LANs and **wide area networks** (WANs), especially the latter.

1.3 Which Aspects of Networks Are Important to Know?

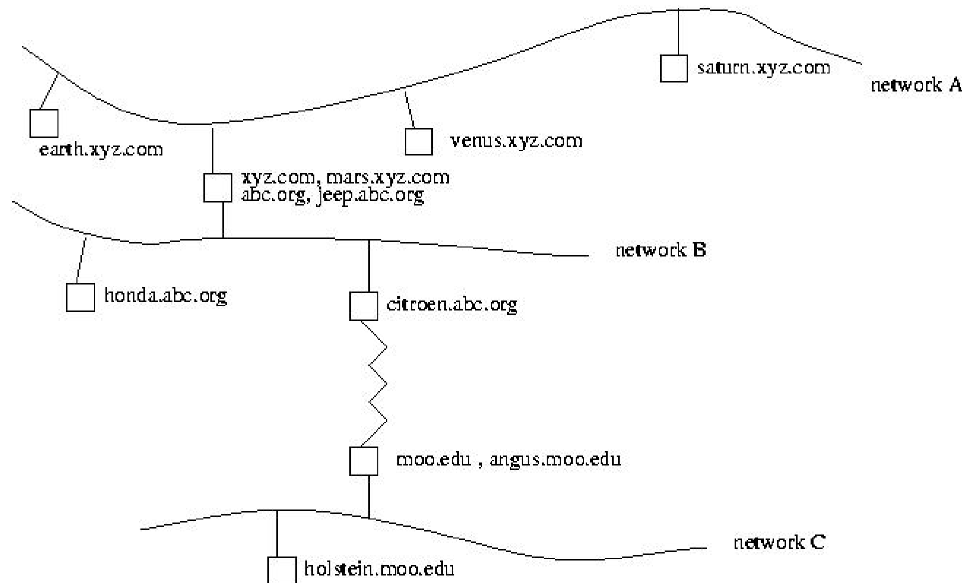
Networks touch upon virtually every aspect of computing today. A good working knowledge (i.e. not of the "partial credit on an extra problem" type) is of enormous value.

It should be noted carefully that it is important to know both the software and non-software aspects of networks—the latter meaning the hardware, the protocols,[1] the quantitative performance issues and so on. If you interview for a job involving networks, you can expect that at least half of the questions will be on the non-software aspects, even if the job itself deals mainly with software. Moreover, many jobs you might interview for might be as system administrators; in such positions, knowing the physical structure of networks is just as important as knowing how the network software works.

2 An Introductory Example

Consider the following structure:

[1] A **protocol** is simply an organized set of rules for performing some task, usually with two entities cooperating with each other.



Here we have three individual LANs, labeled A, B and C, tied together in an **internet**. (Note the lower-case 'i'.) The address names of the nodes here, such as **saturn.xyz.com**, use the IP format which will be discussed later. This internet should not be confused with "the Internet," distinguished by the capital 'I'. The Internet is a special internet connecting TCP/IP networks worldwide.[2]

Each node here is a computer, connected to one or more networks via one or more network interface cards (NICs). Recall that in any computer, each I/O device's interface has a specific **port number** or **address**. The keyboard interface, for instance, might be port 50. Whenever a program, usually the operating system (OS), reads from the keyboard, the value 50 would go onto the computer's address bus. Similarly, a NIC has an address too, say 60. Whenever the OS reads from or writes to the LAN, it does so via the NIC, using the NIC's assigned address.

This can be confusing, because there will be several addresses at work here:

- The NIC address from the CPU's point of view is 60, in the sense that the NIC is connected to the computer's address bus at that address.
- The NIC also has a LAN address, typically a 48-bit Ethernet ID number hardwired into the NIC at the time of manufacture and unique among all NICs in the world of that LAN type.
- If the computer is part of the Internet, the NIC will be assigned an Internet (IP) address, a 32-bit identifier unique among all Internet hosts in the world.[3]
- At the TCP layer of the network software, we also speak of **ports** which are purely software IDs that have nothing to do with port numbers like 60 above. In essence, each TCP port is a different element in a large array in the TCP software, each element corresponding to a different service such as **sendmail** (a program that actually puts your outgoing e-mail onto the network, and receives incoming mail for you, as opposed to the program you use to compose and read mail with), **ftp**, **ssh** and so on.

[2]Another similar term is **intranet**, which refers to a private collection of networks, say within a particular company, which we do not wish to make accessible worldwide.

[3]Extended to 128 bits in IP version 6.

Suppose I buy an Ethernet card and install it in **saturn**, but later swap it with the one in *citroen*. Then the card's Ethernet ID will still be the same, but its IP address will change, and maybe its I/O port number (previously 60) will change too.

A computer can be connected to more than one network, as is the case with **mars.xyz.com here**. For each NIC a given computer has, the computer would answer to a different name. In addition, a computer can even have more than one name associated with the same NIC. Here the machine has the names **mars.xyz.com** and **xyz.com** associated with the interface to network A, and its names **abc.com** and **jeep.abc**.com are associated with network B.

Basically, TCP/IP is ubiquitous today for general-purpose usage. But there are other protocols for special-purpose usage, such as one designed for fast communication in parallel processing applications.

Since the protocol ID (e.g. 0x0800 for TCP/IP, 0x8137 for Netware, etc.) will be placed into the bit frames going out onto the Ethernet, each computer on the network will be able to determine what protocol an incoming message was sent under. This is especially useful if the same computer is running two or more protocols. An incoming message from the Ethernet at the computer causes a hardware interrupt, and the interrupt service routine can then pass on the message to the proper protocol software, according to the protocol ID specified in the Ethernet frame.

The jagged line between **citroen.abc.com** and **angus.moo.edu** is a phone line, dedicated to communication between the two machines, operating 24 hours a day.

So you can see that there are two ways that we can connect two LANs together:

- We can install a computer and connect it to both LANs.
- We can put in a phone line (or microwave link, etc.) between a computer on one LAN and a computer on another LAN.

3 The Famous, Overrated But Useful 7-Layer Model

Every network textbook includes a picture of the famous "seven-layer" model. Actually, this model is vague, and it does not always correspond to specific portions of specific networks. Nevertheless, it serves as a useful overview of the field. Here is how some of the layers relate to our sample network above.

3.1 Overview of the Layers

The layers collectively are often referred to as the **protocol stack**.

3.1.1 Physical Layer

This is concerned with the nature of the physical media (metal or optical cable, free-space microwave, etc.) used to send signals, the nature of the signals themselves, and so on.

There is also the question of signal form; the signals themselves may be in the form of pure 0-1 bits, or may be in the form of certain frequencies. In addition there are questions concerning how a receiver distinguishes two bits which are adjacent in time.

A major issue is the form of the medium, both in terms of the materials it uses and its topology. A basic wired Ethernet, for example, consists of cable conducting electrical signals; the connections could also be wireless. More complicated networks, including Ethernets, may consist of more than one cable, with all of them connected via a **hub**. The latter has become common even at the household level.

3.1.2 Data Link Layer

For example, in an Ethernet, this layer is concerned with ensuring that two network stations connected to the same cable do not try to access the line at the same time.[4] For this reason the Ethernet operation is an example of what is called a **Medium Access Control** (MAC) Protocol.[5]

Here is an overview of how the Ethernet MAC protocol works, using a "listen before talk" approach. When a network node has a message ready to send, it first senses the cable to see if any node is currently sending. If so, it generates a random **backoff time**, waiting this amount of time before trying again. If the node does not "hear" any other node sending, it will go ahead and send.

