–213 Handout 9: Network Programming

Article · February 2001

Source: CiteSeer

CITATIONS
0

READS
5,306

2 authors, including:

Randal E. Bryant
Carnegie Mellon University
265 PUBLICATIONS **25,408** CITATIONS

SEE PROFILE

15-213 Handout #9: Network Programming*

Randal E. Bryant
David R. O'Hallaron

January 17, 2001

1 Introduction

Network applications are everywhere. Any time you browse the Web, send an email message, or pop up an X window, you are using a network application. Interestingly, all network applications are based on the same basic programming model, have similar overall logical structures, and rely on the same programming interface.

Network applications rely on many of the concepts that we have already learned in our study of systems. For example, processes, signals, threads, reentrancy, byte ordering, memory mapping, and dynamic storage allocation all play important roles. There are new concepts to master as well. We will need to understand the basic client-server programming model and how to write client-server programs that use the services provided by the Internet. Since Linux models network devices as files, we will also need a deeper understanding of Linux file I/O. At the end, we will tie all of these ideas together by developing a small but functional Web server that can serve both static and dynamic content with text and graphics to real Web browsers.

2 Client-server programming model

Every network application is based on the *client-server model*. With this model, an application consists of a *server* process and one or more *client* processes. A server manages some *resource*, and it provides some *service* for its clients by manipulating that resource.

For example, a Web server manages a set of disk files that it retrieves for clients. An FTP server manages a set of disk files that it stores and retrieves for clients. An X server manages a bit-mapped display, which is paints for clients, and a keyboard and mouse, which it reads for clients. The X server is interesting because it is always close to the user while the client can be far away. Thus proximity plays no role in the definitions of clients and servers, even though we often think of servers as being remote and clients being local.

The fundamental operation in the client-server model is the *transaction* depicted in Figure 1.

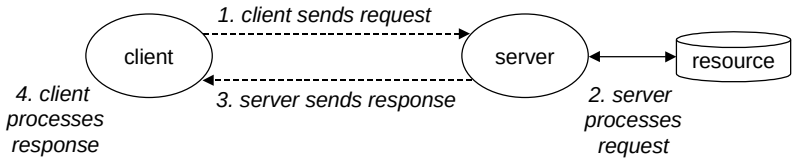

Figure 1: **A client-server transaction.**

A transaction consists of four steps:[1]

1. When a client needs service, it initiates a transaction by sending a *request* to the server. For example, when a Web browser needs a file, it sends a request to a Web server.

2. The server receives the request, interprets it, and manipulates its resource in the appropriate way. For example, when a Web server receives a request from a browser, it reads a disk file.

3. The server sends a *response* to the client, and then waits for the next request. For example, a Web server sends the file back to a client.

4. The client receives the response and manipulates it. For example, after a Web browser receives a page from the server, it displays it on the screen.

It is important to realize that clients and servers are processes, and not machines, or *hosts* as they are often called in this context. A single host can run many different clients and servers concurrently, and a client and server transaction can be on the same or different hosts. The client-server model is the same, regardless of the mapping of clients and servers to hosts.

3 Networks

Clients and servers often run on separate hosts and communicate using the hardware and software resources of a *computer network.* Networks are sophisticated systems and we can only hope to scratch the surface here. Our aim is to give you a a workable mental model from a progammer's perspective.

To a host, a network is just another I/O device that serves as a source and sink for data, as shown in Figure 2. An adapter plugged into an expansion slot on the I/O bus provides the physical interface to the network. Data received from the network is copied from the adapter across the I/O and memory buses into memory, typically by a DMA transfer. Similarly, data can also be copied from memory to the network.

Physically, a network is a hierarchical system that is organized by geographical proximity. At the lowest level is a LAN (Local Area Network) that spans a building or a campus. The most popular LAN technology by far is *ethernet,* which was developed in the mid-1970's at Xerox PARC. Ethernet has proven to be remarkably resilient, evolving from 10 Mb/s transfer rates, to 100 Mb/s, and more recently to 1 Gb/s.

[1]Client-server transactions are *not* database transactions and do not share any of their properties. In this context, a transaction simply connotes a sequence of steps by a client and server.

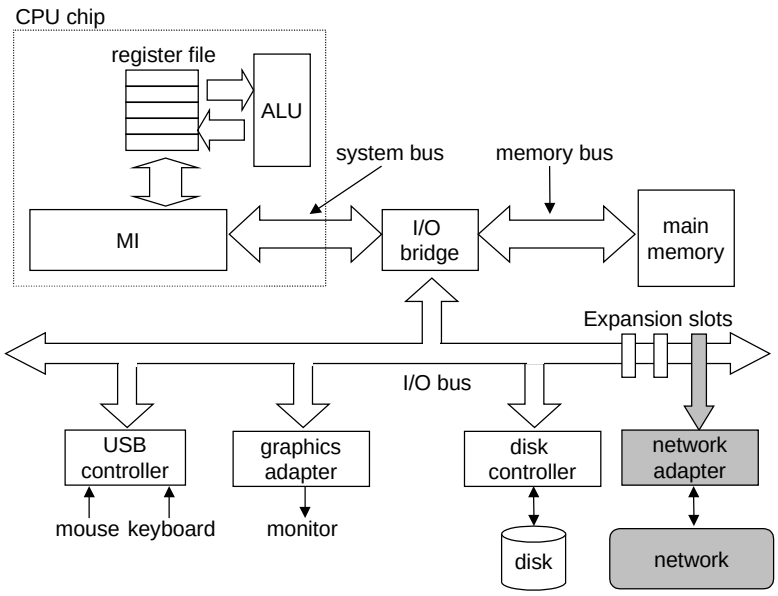

Figure 2: **Hardware organization of a network host.**

An *ethernet segment* consists of some wires (usually coaxial cables) and a small box called a *hub*, as shown in Figure 3. Ethernet segments typically span small areas, such as a room or a floor in a building. Each wire

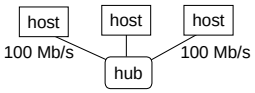

Figure 3: **Ethernet segment.**

has the same maximum bit bandwidth, typically 100 Mb/s or 1 Gb/s. One end is attached to an adapter on a host, and the other end is attached to a *port* on the hub. A hub slavishly copies every bit that it receives on each port to every other port. Thus every host sees every bit.

Each ethernet adapter has a globally unique 48-bit address that is stored in a persistent memory on the adapter. A host can send a chunk of bits called a *frame* to any other host on the segment. Each frame includes some fixed number of *header* bits that identify the source and destination of the frame and the frame length, followed by a *payload* of data bits. Every host adapter sees the frame, but only the destination host actually reads it.

Multiple ethernet segments can be connected into larger LANs, called *bridged ethernets*, using a set of wires and small boxes called *bridges*, as shown in Figure 4. Bridged ethernets can span entire buildings or campuses. In a bridged ethernet, some wires connect bridges to bridges, and others connect bridges to hubs. The bandwidths of the wires can be different. In our example, the bridge-bridge wire has a 1 Gb/s bandwidth, while the four hub-bridge wires have bandwidths of 100 Mb/s.

Bridges make better use of the available wire bandwidth than hubs. Using a clever distributed algorithm, they automatically learn over time which hosts are reachable from which ports, and then selectively copy

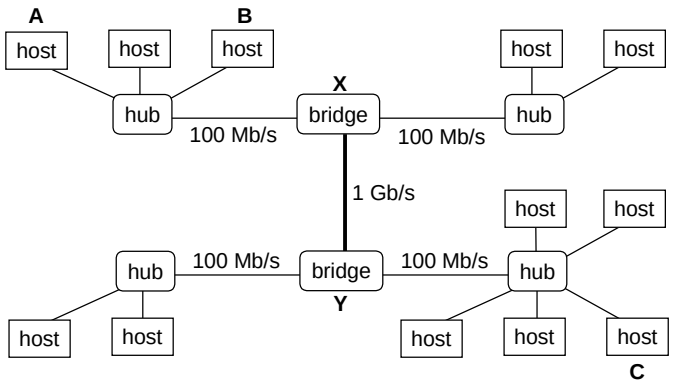

Figure 4: **Bridged ethernet segments.**

frames from one port to another only when it is necessary. For example, if host A sends a frame to host B, which is on the segment, then bridge X will throw away the frame when it arrives at its input port, thus saving bandwidth on the other segments. However, if host A sends a frame to host C on a different segment, then bridge X will copy the frame only to the port connected to bridge Y, which will copy the frame only to the port connected to bridge C's segment.

To simplify our pictures of LANs, we will draw the hubs and bridges and the wires that connect them as a single horizontal line, as shown in Figure 5.

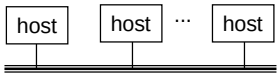

Figure 5: **Conceptual view of a LAN.**

At a higher level in the hierarchy, multiple incompatible LANs can be connected by specialized computers called *routers* to form an *internet* (interconnected network).[2] Each router has an adapter (port) for each network that it is connected to. Routers can also connect high-speed point-to-point phone connections, which are examples of networks known as WANs (Wide-Area Networks), so called because they span larger geographical areas than LANs. In general, routers can be used to build internets from arbitrary collections of LANs and WANs. For example, Figure 6 shows an example internet with a pair of LANs and WANs connected by three routers.

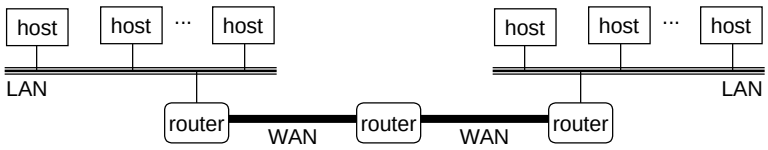

Figure 6: **A small internet.** Two LANs and two WANs are connected by three routers.

The crucial property of an internet is that it can consist of different LANs and WANs with radically different

[2]Lower-case *internet* denotes the general concept. Upper-case *Internet* denotes a specific implementation, the global IP Internet.

and incompatible technologies. Each host is physically connected to every other host, but how is it possible for some *source host* to send data bits to another *destination host* across all of these incompatible networks?

The solution is a layer of *protocol software* running on each host and router that smoothes out the differences between the different networks. This software implements a *protocol* that governs how hosts and routers cooperate in order to transfer data. The protocol must provide two basic capabilities:

- *Naming scheme.* Different LAN technologies have different and incompatible ways of assigning addresses to hosts. The internet protocol smoothes these differences by defining a uniform format for host addresses. Each host is then assigned at least one of these *internet addresses* that uniquely identifies it.
- *Delivery mechanism.* Different networking technologies have different and incompatible ways of encoding bits on wires and of packaging these bits into frames. The internet protocol smoothes these differences by defining a uniform way to bundle up data bits into discrete chunks called *packets*. A packet consists of a *header*, which contains the packet size and addresses of the source and destination hosts, and a *payload*, which contains data bits sent from the ource host.

Figure 7 shows an example of how hosts and routers use the internet protocol to transfer data across incompatible LANs.

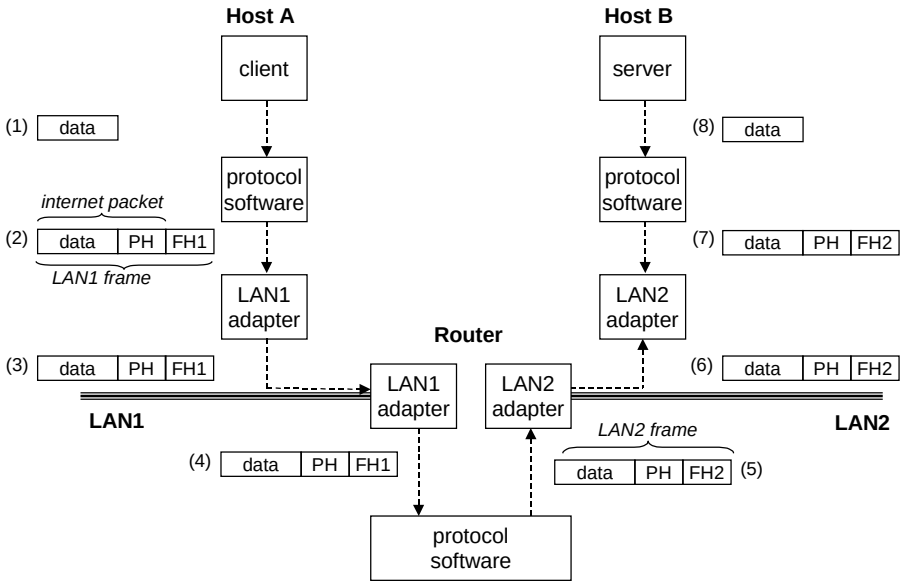

Figure 7: **How data travels from one host to another on an internet.** Key: PH: internet packet header, FH1: frame header for LAN1, FH2: frame header for LAN2.

The example internet consists of two LANs connected by a router. A client running on host A, which is attached to LAN1, sends a sequence of data bytes to a server running on host B, which is attached to LAN2. There are eight basic steps:

1. The client on host A invokes a system call that copies the data from the client's virtual address space into a kernel buffer.

2. The protocol software on host A creates a LAN1 frame by appending an internet header and a LAN1 frame header to the data. The internet header is addressed to internet host B. The LAN1 frame header is addressed to the router. It then passes the frame to the adapter. Notice that the payload of the LAN1 frame is an internet packet, whose payload is the actual user data. This kind of *encapsulation* is one of the fundamental insights of internetworking.

3. The LAN1 adapter copies the frame to the network.

4. When the frame reaches the router, the router's LAN1 adapter reads it from the wire and passes it to the protocol software.

5. The router fetches the destination internet address from the internet packet header and uses this as an index into a routing table to determine where to forward the packet, which in this case is LAN2. The router then strips off the old LAN1 frame header, prepends a new LAN2 frame header addressed to host B, and passes the resulting frame to the adapter.

6. The router's LAN2 adapter copies the frame to the network.

7. When the frame reaches host B, its adapter reads the frame from the wire and passes it to the protocol software.

8. Finally, the protocol software on host B strips off the packet header and frame header. The protocol software will eventually copy the resulting data into the server's virtual address space when the server invokes a system call that reads the data.

Of course, we are glossing over many difficult issues here. What if different networks have different maximum frame sizes? How do routers know where to forward frames? How are routers informed when the the network topology changes? What if packet gets lost? Nonetheless, our example captures the essence of the internet idea, and encapsulation is key.

4 The global IP Internet

The global IP Internet is the most famous and successful implementation of an internet. It has existed in one form or another for over thirty years. While the internal architecture of the Internet is complex and constantly changing, the organization of client-server applications has remained remarkably stable for more than 20 years. Figure 8 shows the basic hardware and software organization of an Internet client-server application.

Each Internet host runs software that implements the *TCP/IP* protocol (Transmission Control Protocol/Internet Protocol), which is supported by almost every modern computer system. Internet clients and servers communicate using a mix of *sockets interface* functions and Linux file I/O functions. (We will describe Linux file I/O in Section 5 and the sockets interface in Section 6.) The sockets functions are typically implemented as system calls that trap into the kernel and call various kernel-mode functions in TCP/IP.

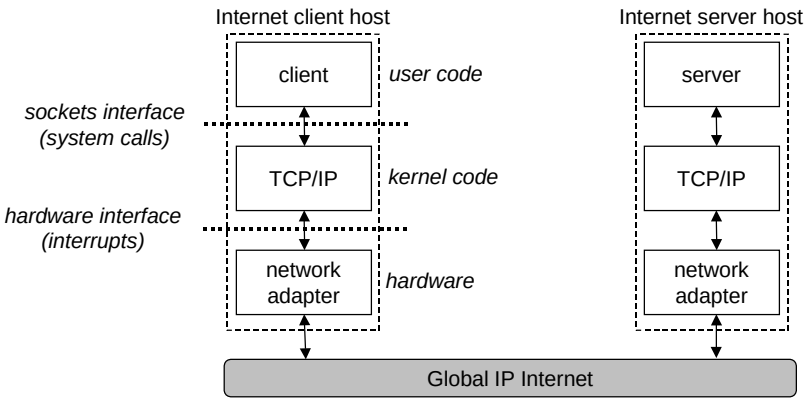

Figure 8: **Hardware and software organization of an Internet application.**

TCP/IP is actually of family of protocols, each of which contributes different capabilities. For example, the IP protocol provides the basic naming scheme and a delivery mechanism that can send packets known as *datagrams* from one Internet host to any another host. The IP mechanism is unreliable in the sense that it makes no effort to recover if datagrams are lost or duplicated in the network. UDP (Unreliable Datagram Protocol) extends IP slightly so that packets can be transfered from process to process, rather than host to host. TCP is a complex protocol that uses IP to provide reliable full-duplex connections between processes. To simplify our discussion, we will treat TCP/IP as a single monolithic protocol. We will not discuss its inner workings, and we will only discuss some of the basic capabilities that TCP and IP provide to application programs. We will not discuss UDP.

From a programmer's perspective, we can think of the Internet as a worldwide collection of hosts with the following properties:

- Hosts are mapped to a set of 32-bit *IP addresses.*
- The set of IP addresses is mapped to a set of identifiers called *Internet domain names.*
- A process on one host communicates with a process on another host over a *connection.*

The next three sections discuss these fundamental ideas in more detail.

4.1 IP addresses

An IP address is an unsigned 32-bit integer. Network programs store IP addresses in the *IP address structure* shown in Figure 9.[3] Because Internet hosts can have different host byte orders, TCP/IP defines a uniform *network byte order* (i.e., big-endian byte order) for any integer data item, such as an IP address, that is carried across the network in a packet header. Addresses in IP address structures are always stored in big-endian

[3]Storing a scalar address in a structure is an unfortunate historical artifact from the early Berkeley implementations of the sockets interface. It would make more sense to define a scalar type for IP addresses, but it is too late to change now because of the enormous installed base of applications.

netinet/in.h

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

netinet/in.h

Figure 9: **IP address structure.**

network byte order, even if the host byte order is little-endian. Linux provides the following functions for converting between network and host byte order.

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
                                        both return: value in network byte order
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
                                        both return: value in host byte order
```

The `hotnl` function converts a 32-bit integer from host byte to network byte order. The `ntohl` function converts a 32-bit integer from network byte order to host byte order. The `htons` and `ntohs` functions perform corresponding conversions for 16-bit integers.

IP addresses are typically presented to humans in a form known as *dotted-decimal notation*, where each byte is represented by its decimal value and separated from the other bytes by a period. For example, `128.2.194.242` is the dotted-decimal representation of the address `0x8002c2f2`. You can use the Linux `hostname` command to determine the dotted-decimal address of your own host:

```
linux> hostname -i
128.2.194.242
```

Internet programs convert back and forth between IP addresses and dotted-decimal strings using the `inet_aton` and `inet_ntoa` functions:[4]

```
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
                                        returns: 1 if OK, 0 on error

char *inet_ntoa(struct in_addr in);
                                        returns: pointer to a dotted-decimal string
```

[4]The `"n"` denotes *network* representation. The `"a"` denotes *application* representation.

The `inet_aton` function converts a dotted-decimal string (`cp`) to an IP address in network byte order (`inp`). Similarly, the `inet_ntoa` function converts an IP address in network byte order to its corresponding dotted-decimal string. Notice that a call to `inet_aton` passes a pointer to a structure, while a call to `inet_ntoa` passes the structure itself.

Problem 1 [Category 1]:
Complete the following table.

Hex address	Dotted-decimal address
`0x0`	
`0xffffffff`	
`0xef000001`	
	`205.188.160.121`
	`64.12.149.13`
	`205.188.146.23`

Problem 2 [Category 2]:
Write a program `hex2dd.c` that converts its hex argument to a dotted-decimal string and prints the result. For example,

```
linux> hex2dd 0x8002c2f2
128.2.194.242
```

Problem 3 [Category 2]:
Write a program `dd2hex.c` that converts its dotted-decimal argument to a hex number and prints the result. For example,

```
% dd2hex 128.2.194.242
0x8002c2f2
```

4.2 Internet domain names

Internet clients and servers use IP addresses when they communicate with each other. However, large integers are difficult for people to remember, so the Internet also defines a separate set of more human-friendly *domain names* as well as a mechanism that maps the set of domain names to the set of IP addresses. A domain name is a sequence of words (letters, numbers, and dashes) separated by periods. For example,