There is a small chance that another node actually had been sending but due to signal propagation delay the transmission had not yet reached the first node. In that case a **collision** will occur, destroying both messages. Both nodes will sense the collision, and again wait random amounts of time before trying again.

This layer also does the setting up of **frames** of bits (i.e. sets of consecutive bits sent along the wire), which not only include the message itself but also information such as (say, in the Ethernet case) the Ethernet ID number of the destination machine.[6]

Messages may be broken up into pieces before being sent. This may be handled at the transport level (see below), but may also be done at the data link level.

3.1.3 Network Layer

This is the routing layer. Questions addressed in this layer include: If in our example above **saturn** wants to send a message to **holstein**, how is that accomplished? Obviously its first step is to send the message to *mars*; how does **saturn** know this? How can alternate routes be found if traffic congestion occurs?

3.1.4 Transport Layer

Suppose **saturn**'s message to **holstein** consists of a large file transfer, say 100 megabytes. This transfer will take a long time (by network standards), and we certainly don't want it to monopolize the network during

[4]In a small Ethernet, we would not even have two nodes on one cable; we would just have each station connected to a different port in the hub.

[5]And the address of an Ethernet or other LAN card is known as the **MAC address** of the card.

[6]This phrasing implies that the frame will be sent only to another machine on the same Ethernet. This is true, but it may be that the frame's ultimate destination is on another LAN, and the current Ethernet destination ID is for a machine which plays the role of a intermediary router to other LANs. See the description of the Network layer below.

that time. We also must deal with the fact that the buffer space at **holstein** won't be large enough to deal with a 100-megabyte message. Also, one 100-megabyte message would have a sizable probability of having at least one bit in error, and if so, we would have to retransmit the entire message!

So, the file transfer must be done in pieces. But we don't want to burden the user at **saturn** with the task of breaking up the 100 megabytes into pieces, nor do we want to burden the user at **holstein** with the reassembly of the messages.[7] Instead, the network software (again, typically in the OS) should provide these services, which it does at the transport layer, as for example is the case with TCP.

3.1.5 Session Layer

This layer is concerned with management of a **session**, i.e. the duration of a connection between two network nodes. The word *connection* here does not mean something physical, but rather refers to an agreement between two nodes that some chunks of data with some relation to each other will be exchanged for some time. Actually, TCP does this in some senses, as does the **socket** interface to TCP, which is very much like the interfaces for reading or writing a file (described in more detail later).

3.1.6 Presentation Layer

This layer deals with such matters as translating between character codes, if the source uses one and the destination the other. In the old days, this could mean ASCII at one end and EBCDIC on the other end. Today, though, it could mean for example two different coding systems for Chinese characters, Big 5 and GB.

3.1.7 Application Layer

You can write programs at the application layer yourself, and of course you use many programs written by others, such as **ftp**, Web browsers, e-mail utilities, and so on.

3.2 How the Layers Interact

The Physical Layer is obviously implemented in hardware. So is the Data Layer, in the sense that the NIC will handle this layer and is hardware.[8] These days the Network Layer is also usually hardware; as noted above, Ethernet hubs are now common in the home, and include routing capabilities. The Transport Layer and above are usually implemented in software, actually as part of the OS.

Information is communicated from one layer to the next.[9] For instance, think of the file-transfer example presented earlier. The file-transfer program, say **ftp**, works in the Application Layer. It will call socket service functions in the Session Layer, such as the **socket()** function, which opens a network connection in a manner very similar to the **open()** function which opens a file. These functions will in turn call functions

[7] The "users" here are the application programs being run, e.g. **ftp** at **saturn** and **ftpd** (the FTP server) at **holstein**.

[8] More technically, *firmware*—software stored in ROM.

[9] In some cases a layer is "skipped." At the Session Layer, a **raw socket** can be opened, in which the socket communicates directly with IP in the Network Layer.

for TCP operations in the Transport Layer, which will themselves call functions for IP operations at the Network layer. The latter will then—say we are on an Ethernet—issue machine instructions (e.g. IN and OUT in the case of Intel CPUs) to the Ethernet NIC, which will use the Data Link and Physical Layers to put frames out onto the LAN. At any given layer, a function passes the message in a **packet** of bits to the next lower layer. The packet grows larger at each layer, because each layer adds more information.

Say we are using the **put** command in **ftp** to copy a file named **zyx** to the destination machine. When we do this, **ftp** calls the **write()** function to send data,[10] A typical call to **write()** will contain the actual data to be transferred, in this case part of **zyx**. At this point the packet consists only of the data, the socket number and the number of bytes of data, and will be handed over to TCP.

TCP will then add to the frame the TCP source and destination port numbers, packet **sequence numbers** (when a long message is being sent in small pieces, each piece gets a sequence number to identify it), and so on, and then pass the packet to IP.

IP will add to the packet a code indicating the fact that this is a TCP packet (as opposed to UDP, another type of communication service offered in the TCP/IP protocol), plus the source and destination IP addresses, and so on, and pass the packet to the NIC.

In for example the Ethernet case, the NIC will then add to the packet the source and destination Ethernet addresses, a code indicating that this message uses the TCP/IP protocol suite.

The packets sent at the physical level have a special name, **frames**.

When a frame reaches the destination machine, a mirror image of the above process occurs. The packet will now travel up the protocol stack, and will shrink as it does so. In the **ftp** example, the "top" of the stack will be **ftpd**, which is the "partner" program of **ftp** running on the destination machine.

Note also that as the packet gets routed through intermediate machines on its way to the destination, at each of these intermediate machines it will travel up the protocol stack to the Network Layer (i.e. IP), which will check to see whether it has reached the destination, and then upon finding that it hasn't, it will be sent back down the stack for transmission to the next machine in its path to the destination.

4 More on TCP/IP

TCP/IP is a very complex system, the subject of numerous thick books, and we cannot go into detail on it in this document. We will give only a short introduction.

4.1 TCP/IP Overview

A famous and very common network protocol is TCP/IP. It was originally invented as part of the UNIX OS, and later became the basis for the Internet (the Internet was developed mainly on UNIX machines). For the latter reason, it is now part of other OSs, such as Windows.

[10]We are "writing" to the socket, as if we were writing to a file, but the effect is to send the data through the network. This is a nice feature in UNIX, as it gives us a uniform interface to both files and sockets. In Windows systems, this is not available and thus use, for example, the **sendto()** function instead; this function is also available in UNIX.

TCP/IP actually includes two protocols at the transport level, TCP and UDP, and one at the network level, IP.

4.1.1 TCP

TCP is a **connection-oriented** protocol. As mentioned earlier, the term *connection* does not refer to a physical connection, but rather to a temporary agreement set up between the source and destination nodes concerning the processing of a sequence of ordered packets, such as the sizes of the pieces of a file sent during a file transfer. The TCP Layer at the destination end will:

- let the TCP Layer at the source know how many packets the destination currently has room for
- watch for the packets
- piece them together as they arrive, possibly out of order (again, to reassemble them properly, we use the sequence numbers within the packets)
- send acknowledgement messages to the source node as packets arrive

The destination will also look at the error-checking bits in each package, and will give negative acknowledgements if errors are detected. If the source does not receive an acknowledgement for a given packet within a preset amount of time, it will **time out** and resend the packet. It will also do so if it receives a negative acknowledgement for a packet. For this reason, TCP is called a **reliable** protocol (though this term should not be taken to mean "100% reliable").