```
kittyhawk.cmcl.cs.cmu.edu
```

The set of domain names forms a hierarchy and each domain name encodes its position in the hierarchy. An example is the easiest way to understand this. Figure 10 shows a portion of the domain name hierarchy. The hierarchy is represented as a tree. The nodes of the tree represent domain names that are formed

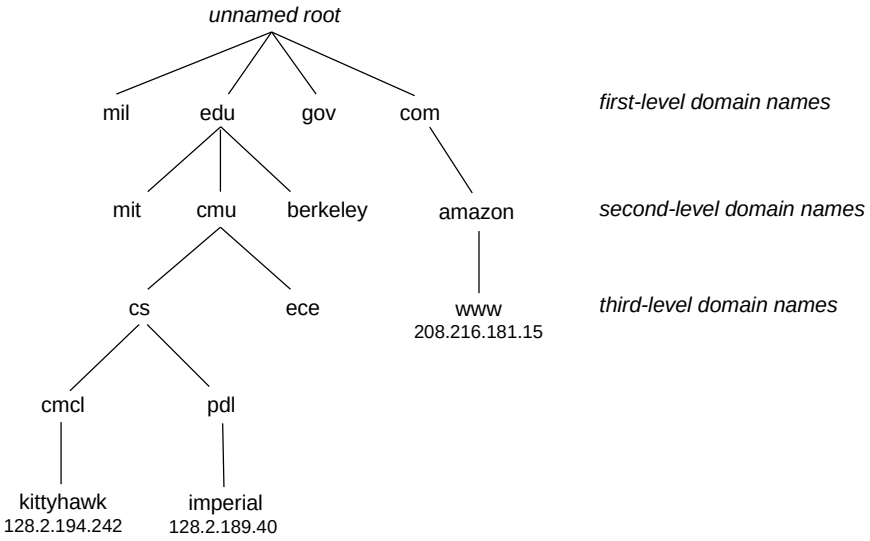

Figure 10: **Subset of the Internet domain name hierarchy.**

by the path back to the root. Sub-trees are referred to as *subdomains.* The first level in the hierarchy is an unnamed root node. The next level is a collection of *first-level domain names* that are defined by a non-profit organization called ICANN (Internet Corporation for Assigned Names and Numbers). Common first-level domains include `com`, `edu`, `gov`, `org`, and `net`.

At the next level are *second-level* domain names such as `cmu.edu`, which are assigned on a first-come first-serve basis by various authorized agents of ICANN. Once an organization has received a second-level domain name, then it is free to create any other new domain name within its subdomain.

The Internet defines a mapping between the set of domain names and the set of IP addresses. Until 1988, this mapping was maintained manually in a single text file called `hosts.txt`. Since then, the mapping has been maintained in a distributed world-wide database known as *DNS* (Domain Naming System).

The DNS database consists of millions of the *host entry structures* shown in Figure 11, each of which defines the mapping between a set of domain names (an official name and a list of aliases) and a set of IP addresses. In a mathematical sense, we can think of each host entry as an equivalence class of domain names and IP addresses.

Internet applications retrieve arbitrary host entries from the DNS database by calling the `gethostbyname` and `gethostbyaddr` functions.

netdb.h

```
/* DNS host entry structure */
struct hostent {
    char   *h_name;        /* official domain name of host */
    char   **h_aliases;    /* null-terminated array of domain names */
    int    h_addrtype;     /* host address type (AF_INET) */
    int    h_length;       /* length of an address, in bytes */
    char   **h_addr_list;  /* null-terminated array of in_addr structs */
};
```

netdb.h

Figure 11: **DNS host entry structure.**

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
                                    returns: non-NULL pointer if OK, NULL pointer on error with h_errno set
struct hostent *gethostbyaddr(const char *addr, int len, 0);
                                    returns: non-NULL pointer if OK, NULL pointer on error with h_errno set
```

The `gethostbyname` returns the host entry associated with the domain name `name`. The `gethostbyaddr` function returns the host entry associated with the IP address `addr`. The second argument gives the length in bytes of an IP address, which for the current Internet is always four bytes. For our purposes, the third argument is always zero.

We can explore some of the properties of the DNS mapping with the `hostinfo` program in Figure 12, which reads a domain name or dotted-decimal address from the command line and displays the corresponding host entry.

Each Internet host has the locally-defined domain name `localhost`, which always maps to the *loopback address* `127.0.0.1`:

```
kittyhawk> hostinfo localhost
official hostname: localhost
alias: localhost.localdomain
address: 127.0.0.1
```

The `localhost` name provides a convenient and portable way to reference clients and servers that are running on the same machine, which can be especially useful for debugging. We can use the Linux `hostname` program to determine the real domain name of our local host:

```
linux> hostname
kittyhawk.cmcl.cs.cmu.edu
```

In the simplest case there is a one-to-one mapping between a domain name and an IP address:

../code/net/hostinfo.c

```
1  #include "ics.h"
2
3  int main(int argc, char **argv)
4  {
5      char **pp;
6      struct in_addr addr;
7      struct hostent *hostp;
8
9      if (argc != 2) {
10         fprintf(stderr, "usage: %s <domain name or dotted-decimal>\n",
11                 argv[0]);
12         exit(0);
13     }
14
15     if (inet_aton(argv[1], &addr) != 0)
16         hostp = Gethostbyaddr((const char *)&addr, sizeof(addr), AF_INET);
17     else
18         hostp = Gethostbyname(argv[1]);
19
20     printf("official hostname: %s\n", hostp->h_name);
21
22     for (pp = hostp->h_aliases; *pp != NULL; pp++)
23         printf("alias: %s\n", *pp);
24
25     for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
26         addr.s_addr = *((unsigned int *)*pp);
27         printf("address: %s\n", inet_ntoa(addr));
28     }
29     exit(0);
30 }
```

../code/net/hostinfo.c

Figure 12: `hostinfo`**: Retrieves and prints a DNS host entry.**

```
linux> hostinfo kittyhawk.cmcl.cs.cmu.edu
official hostname: kittyhawk.cmcl.cs.cmu.edu
address: 128.2.194.242
```

However, in some cases, multiple domain names are mapped to the same IP address:

```
linux> hostinfo cs.mit.edu
official hostname: EECS.MIT.EDU
alias: cs.mit.edu
address: 18.62.1.6
```

In the most general case, multiple domain names can be mapped to multiple IP addresses:

```
linux> hostinfo www.aol.com
official hostname: aol.com
alias: www.aol.com
address: 205.188.160.121
address: 64.12.149.13
address: 205.188.146.23
```

Finally, we notice that some valid domain names are not mapped to any IP address:

```
kittyhawk> hostinfo edu
Gethostbyname error: No address associated with name
kittyhawk> hostinfo cmcl.cs.cmu.edu
Gethostbyname error: No address associated with name
```

Problem 4 [Category 1]:
Run `hostinfo aol.com` three times in a row on your system.

A. What do you notice about the ordering of the IP addresses in the three host entries?

B. How might this ordering be useful?

4.3 Internet connections

Internet clients and servers communicate by sending and receiving streams of bytes over *connections.* A connection is *point-to-point* in the sense that it connects a pair of processes. It is *full-duplex* in the sense that data can flow in both directions at the same time. And it is *reliable* in the sense that — barring some catastrophic failure such as a cable cut by a careless backhoe operator — the stream of bytes sent by the source process is eventually received by the destination process in the same order it was sent.

A *socket* is an endpoint of a connection. Each socket has a corresponding *socket address* that consists of an Internet address and an 16-bit integer *port*, and is denoted by `address:port`. The port in the client's

socket address is assigned automatically by the kernel when the client makes a connection request, and is known as an *ephemeral port.* However, the port in the server's socket address is typically some *well-known port* that is associated with the service. For example, Web servers typically use port 80, and email servers use port 25. On Linux machines, the file `/etc/services` contains a comprehensive list of the services provided on that machine, along with their well-known ports.

A connection is uniquely identified by the socket addresses of its two endpoints. This pair of socket addresses is known as a *socket pair* and is denoted by the tuple

```
(cliaddr:cliport, servaddr:servport)
```

where `cliaddr` is the client's IP address, `cliport` is the client's port, `servaddr` is the server's IP address, and `servport` is the server's port. For example, Figure 13 shows a connection between a Web client and a Web server.

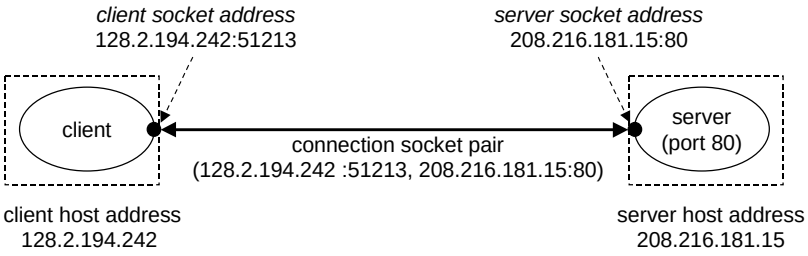

Figure 13: **Anatomy of an Internet connection**

In this example, the Web client's socket address is

```
128.2.194.242:51213
```

where port `51213` is an ephemeral port assigned by the kernel. The Web server's socket address is

```
208.216.181.15:80
```

where port `80` is the well-known port associated with Web services. Given these client and server socket addresses, the connection between the client and server is uniquely identified by the socket pair

```
(128.2.194.242:51213, 1208.216.181.15:80).
```

In Section 6 we will learn how C programs use the sockets interface to establish connections between clients and servers. But since sockets are modeled in Linux as files, we must first develop an understanding of Linux file I/O, which is the topic of the next section.

5 Linux file I/O

A Linux *file* is a sequence of n bytes

$$B_0, B_1, \ldots, B_k, \ldots, B_{n-1}.$$

All I/O devices, such as networks, disks, and terminals, are modeled as files, and all input and output is performed by reading and writing the appropriate files. This elegant mapping of devices to files allows Linux to export a simple low-level application interface, known as *Linux I/O*, that enables all input and output to be performed in a uniform and consistent way.