4.1.2 UDP

UDP, on the other hand, is **connectionless** and **unreliable**. UDP is pretty reliable if confined to a LAN, but problems may occur elsewhere, because for example a buffer at a router might be full and the **datagram** (the term used instead of *packet* in the UDP case) is dropped. UDP's virtue is that it is simple and thus has very little overhead, compared to TCP, which spends a lot of time negotiating and maintaining a connection between the source and destination nodes.

UDP is useful in applications in which we can afford to lose some messages, such as a time server, which broadcasts time of day to client machines; if a client misses one message this is no problem, as it will pick up the next one. Thus the unreliability of UDP is not a problem, and the low overhead of UDP is a virtue.

Similarly, suppose we are broadcasting a graphics animation or a movie over a network. Loss of one message would result in nothing more than a tiny "blip" on the screen, barely noticeable to the viewer, so again reliability is not an issue. Moreover, in **real-time** applications like this, we can hardly afford the delay caused by retransmitting when messages are lost or corrupted, which is what TCP would do.

Also, UDP is capable of **broadcasting**, i.e. sending out just one copy of a message to all machines connected to the same Ethernet. If we do wish to send the same message to everyone, the ability to do so using just one copy can really help reduce traffic on the network.

4.1.3 Stream Vs. Datagram Communication

It is extremely important to keep in mind that TCP views all the bytes it sends during one socket connection to consist of one long stream of bytes, with no subdivisions of any kind.

Suppose for example machine A executes

```
write(SDA,BufA1,20);
write(SDA,BufA2,30);
```

where **SDA** is a TCP **socket**. You will see the details of writing to a socket later, but for now suffice it to say that these two calls write 20 bytes from an array **BufA1** and 30 bytes from an array **BufA2** to the socket, and those bytes will be sent to machine B.

From TCP's point of view this is just a set of 50 bytes—not two sets of bytes, one of 20 and the other of 30. There will be no "fence", no "dotted line," etc. delineating some kind of boundary between the two sets of bytes. Moreover, on the receiver end at machine B, with a **read()** inside a **while** loop, we may, for instance, receive first 15 bytes, then 25, then 10. (However, the call to **read()** will not return until at least one byte is read, unless the socket is **nonblocking**, to be discussed later.) In that second chunk of 25 bytes, the first 5 of them would in this example be from the first call to **write()** on the sender end, with the other 5 from the second call, but again there will be no demarcation between these two sets of 5.

On the receiver end, we can keep reading until **read()** returns a nonpositive value; this indicates that the sender has closed the socket, either by calling **close()** or by simply exiting the program. (Just as exiting a program automatically closes all files, it also closes all sockets.) We would do this by, say, code like

```
char Buf[BUFSIZE],*Ptr;
...
Ptr = Buf;
do  {
   NBytesRead = read(SD,Ptr,BUFSIZE);
   TotMsgSize += NBytesRead;
   Ptr += NBytesRead;
} while (NBytesRead > 0);
```

So, even a single call to **write()** at the sender will need a loop containing **read()** on the receiver end, rather than just a single call to **read()**.[11]

By contrast, in UDP one call to write() at the sender does need to be matched by only one call to read() by the receiver. As opposed to TCP, in which the totality of bytes sent by the sender is just considered to be one long **stream**, the bytes sent by separate calls to **write()** are considered to be separate from each other. The set of bytes sent by a single call to write() is called a **datagram**.[12]

So, the application programmer must make a decision. If TCP is used, the programmer must add his/her own code to separate the various messages within the stream. Things would thus be easier under UDP,

[11] On the sending end, it is conceivable that even a call to **write()** may not send as many bytes as requested, if the OS kernel's write buffer is nearly full. However, the best reference on TCP/IP programming (*UNIX Network Programming*, by Richard Stevens, pub. Prentice-Hall, vol. 1, 2nd ed., p.77) says that this "is normally seen with **write()** only if the socket is nonblocking."

[12] Not to be confused with the same term for a packet at the IP level.

since there messages are received in the same sizes as they are sent. But on the other hand, with UDP the programmer would have to add his/her own code for "chunking," error checking, etc. (if needed).

Note that TCP has a slow startup time, due to the **handshaking** between the two nodes as the connection is established. Note also that there are limits to the length of a UDP datagram, typically around 8K or so, depending on the platform.

4.1.4 IP Addresses

Each node on the Internet has an **IP address**, a 32-bit number.[13] Usually this is written for human consumption as four numbers, specifically the values in each of the four bytes of the address. The IP address of the machine **garnacha.engr.ucdavis.edu**, for instance, is 169.237.126.236.[14]

When TCP or UDP passes a packet to IP, the latter will determine where it should be sent in order to ultimately reach its proper destination. For instance, if the machine has two NICs, IP must decide which one to pass the packet to.

4.1.5 Peer Communication

An application program using TCP or UDP will be communicating with its **peer** using TCP or UDP on the remote machine. Typically one of these programs will be a service provider and thus is termed the **server**, and the other will be the service requester, called the **client**. For instance, when you use the **ftp** program, it is a client, and the server is the **ftpd** program ("ftp daemon") running on the remote machine.[15]

For the application programmer, it would be extremely inconvenient to have to code the actual packet formation. Thus it would be nice to provide functions which access TCP and IP from a somewhat higher level. One popular type of such functions is **sockets**, which to the programmer look very similar to file handles. Again the sockets form peer relationships with each other. Consider **ftp** again, for instance, with a **put** operation. On your end, **ftp** writes data to its socket, and at the destination machine **ftpd** will read from its socket.[16] The two sockets were associated with each other at the time they were created. Similarly, the TCP layer on your machine will think of itself communicating with the TCP layer at the destination machine.

4.1.6 Viewing Current Socket Status

On UNIX machines, the **netstat** command will show you the current status of all open sockets; there is a version on Microsoft Windows machines too.

Next time you are on a UNIX machine, run the **netstat** command twice, once before and once during an ftp

[13] 128 bits in the new IP 6 system.

[14] This node no longer exists. The College of Engineering "Mexican food series" of machines was phased out some years ago, but I've kept the examples. Of course, the principles are still valid, regardless of which machines we use.

[15] The next time you are on a UNIX system, type "ps ax" (or "ps -e" or "ps -ux", depending on the system) to see all the currently-resident processes. You will may see one or more **ftpd** processes there. If not, do an ftp from another machine to this one, and list the processes again; you should now see **ftpd**. There is another daemon, **inetd**, which intercepts calls to services like **ftpd**, invoking these services as needed.

[16] In the case of a **get** opeation, these two operations would be reversed.

operation; you should see the new socket listed.[17]

4.1.7 What Makes a Connection Unique

Several socket programs might be running concurrently on the same machine. They may even all be accepting messages from the same remote machine. So, when a message arrives, how can the OS tell which program it should be routed to? The answer is that different socket programs are distinguished by their **port** numbers. The **ftpd** service is on port 21, for instance. A communication between two machines must be defined by five pieces of information:

- protocol (TCP, UDP, etc.)
- server IP address
- server socket port number
- client IP address
- client (ephemeral) socket port number

The server, for example, may be involved in several TCP transactions sent from the same machine, and thus the server needs to have some way of distinguishing between them. And similarly, when the server sends messages back to a client, the TCP system at the client's machine has to have some way of determining which program the server's message is intended for.

Say for example, there are two users currently on machine X, one using **ftp** to port 21 at machine Y, and another using **telnet** to port 23 at machine Z. But the programs being run by the two users will also have **ephemeral** ports at their own machines, say 2592 and 5009, temporarily assigned to them by the TCP system at machine X. The **ftp** program will inform the FTP server at machine Y about this 2592 number when it first connects to Y, and when the server sends back to X it will use this number. The TCP system at X will see this number, and route the message to the **ftp** program accordingly. Similar statements hold for **telnet**, etc. And even if the two users had both been using **ftp**, their two different ephemeral port numbers would distinguish them from each other.

4.2 Sample TCP/IP Application: NFS

As mentioned earlier, NFS allows machines to share files, in a manner transparent to the user. NFS uses the UDP transport protocol.[18]

When I ran the **df** command on the ACS machine **taco**, the output included a line

```
rosarita:/usr/pkg    2077470 1556279  313444    83%    /usr/pkg
```

[17] You may also some "UNIX" sockets, which don't involve networks.

[18] NFS actually uses Remote Procedure Call (RPC), which in turn uses UDP. RPC is just what the name implies: A program can actually call a function—with parameters—which will execute on another machine. We have earlier referred to UDP's "unreliable" nature, which means RPC must take its own reliability measures.

This says that the directory **/usr/pkg** on the machine **rosarita** has been mounted on **taco**, in a directory of the same name, **/usr/pkg** (the name need not have been the same). This directory contains a number of utilities, such as the GNU C compiler, **gcc**, in the file **/usr/pkg/gnu/bin/gcc**. Thus for instance the user can type

```
gcc x.c
```

as if **gcc** were on **taco**'s local disk.

How does this work? At boot time, **taco** is set up to execute shell scripts in files like **/etc/rc**, **/etc/rc.local** and so on. In one of these files, there will be a command like

```
mount -t nfs rosarita.engr.ucdavis.edu:/usr/pkg /usr/pkg
```

On the other end, one of **rosarita**'s bootup files will include an **export** command, saying that it is all right to allow its directory **/usr/pkg** to be mounted by other machines. The OS is set up so that if on **taco** I try to access a file in **/usr/pkg**, my request will actually be redirected to **rosarita**, transparently to me.

Here is why the RPC mechanism is convenient. On a UNIX system, input/output is done via system calls **open()**, **read()**, **write()** and so on.[19] So, in the example above, when a file within **/usr/pkg** is referenced at **taco** (our use of **gcc** will consist of a read to the file **/usr/pkg/gnu/bin/gcc**), the function **open()**, **read()**, and so on will then do RPCs to the corresponding functions at **rosarita**.

5 Network Programming

5.1 TCP Socket Example

Below are two C programs, a client and a server. To explain what they do, suppose the server is running on machine X and the client on Y. The server will report to the client the load at X (defined by the output of the UNIX commands **w** and **ps -ax**). Suppose for example we compiled the client and server under the names **wps** and **svr** on **toto.berkeley.edu** and **garnacha.engr.ucdavis.edu**, respectively. We would run **svr** on the latter (probably as a background process), and on the former might test **wps** as follows:

```
toto% wps garnacha.engr.ucdavis.edu w
  1:01pm  up 37 days, 17:54,  2 users,  load average: 0.06, 0.00, 0.00
User     tty       login@  idle   JCPU   PCPU  what
matloff  ttyp1    12:58pm     1      4      1  script
matloff  ttyp2     1:00pm     1      2      1  w
bslouie  ttyp3    11:43am     1     42     13  tin-new
toto%
```

(Note that my name appeared in the output. This is just a coincidence, arising from the fact that I had telnet-ed into **garnacha** from **toto** in order to start up **svr**.)

[19] You might not have used these before, and instead be more familiar with **fopen()**, **fscanf()**, **fprint()**, and so on.

5.1.1 Source Code

Here are the programs:

```
 1
 2  /* WPsClient.c */
 3
 4  /*  Client for the server for remote versions of the
 5      w and ps commands.
 6
 7      User can check load on machine without logging in
 8      (or even without being able to log in).
 9
10      Usage:  wps remotehostname command (where "command"
11      is either w or ps).
12  */
13
14
15  /* these are needed for socket calls */
16  #include <stdio.h>
17  #include <sys/types.h>
18  #include <sys/socket.h>
19  #include <netinet/in.h>
20  #include <netdb.h>
21
22
23  #define WPSPORT 2000  /* server port number */
24  #define BUFSIZE 1000
25
26
27  main(argc,argv)
28     int argc;  char **argv;
29
30  {  int SD,MsgSize;
31     struct sockaddr_in Addr;
32     struct hostent *HostPtr,*gethostbyname();
33     char Buf[BUFSIZE];
34
35     /* open a socket */
36     SD = socket(AF_INET,SOCK_STREAM,0);
40
41     /* set up for an Internet connection to the host whose
42        name was specified by the user on the command line */
43     Addr.sin_family = AF_INET;
44     Addr.sin_port = WPSPORT;
45     /* get IP address of host */
46     HostPtr = gethostbyname(argv[1]);
47     memcpy(&Addr.sin_addr.s_addr,
48        HostPtr->h_addr_list[0],HostPtr->h_length);
49
50     /* OK, now connect */
51     connect(SD,&Addr,sizeof(Addr));
55
56     /* send user command */
57     write(SD,argv[2],strlen(argv[2]));
58
59     /* display response */
60     MsgSize = read(SD,Buf,BUFSIZE);
```

```
61     write(1,Buf,MsgSize);
62  }
63
```

```
 1  /* WPsServer.c */
 2
 3  /*  A server for remote versions of the w and ps
 4      commands.
 5
 6      User can check load on machine without logging
 7      in (or even without being able to log in).
 8  */
 9
10
11  /* these are needed for socket calls */
12  #include <stdio.h>
13  #include <sys/types.h>
14  #include <sys/socket.h>
15  #include <netinet/in.h>
16  #include <netdb.h>
17
18
19  /* this is needed for the disk read */
20  #include <fcntl.h>
21
22
23  #define WPSPORT 2000  /* server port number */
24  #define BUFSIZE 1000
25
26
27  int ClntDescriptor,  /* socket descriptor to client */
28      SrvrDescriptor;  /* socket descriptor for server */
29
30
31  char InBuf[BUFSIZE],  /* messages from client */
32       OutBuf[BUFSIZE];  /* messages to client */
33
34
35  Write()
36
37  {  int FD,NB;
38
39     FD = open("tmp.client",O_RDONLY);
40     NB = read(FD,OutBuf,BUFSIZE);
41     write(ClntDescriptor,OutBuf,NB);
42     unlink("tmp.client");
43  }
44
45
46  Respond()
47
48  {  memset(OutBuf,0,sizeof(OutBuf));  /* clear buffer */
49     if (!strcmp(InBuf,"w"))
50        system("w > tmp.client");
51     else if (!strcmp(InBuf,"ps"))
52        system("ps -ax > tmp.client");
53     else
54        system("echo 'invalid command' > tmp.client");
```

```
 55     Write();
 56  }
 57
 58
 59  main(argc,argv)
 60     int argc;  char **argv;
 61
 63  {  struct sockaddr_in BindInfo;
 66
 67     /* create an Internet TCP socket */
 68     SrvrDescriptor = socket(AF_INET,SOCK_STREAM,0);
 73
 74     /* bind it to port 2000 (> 1023, to avoid the
 75        "well-known ports"), allowing connections from
 76        any NIC */
 77     BindInfo.sin_family = AF_INET;
 78     BindInfo.sin_port = WPSPORT;
 79     BindInfo.sin_addr.s_addr = INADDR_ANY;
 80     bind(SrvrDescriptor,&BindInfo,sizeof(BindInfo));
 86
 87     /* OK, set queue length for client calls */
 88     listen(SrvrDescriptor,5);
 91
 93     while (1)  {
 94        /* wait for a call */
 95        ClntDescriptor = accept(SrvrDescriptor,0,0);
106        /* read client command */
107        memset(InBuf,0,sizeof(InBuf));
108        read(ClntDescriptor,InBuf,sizeof(InBuf));
113        /* process the command */
114        Respond();
115     }
116  }
117
```

Note the #include files. Also, here are some other notes on programming:

- On Solaris machines, you may need to add **-lsocket** and possibly **-lxnet** to your compile command.
- On some platforms, you will have to be more careful with C casts than I have in the code above, for example in the second argument to **connect()**. This is especially true for C++.
- Though in the code here we have not done error-checking, in order to avoid distraction from the main concepts, you definitely should check the return values of the system calls, so that if one fails you will know which one it was.
- Even after a socket is closed, TCP will keep that socket alive for a while (e.g. 30 seconds or so), just in case there are still some packets to come in. Thus a given port will not be immediately reusable, unless we call **setsockopt()** with SO_REUSEADDR after opening the socket on the server end.[20]

Let's see how the programs work, taking the client first.