An application announces its intention to access an I/O device by asking the kernel to *open* the corresponding file. The kernel returns a small non-negative integer, called a *descriptor*, that identifies the file in all subsequent operations on the file. The kernel keeps track of all information about the open file; the application keeps keep track of only the descriptor.

The kernel maintains a *file position* k, initially 0, for each open file. An application can set the current file position k explicitly by performing a *seek* operation.

A *read* operation copies $m > 0$ bytes from the file to memory, starting at the current file position k, and then incrementing k by m. A read operation with $k \geq n$ triggers a condition known as *end-of-file* (EOF), which can be detected by the application. Notice that there is no explicit "EOF character" at the end of a file. Similarly, a *write* operation copies $m > 0$ bytes from memory to a file, starting at the current file position k, and then updating k.

When an application is finished reading and writing the file, it informs the kernel by asking it to *close* the file. The kernel frees the structures it created when the file was opened and restores the descriptor to a pool of available descriptors. The next file that is opened is guaranteed to receive the smallest available descriptor in the pool. When a process terminates for any reason, the kernel closes all open files, and frees their memory resources.

By convention, each process created by a Linux shell begins life with three open files: *standard input* (descriptor 0), *standard output* (descriptor 1), and *standard error* (descriptor 2). The system header file `unistd.h` defines the following constants,

```
#define STDIN_FILENO  0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

which for clarity can be used instead of explicit descriptor values.

5.1 The `read` and `write` functions

Applications perform input and output by calling the `read` and `write` functions, respectively.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
                                    returns: number of bytes read if OK, 0 on EOF, -1 on error

ssize_t write(int fd, const void *buf, size_t count);
                                    returns: number of bytes written if OK, -1 on error
```

The `read` function copies at most `count` bytes from the current file position of descriptor `fd` to memory location `buf`. A return value of -1 indicates an error, and a return value of 0 indicates EOF. Otherwise, the return value indicates the number of bytes that were actually transferred.

The `write` function copies at most `count` bytes from memory location `buf` to the current file position of descriptor `fd`. Figure 14 shows a program that uses `read` and `write` calls to copy the standard input to the standard output, one byte at a time.

../code/net/cpstdin.c

```
 1 #include "ics.h"
 2
 3 int main(void)
 4 {
 5     char c;
 6
 7     /* copy stdin to stdout, one byte at a time */
 8     while(Read(STDIN_FILENO, &c, 1) != 0)
 9         Write(STDOUT_FILENO, &c, 1);
10     exit(0);
11 }
```

../code/net/cpstdin.c

Figure 14: **Copies standard input to standard output.**

5.2 Robust file I/O with the `readn` and `writen` functions.

In some situations, `read` and `write` transfer fewer bytes than the application requests. Such *short counts* do not indicate an error, and can occur for a number of reasons:

- *Encountering end-of-file on reads.* If the file contains only 20 more bytes and we are reading in 50-byte chunks, then the current `read` will return a short count of 20. The next `read` will signal EOF by returning a short count of zero.
- *Reading text lines from a terminal.* If the open file is associated with a terminal (i.e., a keyboard and display), then the `read` function will transfer the next text line.

- *Reading and writing network sockets.* If the open file corresponds to a network socket, then internal buffering constraints and long network delays can cause `read` and `write` to return short counts.

Robust applications in general, and network applications in particular, must anticipate and deal with short counts. In Figure 14 we skirted this issue by transferring one byte at a time. While technically correct, this approach is grossly inefficient because it requires $2n$ system calls. Instead, you should use the `readn` and `writen` functions from W. Richard Stevens's classic network programming text [10].

```
#include "ics.h"

ssize_t readn(int fd, void *buf, size_t count);
ssize_t writen(int fd, void *buf, size_t count);
```
both return: number of bytes read (0 if EOF) or written, -1 on error

The code for these functions is shown in Figure 15. The `readn` function returns a short count only when the input operation extends past the end of file. Other short counts are handled by repeatedly invoking `read` until `count` bytes have been transferred. The `writen` function never returns a short count.

Notice that each routine manually restart `read` and `write` if they are interrupted by the return from an application signal handler (lines 9-10). Manual restarts are unnecessary on Linux systems, which automatically restart interrupted `read` and `write` calls. However, other systems such as Solaris do not restart interrupted system calls, and on these systems we must manually restart them.

5.3 Robust input of text lines using the `readline` function

A *text line* is a sequence of ASCII characters terminated by a newline character.[5] Many network applications, such as Web clients and servers, communicate using sequences of text lines. For these programs you should use the `readline` function [10] whenever you input a text line.

```
#include "ics.h"

ssize_t readline(int fd, void *buf, size_t maxlen);
```
returns: number of bytes read (0 if EOF), -1 on error

The `readline` function has the same semantics as the `fgets` function in the C Standard I/O library. It reads the next text line from file `fd` (including the terminating newline character), copies it to memory location `buf`, and terminates the text line with the null character. `Readline` reads at most `maxlen-1` bytes, leaving room for the terminating zero. If the text line is longer than `maxlen-1` bytes, then `readline` simply returns the first `maxlen-1` bytes of the line. Figure 16 shows the code for the `readline` package. It is somewhat subtle and needs to be studied carefully.

The `my_read` function copies the next character in the file to location `ptr`. It returns -1 on error (with

[5] The newline character is the same as the ASCII line feed character (LF) and has a numeric value of `0x0a`.

../code/ics.c

```
ssize_t readn(int fd, void *buf, size_t count)
{
    size_t nleft = count;
    ssize_t nread;
    char *ptr = buf;

    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;      /* and call read() again */
            else
                return -1;      /* errno set by read() */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        ptr += nread;
    }
    return (count - nleft);    /* return >= 0 */
}
```

../code/ics.c

../code/ics.c

```
ssize_t writen(int fd, const void *buf, size_t count)
{
    size_t nleft = count;
    ssize_t nwritten;
    const char *ptr = buf;

    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) <= 0) {
            if (errno == EINTR)
                nwritten = 0;    /* and call write() again */
            else
                return -1;       /* errorno set by write() */
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return count;
}
```

../code/ics.c

Figure 15: `readn` **and** `writen`**: Robust versions of** `read` **and** `write` Adapted from [10].

../code/ics.c

```
1  static ssize_t my_read(int fd, char *ptr)
2  {
3      static int read_cnt = 0;
4      static char *read_ptr, read_buf[MAXLINE];
5
6      if (read_cnt <= 0) {
7      again:
8          if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
9              if (errno == EINTR)
10                 goto again;
11             return -1;
12         }
13         else if (read_cnt == 0)
14             return 0;
15         read_ptr = read_buf;
16     }
17     read_cnt--;
18     *ptr = *read_ptr++;
19     return 1;
20 }
21
22 ssize_t readline(int fd, void *buf, size_t maxlen)
23 {
24     int n, rc;
25     char c, *ptr = buf;
26
27     for (n = 1; n < maxlen; n++) {  /* notice that loop starts at 1 */
28         if ( (rc = my_read(fd, &c)) == 1) {
29             *ptr++ = c;
30             if (c == '\n')
31                 break;          /* newline is stored, like fgets() */
32         }
33         else if (rc == 0) {
34             if (n == 1)
35                 return 0;       /* EOF, no data read */
36             else
37                 break;          /* EOF, some data was read */
38         }
39         else
40             return -1;  /* error, errno set by read() */
41     }
42     *ptr = 0;   /* null terminate like fgets() */
43     return n;
44 }
```

../code/ics.c

Figure 16: `readline` **package: Reads a text line from a descriptor.** Adapted from [10].

errno) set appropriately, 0 on EOF, and 1 otherwise. Notice that my_read is a static function, and thus is not visible to applications.

To improve efficiency, my_read maintains a static buffer that it refreshes in MAXLINE-sized blocks. Variable read_ptr points to the buffer byte to return to the caller, and variable read_cnt is the number of bytes in the buffer that have yet to be returned to the caller. The function initiates a new block-read operation each time read_cnt drops to zero (line 6).

The readline function calls my_read at most maxlen-1 times, terminating either when it encounters a newline character (line 30), when my_read returns EOF (line 35), or when my_read indicates an error (line 40).

> **Problem 5** [Category 2]:
> Modify the cpstdinbuf program in Figure 14 so that it uses readn and writen to copy standard input to standard output, MAXBUF bytes at a time.

5.4 The stat function

An application retrieves information about disk files by calling the stat function.

```
#include <unistd.h>
#include <sys/stat.h>

int stat(const char *filename, struct stat *buf);
```
returns: 0 if OK, -1 on error

The stat function takes as input a filename, such as /usr/dict/words, and fills in the members of a stat structure shown in Figure 17. We will need the st_mode and st_size members of the stat structure when we discuss Web servers in Section 8. The st_mode member encodes both the file type and the file protection bits. The st_size member contains the file size in bytes. The meaning of the other members is beyond our scope.

A Linux system recognizes a number of different file types. For example, a *regular file* contains some sort of binary or text data. To the kernel there is no difference between text files and binary files. A *directory file* contains information about other files. And a *socket* is a file that is used to communicate with another process across a network. Linux provides macro predicates for determining the file type. Figure 18 shows a subset. Each file type macro takes an st_mode member as its argument.