Line 23:

[20]This should be used with care, though, since the streams from two clients might get mixed.

We have chosen to put our server on port 2000. Ports 0-1023 are reserved for the so-called **well-known ports** corresponding to standard TCP/IP services. You can see a list of well-known ports in the UNIX file **/etc/services**.

Line 24: We will use arrays for our **read()** and **write()** system calls. They need to be large enough for the data we send. In this case, 1000 characters will be sufficient. Note carefully, though, that this limit is NOT there for the purpose of keeping our packets small; we can send as long a message as we like, with a single call to **write()**. Remember, TCP will break up long messages into shorter ones for us, without us even seeing it, so we do not have to worry about it.

Lines 36-39:

We open a socket, using the **socket()** system call. The first parameter indicates whether this is an Internet socket, versus a socket purely local to this machine (a "UNIX socket"), not going out onto the Internet; the Internet case here is designated by AF_INET. The second parameter indicates the service, i.e. TCP, UDP or others; it is TCP in this case, designated by SOCK_STREAM.[21] We have defaulted the third parameter here (and will not worry about what other possibilities there are for it).

The function's return value, assigned to **SD** here, is a **socket descriptor**, quite analogous to a file descriptor.

Lines 41-54:

We are building up a data structure named Addr ("address") which we will use in line 51. Its type, **sock-addr_in** (line 31) comes from the #include file and is a standard socket type. Clearly, it is a complex type, with many fields, and we will not go into the details here. See the man pages if you are interested for more information.

One thing to beware of is that for the system call connect() and many other socket functions, you will see that the man pages say you should use a struct of type **sockaddr**, as opposed to the **sockaddr_in** type we have used here. Yet the **sockaddr** type is just meant as a dummy, to be replaced by another struct type which is specific to the network protocol being used. In our case, we are using TCP/IP, so we choose the **sockaddr_in** type, where "in" stands for "Internet." There are also types such as **sockaddr_ns** for the Xerox Networks Systems protocol, though of course TCP/IP has become virtually ubiquitous.

Having already opened a socket, we have to connect it to a specific port at a specific machine. Recall that the user specifies an Internet host name, such as **garnacha.engr.edu**. But we need the machine's numerical Internet address. This is obtained on line 46. The return value is a pointer to another standard data structure.

In line 47, we call **memcopy()**, a system call which copies strings from one part of memory to another, in this case from various fields of the struct pointed to by **HostPtr** to **Addr**. A given host may have several addresses; here we are not bothering to check for that, but simply using the first address, contained in **h_addr_list[0]**. (Here 'h' stands for "host.") The **h_length** field gives the length of the address.

In line 51, we now connect the socket to the destination host. (It is here that the negotiation between source and destination hosts will occur, as to packet sizes, and so on.)

Note that the port number we have specified is for the port at the server, not the client. There is a hidden port number for the client here, which we will discuss later.

Line 57:

[21] There are also various others, such as SOCK_RAW for raw sockets.

Here we write either “w” or “ps” to the destination host, depending on what the user requested. Note that the function **write()** is identical to the one used for low-level file access, except that we have as the first parameter a socket descriptor instead of a file descriptor.

Lines 60-61:

On line 60 we read the message sent by the destination host, which will be the output of that host’s running either the **w** or **ps** command. On line 61 we then write that message to the user’s screen. (For convenience, we do so again using **write()**, making use of the fact that the standard output has file descriptor 1, though of course could have used **printf**().)

Note carefully that if we had been expecting more voluminous data from the server than is the case here, we may have had to do repeated calls to **read()**.

In addition to **read**() and **write**(), we may also access sockets via other very similar system calls **recv()**, **send()**, **recvfrom()** and sendto(). (And in non-UNIX environments, we *must* use these, since those OSs do not treat socket and file I/O the same.)

Now let’s take a look at the server code.

Line 68:

Here the server creates a socket.

Lines 77-85:

The system function **bind()** is called in line 80. This associates the socket with a particular port and with a particular IP address on the local machine, i.e. the machine on which the program which calls **bind()** is running, in this case the server machine. Recall that machine may have multiple IP addresses, either because it has several NICs, or because it may have multiple IP addresses assigned to the same NIC.

For example, suppose one of the NICs at the server machine corresponds to a local private intranet. Then we could specify that particular address in our call to **bind()**, which would enable our making the program available only on this intranet.

Or, suppose we are running an Internet service provider (ISP). We may be hosting many different customer Web sites, all with different names (**www.acmegroceries.com**, **www.flatearthsociety.org**, etc.), each with a different IP address. We want a given server, say the one for Acme Groceries, to respond only to clients accessing the **www.acmegroceries.com**, so our call to **bind()** would specify that address.

In our case here, we wish this server to be accessible from via of its IP addresses, which we specify in line 79 by using the constant INADDR_ANY.

Note that we did not have a call to **bind()** in our client code. We could have had one if we wanted the client to access this port only through a particular one of the IP addresses of the client machine. In the intranet example above, for instance, if the information to be exchanged by the client and server really needs full security, it might be safer to make sure the client does not accidentally send its information outside the intranet (say because a routing table becomes corrupted).[22]

To summarize, in a server program, calling **bind()** associates with the given socket the port number and IP address that “phone calls” to this socket will be allowed on.

[22] If we do call **bind()** in a client, this must be done before calling **connect()**.

Normally we do not need to call **bind()** in a client program. Yet the client still needs a port number and IP address to use in accepting messages which come from the server. If we do not call **bind()** in the client, then calling **connect()** in the client will cause the OS to assign to the client a port, called an **ephemeral port**, as well as an IP address (in the case that the client has more than one IP address).[23]

Note that nowhere in the client or server code above do we see any mention of the client's ephemeral port number. (WPSPORT is the number of a port at the server, not at the client.) But the OS at the client will indeed notify the server regarding the identity of the ephemeral port (when the client calls **connect**()),

Normally we do not need to know which ephemeral port has been assigned at a client, but if we need it then we can get it by calling **getsockname()** in the client after calling **connect()**.

Line 88:

Be calling the **listen()** function, we are notifying the OS that this program will be a server, not a client. We also notify the OS as to how many incoming calls (in the form of clients invoking the **connect()** function) will be allowed to be pending at one time, in this case five. If a call arrives when the queue is full, the call will be discarded (so it is best when writing the client to put the call to **connect()** in a loop, looping until **connect()** succeeds).

Lines 93-115:

Here we loop indefinitely, continuing to process calls one at a time. The **accept()** function (line 95) accepts a pending call, returning a socket which we will use to exchange messages to the client. By making it a separate socket, we can have several client sessions active simultaneously, though we are not doing so here. The original socket is then called a **listening** socket, whose job is only to listen for connection requests from clients, rather than for actual information exchange with client. The sockets created by **accept()**, which do the actual message exchange with the clients, are called **connected sockets**.

We read the client's command, "w" or "ps", on line 108, and then respond to the client on line 114.

Lines 35-56:

There are two systems calls you might not be familiar with here. The function **system()** (lines 50, 52 and 54) actually submits a shell-level command. Note that we are saving the response in a file, **tmp.client**. The file is later removed, using the **unlink()** function (line 42), which is the system-call level analog of the shell-level **rm** command.

5.1.2 Who Shall I Say Is Calling?

There are many other TCP/IP functions available. For example, after line 95 in the server code, we could call **getpeername()** if we needed to know the Internet address of the client. To use this function, declare a variable, say **s**, of type **sockaddr_in** and initialize its **sin_family** field to AF_INET. Also, declare a variable, say **i**, of type **int** and initialize it to **sizeof(sockaddr_in)**. Then the call to **getpeername()** will have as its arguments to the socket descriptor (**ClntDescriptor** in our example here), **&s** and **&i**. To then get the address as a character string in "Internet dot" form, call the function **inet_ntoa()** on **s.sin_addr**.

That would give us the numerical Internet address, and if we needed the alphabetic name, we could get this

[23]This is for TCP. In the case of UDP, where one normally does not call **connect**(), this function is performed by the call to **sendto**().

by calling **gethostbyaddr()**, as follows. Define a variable, say hp, of type "struct hostent *". Then call **gethostbyaddr()** with the arguments &s.sin_addr, 4 and "AF_INET", assigning the return value to hp. The alphabetic host name will then be available as a character string in `hp->h_name`.

5.2 UDP Socket Examples

5.2.1 Basic Example

Following are a pair of programs which communicate using UDP sockets:

```
1  /* BasicCln.c */
2
3  /* introductory UDP example (client), with client sending a one-line
4     message to server at port 4000 */
5
6  #include <sys/types.h>
7  #include <sys/time.h>
8  #include <sys/socket.h>
9  #include <netinet/in.h>
10 #include <netdb.h>
11
12 main()
13
14 {  struct hostent *h_name;
15    int sockfd;
16    char buf[10];
17    struct sockaddr_in your;
18
19    your.sin_family = AF_INET;
20    your.sin_port = htons(4000);
21    h_name = gethostbyname("sgi8.cs.ucdavis.edu");
22    your.sin_addr.s_addr = *(u_long *) h_name->h_addr_list[0];
23
24    if ((sockfd = socket(AF_INET,SOCK_DGRAM,0)) < 0)  {
25       printf("socket error\n");
26       exit(1);
27    }
28
29    strcpy(buf,"OK");
30
31    if (sendto(sockfd,buf,strlen(buf),0,&your,sizeof(your)) < strlen(buf)) {
32       printf("send error\n");
33       exit(1);
34    }
35 }
```

```
1  /* BasicSrv.c */
2
3  /* introductory UDP example (server), with client sending a one-line
4     message to server at port 4000 */
5
6  #include <sys/types.h>
7  #include <sys/time.h>
8  #include <sys/socket.h>
```

```
 9  #include <netinet/in.h>
10  #include <netdb.h>
11
12  main()
13
14  {  struct hostent *h_name;
15     int sockfd;
16     char buf[10];
17     struct sockaddr_in mine;
18     /* these two are needed as placeholders but are not used here: */
19     struct sockaddr_in rcvaddr;
20     int addlen;
21
22     mine.sin_family = AF_INET;
23     mine.sin_port = htons(4000);
24     mine.sin_addr.s_addr = INADDR_ANY;
25
26     if ((sockfd = socket(AF_INET,SOCK_DGRAM,0)) < 0)  {
27        printf("socket error\n");
28        exit(1);
29     }
30
31     if (bind(sockfd,(struct sockaddr *)&mine,sizeof(mine)) < 0)  {
32        close(sockfd);
33        printf("bind error\n");
34        exit(1);
35     }
36
37     recvfrom(sockfd,buf,sizeof(buf),0,&rcvaddr,&addlen);
38     printf("%s\n",buf);
39  }
```

We have used the **htons()** function error, in order to guard against problems in communicating between big- and little-endian machines. The Internet uses big-endian order, while for example Intel-based machines use little-endian order. To be safe, this should always be used.

Note that the **recvfrom()** function has an argument in which the system will place the client's address. If the server needs to reply (which it doesn't in this case but does in most applications), it does a call to **sendto()**, using the same socket and the client address which it discovered in its call to **recvfrom()**.

5.2.2 Advanced Use of Sockets

There are myriad options available for sockets, which can be set by calling the function **setsockopt()**. Here is an example:

One nice feature of UDP is that if it is used on a single network on which broadcast is physically possible, UDP can arrange us to *simultaneously* send a packet to every node on that network. A typical example of this is that of an Ethernet; a packet going to one node on the network will be "seen" by all other nodes on the network, so a physical broadcast is possible, and UDP has the capability of exploiting this.[24]

You can determine the broadcast address (given in IP address form) by running the **ifconfig** command on UNIX machines, or **winipcfg** on Microsoft Windows platforms.

[24] The College of Engineering ACS machines apparently have the broadcast capability turned off, but it works on CSIF.

Here is how we could change the client in the above example to do a broadcast (using the same server, but now running on many nodes on the Ethernet):

```
1  /* BCastCln.c */
2
3  /* introductory UDP example (client), with client sending a one-line
4     message to all servers on the given network (see below), at port 4000
5
6     the server is still BasicSrv.c, no change */
7
8  #include <sys/types.h>
9  #include <sys/time.h>
10 #include <sys/socket.h>
11 #include <netinet/in.h>
12 #include <netdb.h>
13
14 main()
15
16 {  struct hostent *h_name;
17    int sockfd;
18    char buf[10];
19    struct sockaddr_in your;
20
21    const int turn_option_on = 1;
22
23    your.sin_family = AF_INET;
24    your.sin_port = htons(4000);
25    /* UCD CS and Engrg. machines are on the network 128.120.*.*,
26       i.e. hex IP addresses 0x8078zzzz, so the broadcast address
27       replaces zzzz by ffff (after this program was written,
28       the addresses became 169.237.*.*; in general, run
29       /sbin/ifconfig to determine broadcast address */
30    your.sin_addr.s_addr = 0x8078ffff;
31
32    if ((sockfd = socket(AF_INET,SOCK_DGRAM,0)) < 0)  {
33       printf("socket error\n");
34       exit(1);
35    }
36
37    setsockopt(sockfd,SOL_SOCKET,SO_BROADCAST,&turn_option_on,
38       sizeof(turn_option_on));
39
40    strcpy(buf,"OK");
41
42    if (sendto(sockfd,buf,strlen(buf),0,&your,sizeof(your)) < strlen(buf)) {
43       printf("send error\n");
44       exit(1);
45    }
46 }
```

Another example of the advanced use of sockets is **raw** sockets. Here one can build up one's own IP frame, giving one very minute control of IP operations by setting the various fields ourselves. A program which uses raw sockets must have root privileges.

5.3 Nonblocking I/O

In many applications a server has sockets open to several clients at once. In this case, the server needs a mechanism for determining which sockets have data waiting and which do not. One way to handle this is to make the sockets **nonblocking**, which means that a call to **read()** will not wait until data is ready. If data is ready at that socket, the call to **read()** will read that data, but if not, the call immediately returns, with a value of -1. You code can then repeatedly poll all sockets, testing for input data at each one, and reading that data if it is there. Here is an example of code to make a socket nonblocking:

```
Flag = 1;
ioctl(S,FIONBIO,&Flag);
```

Here **S** was the return value from a call to **socket()**. You will need the proper include-files; check the **man** page for **ioctl()**.