The protection bits in st_mode can be tested using the bit masks in Figure 19. For example, the following code fragment checks for a regular file that the current process has permission to read:

```
if (S_ISREG(stat.st_mode) && (stat.st_mode & S_IRUSR))
    printf("This is a regular file that I can read\n");
```

statbuf.h (included by sys/stat.h)

```
/* file info returned by the stat function */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

statbuf.h (included by sys/stat.h)

Figure 17: **The `stat` structure.**

Macro	Description
S_ISREG()	Is this a regular file?
S_ISDIR()	Is this a directory file?

Figure 18: **Some macros for determining the type of file.** Defined in `sys/stat.h`

`st_mode` mask	Description
S_IRUSR	User (owner) can read this file
S_IWUSR	User (owner) can write this file
S_IXUSR	User (owner) can execute this file
S_IRGRP	Group members can read this file
S_IWGRP	Group members can write this file
S_IXGRP	Group members can execute this file
S_IROTH	Others (anyone) can read this file
S_IWOTH	Others (anyone) can write this file
S_IXOTH	Others (anyone) can execute this file

Figure 19: **Masks for checking protection bits.** Defined in `sys/stat.h`

5.5 The `dup2` function

Linux shell provides an I/O redirection operator that allows users to redirect the standard output to a disk file. For example,

```
linux>  ls >foo
```

writes the standard output of the `ls` program to the file `foo`. As we shall see in Section 8, a Web server performs a similar kind of redirection when it runs a CGI program on behalf of the client. One way to accomplish I/O redirection is to use the `dup2` function.

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```
returns: nonnegative descriptor if OK, -1 on error

The `dup2` function duplicates descriptor `oldfd`, assigns it to descriptor `newfd`, and returns `newfd`. If `newfd` was already open, then `dup2` closes `newfd` before it duplicates `oldfd`.

For each process, the kernel maintains a *descriptor table* that is indexed by the process's open descriptors. The entry for an open descriptor points to a *file table entry* that consists of, among other things, the current file position and a *reference count* of the number of descriptor entries that currently point to it. The file table entry in turn points to an *i-node* table entry that characterizes the physical location of the file on disk, and contains most of the information in the `stat` structure, including the `st_mode` and `st_size` members.

Typically there is a one-to-one mapping between descriptors and files. For example, suppose we have the situation in Figure 20 where descriptor 1 (stdout) corresponds to file A (say a terminal), while descriptor 4 corresponds to file B (say a disk). The reference counts for the A and B are both equal to 1.

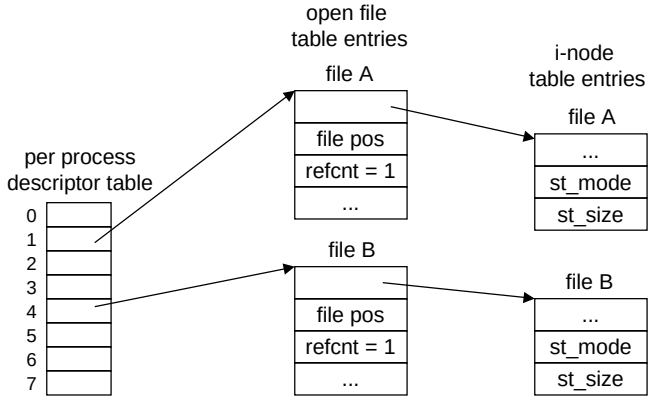

Figure 20: **Kernel data structures before** `dup2(4,1)`

The `dup2` function allows multiple descriptors to be associated with the same file. For example, Figure 21 shows the situation after calling `dup2(4,1)`. Both descriptors now correspond to file B, file A has been

closed, and the reference count for file B has been incremented. From this point on, any data that is written to standard output is redirected to file B.

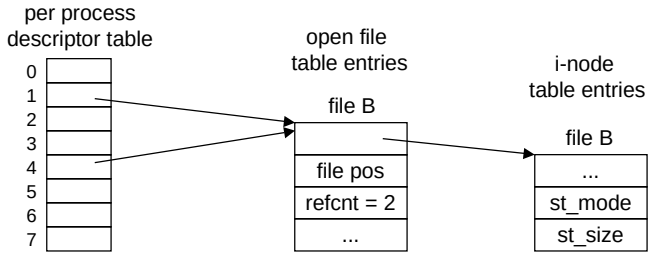

Figure 21: **Kernel data structures after** `dup2(4,1)`

5.6 The `close` function

A process informs the kernel that is finished reading and writing a file by calling the `close` function.

```
#include <unistd.h>

int close(int fd);
```

returns: zero if OK, -1 on error

The kernel does not delete the associated file table entry unless the reference count is zero. For example, suppose we have the situation in Figure 21, where descriptors 1 and 4 both point to the same file table entry. If we were to close descriptor 1, then we could still perform input and output on descriptor 4.

Closing a closed descriptor is an error, but unfortunately programmers rarely check for this error. In Section 7.2 we will see that threaded programs that close already closed descriptors can suffer from a subtle race condition that sometimes causes a thread to catastrophically close another thread's open descriptor. The bottom line: always check return codes, even for seemingly innocuous functions such as `close`.

5.7 Other Linux I/O functions

Linux provides two additional I/O functions, `open` and `lseek`. The `open` function creates new files and opens existing files. In each case, it returns a descriptor that can be used by other Linux file I/O routines. We will not describe `open` in any more depth because a clear understanding requires numerous details about Linux file systems that are not relevant to network programming.

The `lseek` function modifies the current file position. Since it is illegal to change the current file position of a socket, we will not discuss this function either.

5.8 Linux I/O vs. Standard I/O

The ANSII C standard defines a set of input and output functions, known as the *standard I/O library*, that provide a higher-level and more convenient alternative to the underlying Linux I/O functions. Functions such as `fopen`, `fclose`, `fseek`, `fread`, `fwrite`, `fgetc`, `fputc`, `fgets`, `fputs`, `fscanf`, and `fprintf` are commonly used standard I/O functions.

The standard I/O functions are the method of choice for input and output on disk and terminal devices. And in fact, most C programmers use these functions exclusively, never bothering with the the lower-level Linux I/O functions. Unfortunately, standard I/O poses some tricky problems when we attempt to use it for input and output on network sockets.

The standard I/O models a file as a *stream*, which is a higher-level abstraction of a file descriptor. Like descriptors, streams can be full-duplex, so a program can perform input and output on the same stream. However, there are restrictions on full-duplex streams that interact badly with restrictions on sockets:

- An input function cannot follow an output function without an intervening call to `fflush`, `fseek`, `fsetpos`, or `rewind`. For efficiency, standard I/O streams are buffered. The `fflush` function empties the buffer associated with a stream. The latter three functions use the Linux I/O `lseek` function to reset the current file position.
- Similarly, an output function cannot follow an input function without an intervening call to `fseek`, `fsetpos`, or `rewind`, unless the input function encounters an end-of-file.

These restrictions pose a problem for network applications because it is illegal to use the `lseek` function on a network socket. The first restriction on stream I/O can be worked around by a discipline of flushing the buffer before every input operation. The only way to work around the second restriction is to open two streams on the same open socket descriptor, one for reading and one for writing.

```
FILE *fpin, *fpout;

fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");
```

However, this approach has problems as well, because it requires the application to call `fclose` on both streams in order to free the memory resources associated with each stream and avoid a memory leak:

```
fclose(fpin);
fclose(fpout);
```

Each of these operations attempts to close the same underlying socket descriptor, so the second `close` operation will fail. While this is not necessarily a fatal error in a sequential program, closing the same descriptor twice in a threaded program is a recipe for disaster. Thus, we recommend avoiding the standard I/O functions for input and output on network sockets. Use the robust `readn`, `writen`, and `readline` functions instead.

6 The sockets interface

The *sockets interface* is a set of functions that are used in conjunction with the Linux file I/O functions to build network applications. It has been implemented on most modern systems, including Linux and the other Unix variants, Windows, and Macintosh systems. Figure 22 gives an overview of the sockets interface in the context of a typical client-server transaction. You should use this picture as road map when we discuss the individual functions.

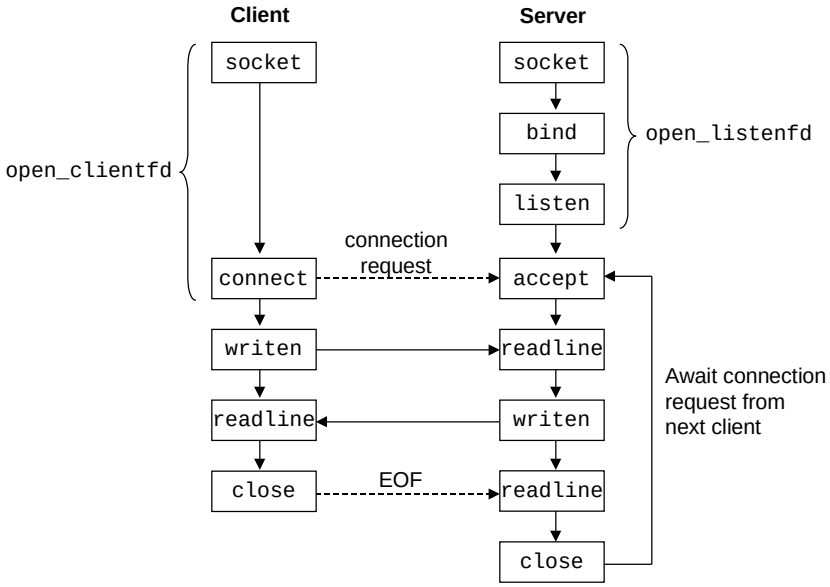

Figure 22: **Overview of the sockets interface.**

6.1 Socket address structures

From the perspective of the Linux kernel, a socket is an endpoint for communication. From the perspective of a Linux program, a socket is an open file with a corresponding descriptor.

Internet socket addresses are stored in 16-byte structures of the type `sockaddr_in` shown in Figure 23.[6] For Internet applications, the `sin_family` member is AF_INET, the `sin_port` member is a 16-bit port number, and the `sin_addr` member is a 32-bit IP address. The IP address and port number are always stored in network (big-endian) byte order.

The generic `sockaddr` structure in Figure 23 is an unfortunate historical artifact that confuses many programmers. The sockets interface was designed in the early 1980's to work with any type of underlying network protocol, each of which was expected to define its own 16-byte protocol-specific `sockaddr_xx` socket address structure. No one at the time had any inkling that TCP/IP would become so dominant.

The `connect`, `bind`, and `accept` functions require a pointer to protocol-specific socket address structure. The problem faced by the designers of the sockets interface was how to define these functions to

[6] The `_in` suffix is short for *internet*, not *input*.

sockaddr: socketbits.h (included by socket.h). sockaddr_in: netinit/in.h

```
/* Generic socket address structure (for connect, bind, and accept) */
struct sockaddr {
    unsigned short  sa_family;    /* protocol family */
    char            sa_data[14];  /* address data.  */
};

/* Internet-style socket address structure */
struct sockaddr_in  {
    unsigned short  sin_family;  /* address family (always AF_INET) */
    unsigned short  sin_port;    /* port number in network byte order */
    struct in_addr  sin_addr;    /* IP address in network byte order */
    unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```

sockaddr: socketbits.h (included by socket.h). sockaddr_in: netinit/in.h

Figure 23: **Socket address structures.** The `in_addr` struct is shown in Figure 9.

accept any kind of socket address structure. Today we would use the generic `void *` pointer, which did not exist in C at that time. The solution was to define sockets functions to expect a pointer to a generic `sockaddr` structure, and then require applications to cast pointers to protocol-specific structures to this generic structure.