A much more flexible and sophisticated tool for dealing with multiple sockets is the **select()** function. A newer such tool is **poll()**.

5.4 Debugging Client/Server Programs

As a quick check, you can first try to use **telnet** to check whether the server has called **bind()**, **listen()** and **accept()** properly. Though you are probably accustomed to using **telnet** simply as a means of remote login, it also can be used to communicate with servers at specified ports. What **telnet** does is open a connection to a given host at a given port; whatever bytes the user types will be sent to that port, and whatever bytes the port sends will appear on the user's screen.

This could be used to help our debugging process. If for example we have a server on **pc8.cs.ucdavis.edu**, running on port 1088, we could type

```
telnet pc8.cs.ucdavis.edu 1088
```

If we get a response here but not from our own client program, the latter may have an error in **connect()** or whatever, such as misspecifying the server's IP address or port. (On the other hand, if we get no response, it may also be due to the system configuration not allowing **telnet** access to that port.)

In general, debugging a server/client pair, using a debugging tool **(which you should do when debugging any program)** will be a bit more difficult, because you will need to invoke the tool once for the server and once for the client. So, even though I typically use a GUI to **gdb**, such as **ddd**, for debugging a server/client pair I sometimes use just the plain-text **gdb**, since my screen would not conveniently fit two GUIs at the same time. Or, I might just use the debugging tool on the client while running the server without a debugging tool, or vice versa.

You may find a tool such as **strace**, available on many UNIX systems (and also similar programs such as **ktrace**, **truss**, etc.) to be useful. It will print out each system call made by a program, and the result of each call. In our case here, that means calls to **accept()**, **connect()**, etc. Since the ouptut of **strace** may be voluminous, you may wish to pipe its **stderr** output through **more**, say as

```
strace application_program_name application_arguments |& more
```

6 Packet/Frame Formats

In order to get a more concrete understanding of some of the concepts introduced here, we now take a look at the specific formats in which some of the protocols send data.

6.1 TCP

Bytes 0-1: Source Port.

Bytes 2-3: Destination Port.

Bytes 4-7: Sequence number.

Bytes 8-11: Acknowledgement.

Byte 12: The first four bits comprise the Header Length field (the other four bits are 0s), meaning the number of words (not bytes) from the beginning of the packet to the first data byte within the packet. Most of the header is of fixed length, so we would not need this, except for the fact that the Options field below is of variable length. The software thus uses this field to deduce where in the packet the data starts.

Byte 13: Flags, which are various bits giving control information such as a PUSH command (which tells the host not to continue accumulating bytes to send; "send whatever you have now, without waiting for more").

Bytes 14-15: Advertised Window, a value that the recipient uses to say, "OK, you can now send me sequence numbers such-and-such."

Bytes 16-19: Check Sum (two bytes), for error checking, and a two-byte Urgent Pointer field.

Bytes 20-whatever: Options field.

Remaining Bytes: Data, e.g. your e-mail message in the case of **sendmail**. (Note: No length field is needed for specifying the amount of data, since this can be deduced from a similar field in the IP header which will contain this TCP packet.)

6.2 IP

Byte 0: The first four bits are the Version Number (currently 4, going to 6), and the other four bits are the Header Length in words.

Byte 1: Type of Service field, intended to give priority to some packets but not used much in practice.

Bytes 2-3: Length field, giving length of the entire IP packet including data.

Bytes 4-7: Miscellaneous fields.

Byte 8: Time to Live field. If this equals, say, k, then this packet will be allowed k more hops through the network. If it hasn't reached its destination by then, it is discarded, to prevent infinite routing loops.

Byte 9: Transport-layer protocol (e.g. 6 for TCP, 17 for UDP).

Bytes 10-11: Checksum, to check for errors within this packet.

Bytes 12-15: Source IP address.

Bytes 16-19: Destination IP address.

Bytes 20-whatever: Options including blank padding to make an integral number of words.

Remaining Bytes: Data. Remember, from the point of view of the IP layer, the "data" consists of a TCP or UDP packet (or other packet from a higher layer).

6.3 Ethernet

Preamble: Start-of-frame indicator, a special 64-bit pattern.

Destination Ethernet ID: 48 bits; burned into the NIC by the manufacturer.

Source Ethernet ID: See Destination Ethernet ID above.

Type (protocol ID): 16 bits. Indicates the protocol being used, e.g. 0x0800 for IP and 0x809b for Appletalk. This is the mechanism by which different protocols can coexist on the same Ethernet.

Data: This consists of the IP packet (in the case of the IP protocol). Its length is inferred by subtracting the lengths of the other fields from the overall frame length.

CRC: 32 bits. An error-checking field for this frame.

Postamble: End-of-frame indicator, a special 8-bit pattern.

7 Putting It All Together

Suppose that in our earlier examples, **svr** is running on **venus**, and **wps** is running on **honda**. Suppose we have the following IP and Ethernet (MAC) addresses:

machine	IP address	MAC address
Earth	192.0.0.0	0x0123456789ab
Mars	192.0.0.1	0x1123456789ab
Venus	192.0.0.2	0x2123456789ab
Saturn	192.0.0.3	0x3123456789ab
Jeep	193.0.0.0	0x4123456789ab
Honda	193.0.0.1	0x5123456789ab

All of the machines here have **Class C** IP addresses, which consist of a 24-bit network number and 8-bit host-within-network number. For example, **mars** is host 1 on network 192.0.0.0. Here is what will happen when **wps** executes

```
write(SD,argv[2],strlen(argv[2]));
```

at **honda**. Recall that **argv[2]** is either "w" or "ps"; let's say it's "ps". Recall also that **SD** is a socket which **wps** has already opened in TCP. Also, the call which **wps** made earlier to **connect()** had connected this socket to port 2000 at **venus**, and the OS at **honda** had assigned **wps** an ephemeral port number, say 3056.

So, the effect of the **write()** is that the socket number **SD** and the string **argv[2]** will be sent from **wps**, which is running in the Application Layer at **honda**, to the Session Layer at **honda**.

The Session Layer will find in its records that this socket is for ephemeral port 3056 on TCP. So, the Session Layer at **honda** will pass "ps" and this port number to TCP in the Transport Layer at **honda**. (Note that this passing is done by a simple function call, since we are at the same machine.)

TCP at **honda** will first have to decide how much "chunking" to do—none in this case, since the data consist of only two bytes! TCP will now prepare a TCP packet containing that data, using the TCP packet format shown above:

TCP will first note that ephemeral port 3056 is associated with the destination port and IP numbers 2000 and 192.0.0.2, respectively. TCP will then fill in the packet, putting 3056 and 2000 for the Source and Destination Port numbers; it will fill in the Sequence number, etc., and finally put "ps" into the Remaining Bytes (i.e. data) field. After creating this packet, TCP at **honda** will pass it to IP in the Network Layer of **honda**, along with the information that the destination will be 192.0.0.2.

IP at **honda** will now prepare an IP packet. It will fill in 193.0.0.1 for the Source IP address, and 192.0.0.2 for the Destination IP. Note carefully that for the Remaining Bytes field in this case, IP will put in the entire packet that it received from TCP.