To simplify our code examples, we will follow Stevens's lead and define the following type

```
typedef struct sockaddr SA;
```

that we use whenever we need to cast a protocol-specific structure to a generic one.

6.2 The `socket` function

Clients and servers use the `socket` function to create a *socket descriptor.*

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

returns: nonnegative descriptor if OK, -1 on error

In our codes, we will always call the `socket` function with the following arguments:

```
sockfd = Socket(AF_INET, SOCK_STREAM, 0);
```

where AF_INET indicates that we are using the Internet, and SOCK_STREAM indicates that the socket will be an endpoint for an Internet connection. The `sockfd` descriptor returned by `socket` is only partially

opened and cannot yet be used for reading and writing. How we finish opening the socket depends on whether we are a client or a server.

6.3 The `connect` function

A client establishes a connection with a server by calling the `connect` function.

```
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen );
                                                    returns: 0 if OK, -1 on error
```

The `connect` function attempts to establish an Internet connection with the server at socket address `serv_addr`, where `addrlen` is `sizeof(sockaddr_in)`. The `connect` function blocks until either the connection is successfully established, or an error occurs. If successful, the `sockfd` descriptor is now ready for reading and writing, and the resulting connection is characterized by the socket pair

```
(x:y, serv_addr.sin_addr:serv_addr.sin_port)
```

where `x` is the client's IP address and `y` is the ephemeral port that uniquely identifies the client process on the client host.

Figure 24 shows our `open_clientfd` helper function that a client uses to establish a connection with a server running on host `hostname` and listening for connection requests on the well-known port `port`. It returns a file descriptor that is ready for input and output using Linux file I/O.

After creating the socket descriptor (line 11), we retrieve the DNS host entry for the server (line 14) and copy the first IP address in the host entry (which is already in network byte order) to the server's socket address structure (lines 17-18). After initializing the socket address structure with the server's well-known port number in network byte order (line 19), we initiate the connect request to the server (line 22). When `connect` returns, we return the socket descriptor to the client, which can immediately begin using Linux I/O operations to communicate with the server.

6.4 The `bind` function

The remaining functions — `bind`, `listen`, and `accept` — are used by servers to establish connections with clients.

```
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
                                                    returns: 0 if OK, -1 on error
```

../code/ics.c

```
1  int open_clientfd(char *hostname, int port)
2  {
3      int clientfd;
4      struct hostent *hp;
5      struct sockaddr_in serveraddr;
6
7      clientfd = Socket(AF_INET, SOCK_STREAM, 0);
8
9      /* fill in the server's IP address and port */
10     hp = Gethostbyname(hostname);
11     bzero((char *) &serveraddr, sizeof(serveraddr));
12     serveraddr.sin_family = AF_INET;
13     bcopy((char *)hp->h_addr,
14           (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
15     serveraddr.sin_port = htons(port);
16
17     /* establish a connection with the server */
18     Connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr));
19
20     return clientfd;
21 }
```

../code/ics.c

Figure 24: `open_clientfd`**: helper function that establishes a connection with a server.**

The `bind` function tells the kernel to associate the server's socket address in `my_addr` with the socket descriptor `sockfd`. The `addrlen` argument is `sizeof(sockaddr_in)`.

6.5 The `listen` function

Clients are active entities that initiate connection requests. Servers are passive entities that wait for connection requests from clients. By default, the kernel assumes that a descriptor created by the `socket` function corresponds to an *active socket* that will live on the client end of a connection. A server calls the `listen` function to tell the kernel that the descriptor will be used by a server instead of a client.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```
returns: 0 if OK, -1 on error

The `listen` function converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients. The `backlog` argument is a hint about the number of outstanding connection requests that the kernel should queue up before it starts to refuse requests. The exact meaning of the `backlog` argument requires an understanding of TCP/IP that is beyond our scope. We will typically set it to a large value, such as 1024.

Figure 25 shows our `open_listenfd` helper function that opens and returns a listening socket ready to receive client connection requests on the well-known port `port`. After we create the `listenfd` socket descriptor (line 11), we use the `setsockopt` function (not described here) to configure the server so that it can be terminated and restarted immediately (lines 14-15). By default, a restarted server will deny connection requests from clients for approximately 30 seconds, which seriously hinders debugging.

In lines 20-23, we initialize the server's socket address structure in preparation for calling the `bind` function. In this case, we have used the INADDR_ANY wild card address to tell the kernel that this server will accept requests to any of the IP addresses for this host (line 22), and to well-known port `port` (line 23). Notice that we use the `htonl` and `htons` functions to convert the IP address and port number from host byte order to network byte order. Finally, we convert `listenfd` to a listening descriptor (line 27) and return it to the caller.

6.6 The `accept` function

Servers wait for connection requests from clients by calling the `accept` function.

```
#include <sys/socket.h>

int accept(int listenfd, struct sockaddr *addr, int *addrlen);
```
returns: nonnegative connected descriptor if OK, -1 on error

../code/ics.c

```
1  int open_listenfd(int port)
2  {
3      int listenfd;
4      int optval;
5      struct sockaddr_in serveraddr;
6
7      /* create a socket descriptor */
8      listenfd = Socket(AF_INET, SOCK_STREAM, 0);
9
10     /* eliminates "Address already in use" error from bind. */
11     optval = 1;
12     Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
13                (const void *)&optval , sizeof(int));
14
15     /* listenfd will be an endpoint for all requests to port
16        on any IP address for this host */
17     bzero((char *) &serveraddr, sizeof(serveraddr));
18     serveraddr.sin_family = AF_INET;
19     serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
20     serveraddr.sin_port = htons((unsigned short)port);
21     Bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr));
22
23     /* make it a listening socket ready to accept connection requests */
24     Listen(listenfd, LISTENQ);
25
26     return listenfd;
27 }
```

../code/ics.c

Figure 25: `open_listenfd`**: helper function that opens and returns a listening socket.**

The `accept` function waits for a connection request from a client to arrive on the listening descriptor `listenfd`, then fills in the client's socket address in `addr`, and returns a *connected descriptor* that can be used to communicate with the client using Linux I/O functions.

The distinction between a listening descriptor and a connected descriptor can be confusing when we first encounter the `accept` function. The listening descriptor serves as an endpoint for client connection requests. It is typically created once and exists for the lifetime of the server. The connected descriptor is the endpoint of the connection that is established between the client and the server. It is created each time the server accepts a connection request and exists only as long as it takes the server to service a client.

Figure 26 outlines the roles of the listening and connected descriptors. In Step 1, the server calls `accept`, which waits for a connection request to arrive on the listening descriptor, which for concreteness we will assume is descriptor 3 (recall that descriptors 0–2 are reserved for the standard files).

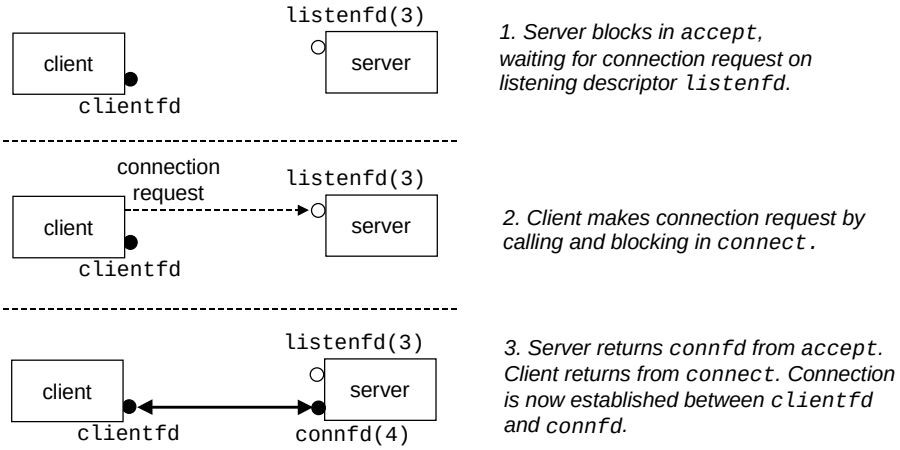

Figure 26: **The roles of the listening and connected descriptors.**

In Step 2, the client calls the `connect` function, which sends a connection request to `listenfd`. In Step 3, the `accept` function opens a new connected descriptor `connfd` (which we will assume is descriptor 4), establishes the connection between `clientfd` and `connfd`, and then returns `connfd` to the application. The client also returns from the `connect`, and from this point, the client and server can pass data back and forth by reading and writing `clientfd` and `connfd` respectively.

6.7 Example echo client and server

The best way to learn the sockets interface is to study example code. Figure 27 shows the code for an echo client. After establishing a connection with the server (line 15), the client enters a loop that repeatedly reads a text from standard input (line 17), sends the text line to the server (line 18), reads the echo line from the server (line 19), and prints the result to standard output (line 20). The loop terminates when `fgets` encounters end-of-file on standard input, either because the user typed `ctrl-d` at the keyboard, or because it has exhausted the text lines in a redirected input file.

After the loop terminates, the client closes the descriptor (line 23). This result in an end-of-file notification

../code/net/echoclient.c

```
1 #include "ics.h"
2
3 int main(int argc, char **argv)
4 {
5     int clientfd, port;
6     char *host, buf[MAXLINE];
7
8     if (argc != 3) {
9         fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
10        exit(0);
11    }
12    host = argv[1];
13    port = atoi(argv[2]);
14
15    clientfd = open_clientfd(host, port);
16
17    while (Fgets(buf, MAXLINE, stdin) != NULL) {
18        Writen(clientfd, buf, strlen(buf));
19        Readline(clientfd, buf, MAXLINE);
20        Fputs(buf, stdout);
21    }
22
23    Close(clientfd);
24    exit(0);
25 }
```

../code/net/echoclient.c

Figure 27: **Echo client main routine.**

being sent to the server, which it detects when it receives a return code of zero from its `readline` function. After closing its descriptor, the client terminates (line 24). Since the client's kernel automatically closes all open descriptors when a process terminates, the `close` in line 23 is not necessary. However, it is good programming practice to explicitly close any descriptors we have opened.

Figure 28 shows the main routine for the echo server. After opening the listening descriptor (line 18), it enters an infinite loop. Each iteration waits for a connection request from a client (line 21), prints the domain name and IP address of the connected client (lines 23-27), and calls the `echo` function that services the client (line 29). When the echo routine returns, the main routine closes the connected descriptor (line 30). Once the client and server have closed their respective descriptors, the connection is terminated.

../code/net/echoserveri.c

```
 1 #include "ics.h"
 2
 3 void echo(int connfd);
 4
 5 int main(int argc, char **argv)
 6 {
 7     int listenfd, connfd, port, clientlen;
 8     struct sockaddr_in clientaddr;
 9     struct hostent *hp;
10     char *haddrp;
11
12     if (argc != 2) {
13         fprintf(stderr, "usage: %s <port>\n", argv[0]);
14         exit(0);
15     }
16     port = atoi(argv[1]);
17
18     listenfd = open_listenfd(port);
19     while (1) {
20         clientlen = sizeof(clientaddr);
21         connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
22
23         /* determine the domain name and IP address of the client */
24         hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
25                            sizeof(clientaddr.sin_addr.s_addr), AF_INET);
26         haddrp = inet_ntoa(clientaddr.sin_addr);
27         printf("server connected to %s (%s)\n", hp->h_name, haddrp);
28
29         echo(connfd);
30         Close(connfd);
31     }
32 }
```

../code/net/echoserveri.c

Figure 28: **Iterative echo server main routine.**

Figure 29 shows the code for the `echo` routine, which repeatedly reads and writes lines of text until the `readline` function encounters end-of-file in line 8.

../code/net/echo.c

```
1 #include "ics.h"
2
3 void echo(int connfd)
4 {
5     size_t n;
6     char buf[MAXLINE];
7
8     while((n = Readline(connfd, buf, MAXLINE)) != 0) {
9         printf("server received %d bytes\n", n);
10        Writen(connfd, buf, n);
11    }
12 }
```

../code/net/echo.c

Figure 29: `echo` **function that reads and echos text lines.**

7 Concurrent servers

The echo server in Figure 28 is known as an *iterative server* because it can only service one client at a time. The disadvantage of iterative servers is that a slow client can preclude every other client from being serviced. For a real server that might be expected to service hundreds or thousands of clients per second, it is unacceptable to allow one slow client to deny service to the others.

A better approach is to build a *concurrent server* that can service multiple clients concurrently. In this section, we will investigate alternative concurrent server designs based on processes and threads.

7.1 Concurrent servers based on processes

A concurrent server based on processes accepts connection requests in the parent and forks a separate child process to service each client. For example, suppose we have two clients and a server that is listening for connection requests on a listening descriptor 2. Now suppose that the server accepts a connection request from client 1 and returns connected descriptor 4, as shown in Figure 30.

After accepting the connection request, the server forks a child, which gets a complete copy of the server's descriptor table. The child closes its copy of listening descriptor 3 and the parent closes its copy of connected descriptor 4, since they will not be needed. This gives us the situation in Figure 31, where the child process is busy servicing the client.

Now suppose that after the parent creates the child for client 1, it accepts a new connection request from client 2 and returns a new connected descriptor (say 5), as shown in Figure 32.

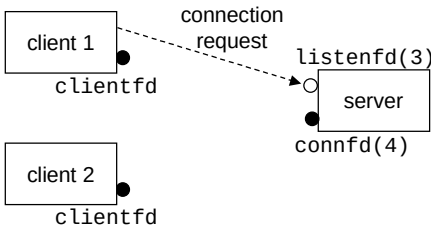

Figure 30: **Server accepts connection request from client.**

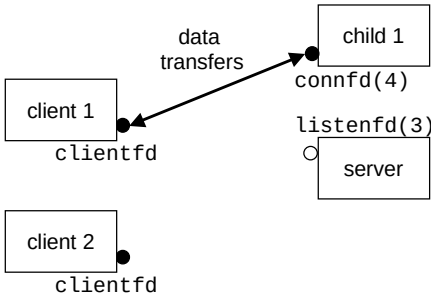

Figure 31: **Server forks a child process to service the client.**

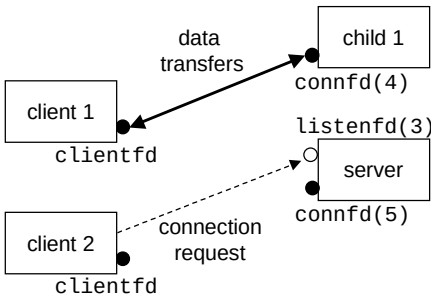

Figure 32: **Server accepts another connection request.**

The parent forks another child, which begins servicing its client using connected descriptor 5, as shown in Figure 33. At this point, the parent is waiting for the next connection request and the two children are servicing their respective clients.

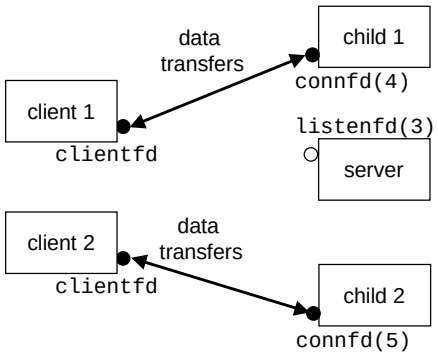

Figure 33: **Server forks another child to service the new client.**

Figure 34 shows the code for a concurrent echo server based on processes. The `echo` function in line 25 is defined in Figure 29. There are several points to make about this server.

- Since servers typically run for long periods of time, we must include a SIGCHLD handler that reaps zombie children (lines 8–14). Since SIGCHLD signals are blocked while the SIGCHLD handler is executing, and since Linux signals are not queued, the SIGCHLD handler must be prepared to reap multiple zombie children.
- Notice that the parent and the child close their respective copies of `connfd` (lines 39 and 36 respectively). This especially important for the parent, which must close its copy of the connected descriptor to avoid a memory leak that will eventually crash the system.
- Because of the reference count in the socket's file table entry (Figure 21), the socket will not be closed until both the parent's and child's copies of `connfd` are closed.

Discussion

Of the concurrent-server designs that we will study in this section, process-based designs are by far the simplest to write and debug. Processes provide a clean sharing model where descriptors are shared and user address spaces are not. As long as we remember to reap zombie children and close the parent's connected descriptor each time we create a new child, the children run independently of each other and the parent and can be debugged in isolation.

Process-based designs do have disadvantages though. If a particular service requires processes to share state information such as a memory-resident file cache, performance statistics that are aggregated across all processes, or aggregate request logs, then we must use explicit IPC mechanisms such as FIFO's, System V shared memory, or System V semaphores (none of which are discussed here). Another disadvantage is that process-based servers tend to be slower than other designs because the overhead for process control and IPC is relatively high. Nonetheless, the simplicity of process-based designs provides a powerful attraction.

../code/net/echoserverp.c

```
#include "ics.h"

void echo(int connfd);

/* SIGCHLD signal handler */
void handler(int sig)
{
    pid_t pid;
    int stat;

    while ((pid = waitpid(-1, &stat, WNOHANG)) > 0)
        ;
    return;
}

int main(int argc, char **argv)
{
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);

    Signal(SIGCHLD, handler);

    listenfd = open_listenfd(port);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* child closes its listening socket */
            echo(connfd);    /* child services client */
            Close(connfd);   /* child closes connection with client */
            exit(0);         /* child exits */
        }
        Close(connfd); /* parent closes connected socket (important!) */
    }
}
```

../code/net/echoserverp.c

Figure 34: **Concurrent echo server based on processes.**

Problem 6 [Category 1]:
After the parent closes the connected descriptor in line 39 of the concurrent server in Figure 34, the child is still able to communicate with the client using its copy of the descriptor. Why?

Problem 7 [Category 1]:
If we were to delete line 36 of Figure 34 that closes the connected descriptor, the code would still be correct, in the sense that there would be no memory leak. Why?

7.2 Concurrent servers based on threads

Another approach to building concurrent servers is to use threads instead of processes. There are several advantages to using threads. First, threads have less run time overhead than processes. We would expect a server based on threads to have better throughput (measured in clients serviced per second) than one based on processes. Second, because all threads share the same global variables and heap variables, it is much easier for threads to share state information.

The major disadvantage of using threads is that the same memory model that makes it easy to share data structures also makes it easy to share data structures unintentionally and incorrectly. As we learned in Chapter **??**, shared data must be protected, functions called from threads must be reentrant, and race conditions must be avoided.

The threaded echo server in Figure 35 illustrates some of the subtle issues that can arise. The overall structure is similar to the process-based design. The main thread repeatedly waits for a connection request (line 22) and then creates a peer thread to handle the request (line 23).