IP at **honda** will now decide how to route the IP packet it has created. It first will ask whether the destination host, is on the same network as **honda**. The answer to that question will be no, since **honda** is on the network 193.0.0 and **venus** is on 192.0.0. So, IP at **honda** will need to send the packet to a router on **honda**'s network. There are two such routers, **jeep** and **citroen**.

IP at **honda** will send the packet to **jeep**. (This will probably have been hand-coded; more on this in our unit on routing.) Note that IP at **honda** will not know that **venus** is just one hop away from **jeep**; it merely knows that the first step should be **jeep**. So, IP will pass the packet, plus **jeep**'s Ethernet address, 0x4123456789ab, to the Data Link Layer at **honda**.

The Data Link Layer will then create an Ethernet frame. It will put 0x4123456789ab for the Destination Ethernet ID, and 0x5123456789ab for the Source Ethernet ID. It will fill in 0x0800 for the Type field. And it will put in the entire IP packet it received from the Network Layer in for the current Data field. Finally, the Data Link Layer will pass this Ethernet frame to the Ethernet device driver on **honda**.

The Ethernet device driver on **honda** will then put the frame on network B. All NICs on that network will see it, including the NIC 0x4123456789ab on **jeep**.

That NIC will say, "Oh, this frame is for me!" Note that the NIC will not notice that the ultimate destination of the frame is **venus**; all the NIC cares about is the Destination Ethernet address, which it has seen is its own. All the NIC will do is pass this frame up to the next layer at **jeep**,[25] which will be the Data Link Layer.

The Data Link Layer at **jeep** will strip off the Ethernet IDs and other Ethernet-related information. The stripped-down frame is now the IP frame which IP at **honda** had produced. The Data Link Layer will know this, since the Type field in the Ethernet frame stated that the protocol was IP. The Data Link Layer at **jeep** will now pass the IP packet to IP in the Network Layer at **jeep**.

IP will now look at the Destination IP Address in the packet, 192.0.0.2. Since that does not match **jeep**'s

[25]Note that it is the same protocol stack as that of **mars**. They are the same machine, but have two different NICs and thus different names.

own address, 193.0.0.0, **jeep** knows that it needs to route this packet. IP also notices that the Destination IP Address is on network number 192.0.0, i.e. network A, which **jeep**'s machine is attached to via another NIC. So, **jeep** will be able to relay the packet directly to **venus**:

At this point the packet will go down the protocol stack at **jeep**, just like we saw earlier at **honda**. The Ethernet Source ID will be **jeep**'s, i.e. 0x4123456789ab, and the Ethernet Destination ID will be **venus**', 0x2123456789ab. The frame will be placed onto network A, and picked up by **venus**.

The frame will then go up the protocol stack at **venus** like it did at **jeep**, but in this case IP at **venus** will discover that the Destination IP Address is that of **venus**. Thus the packet will not be routed from **venus**, but instead will continue to go up the protocol stack. IP at **venus** will see in the Transport Layer Protocol field that this is a TCP packet (and thus not, for example, UDP). IP will now strip off the IP-only fields from the packet, leaving the original TCP packet. IP will pass the packet to TCP, along with information on the Source IP address.

TCP will note the Destination Port, strip off the TCP-only information, and send the remainder to the Session Layer, along with the Destination Port number. The Session Layer will send the remainder to **read()**, and **svr** will be able to read the "ps".

When **svr** writes back to **wps**, a similar sequence of events will occur. Note also that when **wps** first called **connect()**, a similar sequence of events occurred then too, as **honda**'s TCP and **venus**' TCP exchanged messages in order to set up a connection.

8 Application-Layer Protocols

Some applications, such as FTP, e-mail, etc. are so common that they have their own protocols.

For example, e-mail uses the Simple Mail Transfer Protocol (SMTP). Suppose you are **lm@abc.com** and are sending e-mail to **uvw@xyz.org**, and your message is going to be a simple one-line greeting:

```
Hi, how have you been?
```

The client, which will be either the e-mail utility that you use, or an OS function called by that utility, will first establish a TCP connection to the SMTP server at the remote machine, at port 25 of the server. The following messages would then be sent by the client:[26]

```
1 HELO abc.com
2 MAIL FROM: lm@abc.com
3 RCPT TO: uvw@xyz.org
4 DATA
5 Hi, how have you been?
6 .
7 QUIT
```

(The end of the e-mail message itself is indicated by a period on a separate line, as shown above.)

[26]There will be responses from the server for each one, but we will ignore them here.

Similarly, HTTP, the protocol used for Web access, consists of a set of commands similar to the HELO, MAIL FROM:, RCPT TO:, DATA and QUIT commands we saw for SMTP above. The HTTP server is at port 80.

As our example, consider the Web page

```
http://heather.cs.ucdavis.edu/~matloff/gnuplot.html
```

If a user instructs his/her Web browser to access that Web URL, the browser would open the usual TCP connection with **heather.cs.ucdavis.edu**, at the standard HTTP port, 80. The browser would then write the bytes

```
GET /~matloff/gnuplot.html HTTP/1.0 \n\n
```

to port 80. That is the HTTP command to get the given file on that machine. (Note the two blank lines, which actually are part of the command.) The machine would respond by sending back the raw HTML file, which happens to consist of the HTML code

```
<body bgcolor=white>
<P>
<P>
The gnuplot mathematical graphing package is available on most Unix
systems.  It is free, public domain software.
<P>
Here is some documentation:
<P>
<UL>
<LI>
Gnuplot's <A HREF="Gnuplot/Doc.html">official documentation</A> (note
also that online help is available by typing "help" within gnuplot).
<P>
<LI>
My own <A HREF="Gnuplot/NotesGnuplot.NM.html">brief introduction.</A>
</UL>
<P>
```

The browser would then interpret that HTML code and display the Web page.

9 Routing Issues

Again, many thick books exist on routing issues (some of them dealing only with a particular commercial product, such as Cisco routing). But here is a brief overview:

Suppose I tell my Web browser to go to **http://www.yahoo.com** One of the things which will have to be done is to get the IP address for **www.yahoo.com**. Recall that the function **gethostbyname()** does this—but how?

The function first may check the local file **/etc/hosts**, where some frequently accessed destination information might be stored. If not, then it probably will use the Domain Name System (DNS). (Some Microsoft

Windows networks use Microsoft's own name lookup system, WINS.) Briefly, your OS will have one or more DNS machines which it can query to convert the "English" name, say **www.yahoo.com** to an IP number, say 204.71.200.74.[27] The addresses of these machines are typically stored in a file, **/etc/resolv.conf**, in lines labeled "nameserver," or might be specified in other ways.

If, for example, you have just subscribed to a new ISP but can't access all (or some, say UCD) machines on the Internet, try doing so using their numerical IP address. If the latter works, then you likely have some kind of DNS problem.

Once we determine the IP address of the destination, the next question is how to get there. The IP software will first check to see if we are lucky enough that the destination machine is on the same network (e.g. the same Ethernet) as we are on. The way that it does this is to break the destination IP address down into a network number and a host-within-network number. (You'll see how to do this later.) If it discovers that the destination is on the same network as the source, it simply puts the Ethernet address of the destination into our packet and sends it out onto the Ethernet. If not, our IP software will have to send the packet to some machine on our network that serves as a **gateway** to the rest of the Internet. This process will be repeated each time we reach a new network, until we finally reach the network to which our destination machine belongs.

In TCP, each packet within a message may take a different route to the destination. Other protocols may handle things differently. The Asynchronous Transfer Mode (ATM) protocol, for example, sets up a fixed route at the time the connection is made, for all packets to use.

[27]They in turn may have to check other DNS hosts to get the information you want.