The first issue we encounter is how to pass the connected descriptor to the peer thread when we call `pthread_create`. The obvious approach is to pass a pointer to the descriptor,

```
int connfd;

connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
Pthread_create(&tid, NULL, thread, &connfd);
```

and then let the peer thread dereference the pointer and assign it to a local variable.

```
void *thread(void *vargp)
{
   int connfd = *((int *)vargp);
   ...
}
```

However, this would be wrong because it introduces a race between the assignment statement in the peer thread and the `accept` statement in the main thread. If the assignment statement completes before the next `accept`, then the local `connfd` variable in the peer thread gets the correct descriptor value. However,

../code/net/echoservert.c

```
1 #include "ics.h"
2
3 void echo_r(int connfd);
4 void *thread(void *vargp);
5
6 int main(int argc, char **argv)
7 {
8     int listenfd, *connfdp, port, clientlen;
9     struct sockaddr_in clientaddr;
10    pthread_t tid;
11
12    if (argc != 2) {
13        fprintf(stderr, "usage: %s <port>\n", argv[0]);
14        exit(0);
15    }
16    port = atoi(argv[1]);
17
18    listenfd = open_listenfd(port);
19    while (1) {
20        clientlen = sizeof(clientaddr);
21        connfdp = Malloc(sizeof(int));
22        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
23        Pthread_create(&tid, NULL, thread, connfdp);
24    }
25 }
26
27 /* thread routine */
28 void *thread(void *vargp)
29 {
30    int connfd = *((int *)vargp);
31
32    Pthread_detach(pthread_self());
33    Free(vargp);
34
35    echo_r(connfd); /* reentrant version of echo() */
36    Close(connfd);
37    return NULL;
38 }
```

../code/net/echoservert.c

Figure 35: **Concurrent echo server based on threads.**

if the assignment completes *after* the `accept`, then the local `confd` variable in the peer thread gets the descriptor number of the *next* connection. The unhappy result is two threads are now performing input and output on the same descriptor. In order to avoid the potentially deadly race, we must assign each connected descriptor returned by `accept` to its own dynamically allocated memory block, as shown in lines 21-22.

Now consider the thread routine in lines 28-38. To avoid memory leaks, we must detach the thread so that its memory resources will be reclaimed when it terminates (line 32), and we must free the memory block that was allocated by the main thread (line 33). Finally, the thread routine calls the `echo_r` function (line 35) before terminating in line 37.

So why do we call `echo_r` instead of the trusty `echo` function? The `echo` function calls the `readline` function (Figure 16, which in turn calls the `my_read` function (Figure 16), which maintains three `static` variables, and thus is not reentrant. Since `my_read` in not reentrant, neither are `readline` or `echo`.

To build a correct threaded echo server, we must use a reentrant version of `echo` called `echo_r`, which is based on the `readline_r` function, a reentrant version of the `readline` function developed by Stevens [10].

```
#include "ics.h"

ssize_t readline_r(Rline *rptr);          returns: number of bytes read (0 if EOF), -1 on error
```

The `readline` function takes as input an `Rline` structure shown in Figure 36. The first three members correspond to the arguments that users pass to `readline`. The next three members correspond to the static variables that `my_read` uses for buffering.

../code/ics.h

```
 1 typedef struct {
 2     int read_fd;        /* caller's descriptor to read from */
 3     char *read_ptr;     /* caller's buffer to read into */
 4     size_t read_maxlen; /* max bytes to read */
 5
 6     /* next three are used internally by the function */
 7     int rl_cnt;         /* initialize to 0 */
 8     char *rl_bufptr;    /* initialize to rl_buf */
 9     char rl_buf[MAXBUF];/* internal buffer */
10 } Rline;
```

../code/ics.h

Figure 36: `Rline` **structure used by** `readline_r` **and initialized by** `readline_rinit`**.**

The `Rline` structure is initialized by the `readline_rinit` function in Figure 37, which saves the user arguments and initializes the internal buffering information.

Figure 38 shows the code for the `readline_r` package. The only difference between `readline_r` and `readline` is that `readline_r` calls `my_read_r` instead of `my_read` in line 28. The `my_read_r`

../code/ics.c

```
1 void readline_rinit(int fd, void *ptr, size_t maxlen, Rline *rptr)
2 {
3     rptr->read_fd = fd;         /* save caller's arguments */
4     rptr->read_ptr = ptr;
5     rptr->read_maxlen = maxlen;
6
7     rptr->rl_cnt = 0;           /* and init our counter & pointer */
8     rptr->rl_bufptr = rptr->rl_buf;
9 }
```

../code/ics.c

Figure 37: `readline_rint`**: Initialization function for** `readline_r`**.**

function is similar to the original `my_read` function, except that it references members of the `Rline` struct instead of static variables.

Given the reentrant `readline_r` function, we can now create a reentrant version of `echo` (Figure 39) that calls `readline_r` instead of `readline`.

Discussion

Threaded designs are attractive because they promise better performance than process-based designs. But the performance gain can come with a steep price in complexity. Unlike processes, which share almost nothing, threads share almost everything. Because of this, it easy to write incorrect threaded programs that suffer from races, unprotected shared variables, and non-reentrant functions. These bugs are extremely difficult to find because they are usually non-deterministic, and thus not easily repeatable. Nothing is scarier to a programmer than a random non-repeatable bug.

The subtle issues involved in threading our simple echo server are clear evidence of the potential complexity of threaded designs. Nonetheless, if a process-based design would be unacceptably slow for a particular application, then we might need to opt for a threaded design.

8 Web servers

So far we have discussed network programming in the context of a simple echo server. In this section, we will show you how to use the basic ideas of network programming, Linux I/O, and Linux processes to build your own your small, but functional Web server.

8.1 Web basics

Web clients and servers interact using a text-based application-level protocol known as HTTP (Hypertext Transmission Protocol). HTTP is a simple protocol. A Web client (known as a *browser*) opens an Internet

../code/ics.c

```
1  static ssize_t my_read_r(Rline *rptr, char *ptr)
2  {
3      if (rptr->rl_cnt <= 0) {
4      again:
5          rptr->rl_cnt = read(rptr->read_fd, rptr->rl_buf,
6                              sizeof(rptr->rl_buf));
7          if (rptr->rl_cnt < 0) {
8              if (errno == EINTR)
9                  goto again;
10             else
11                 return(-1);
12         }
13         else if (rptr->rl_cnt == 0)
14             return(0);
15         rptr->rl_bufptr = rptr->rl_buf;
16     }
17     rptr->rl_cnt--;
18     *ptr = *rptr->rl_bufptr++ & 255;
19     return(1);
20 }
21
22 ssize_t readline_r(Rline *rptr)
23 {
24     int n, rc;
25     char c, *ptr = rptr->read_ptr;
26
27     for (n = 1; n < rptr->read_maxlen; n++) {
28         if ( (rc = my_read_r(rptr, &c)) == 1) {
29             *ptr++ = c;
30             if (c == '\n')
31                 break;
32         } else if (rc == 0) {
33             if (n == 1)
34                 return(0);      /* EOF, no data read */
35             else
36                 break;          /* EOF, some data was read */
37         } else
38             return(-1); /* error */
39     }
40     *ptr = 0;
41     return(n);
42 }
```

../code/ics.c

Figure 38: `readline_r` **package: Reentrant version of** `readline`**.** Adapted from [10].

../code/net/echo_r.c

```
1 #include "ics.h"
2
3 void echo_r(int connfd)
4 {
5     size_t n;
6     char buf[MAXLINE];
7     Rline rline;
8
9     readline_rinit(connfd, buf, MAXLINE, &rline);
10    while((n = Readline_r(&rline)) != 0) {
11        printf("server received %d bytes\n", n);
12        Writen(connfd, buf, n);
13    }
14 }
```

../code/net/echo_r.c

Figure 39: `echo_r`**: Reentrant version of** `echo`

connection to a server and requests some *content.* The server responds with the requested content and then closes the connection. The browser reads the content and displays it on the screen.

What makes the Web so different from conventional file retrieval services such as FTP? The main reason is that the content can be written in a programming language known as HTML (Hypertext Markup Language). An HTML program (page) contains instructions (tags) that tell the browser how to to display various text and graphical objects in the page. For example,

```
<b> Make me bold! </b>
```

tells the browser to print the text between the `<b>` and `</b>` tags in boldface type. However, the real power of HTML is that a page can contain pointers (hyperlinks) to content stored on remote servers anywhere in the Internet. For example,

```
<a href="http://www.cmu.edu/index.html">Carnegie Mellon</a>
```

tells the browser to highlight the text object "Carnegie Mellon" and to create a hyperlink to an HTML file called `index.html` that is stored on the CMU Web server. If the user clicks on the highlighted text object, the browser requests the corresponding HTML file from the CMU server and displays it.

8.2 Web content

To Web clients and servers, *content* is a sequence of bytes with an associated *MIME* (Multipurpose Internet Mail Extensions) type. Figure 40 shows some common MIME types.

Web servers provide content to clients in two different ways:

MIME type	Description
text/html	HTML page
text/plain	Unformatted text
application/postscript	Postscript document
image/gif	Binary image encoded in GIF format
image/jpg	Binary image encoded in JPG format

Figure 40: **Example MIME types.**

- Fetch a disk file and return its contents to the client. The disk file is known as *static content* and the process of returning the file to the client is known as *serving static content.*
- Run an executable file and return its output to the client. The output produced by the executable at runtime is known as *dynamic content*, and the process of running the program and returning its output to the client is known as *serving dynamic content.*

Thus, every piece of content returned by a Web server is associated with some file that it manages. Each of these files has a unique name known as a *URL* (Universal Resource Locator). For example, the URL

```
http://www.aol.com:80/index.html
```

identifies an HTML file called `/index.html` on Internet host `www.aol.com` that is managed by a Web server listening on port 80. The port number is optional and defaults to well-known port 80.

URLs for executable files can include program arguments after the filename. A '?' character separates the filename from the arguments, and each argument is separated by a '&' character. For example, the URL

```
http://kittyhawk.cmcl.cs.cmu.edu:8000/cgi-bin/adder?15000&213
```

identifies an executable called `/cgi-bin/adder` that will be called with two argument strings: `15000` and `213`.

Clients and servers use different parts of the URL during a transaction. For example, a client uses the prefix

```
http://www.aol.com:80
```

to determine what kind of server to contact, where the server is, and what port it is listening on. The server uses the suffix

```
/index.html
```

to find the file on its filesystem, and to determine whether the request is for static or dynamic content. There are several important points to understand about how servers interpret the suffix of a URL:

- There are no standard rules for determining whether a URL refers to static or dynamic content. Each server has its own rules for the files that it manages. A common approach is to identify a set of directories, such as `cgi-bin`, where all executables must reside.

- The initial '/' in the suffix does *not* denote the Linux root directory. Rather is denotes the home directory for whatever kind of content is being requested. For example, a server might by configured so that all static content is stored in directory `/usr/httpd/html` and all dynamic content is stored in directory `/usr/httpd/cgi-bin`.
- The minimal URL suffix is the '/' character, which all servers expand to some default home page such as `/index.html`. This explains why it is possible to fetch the home page of a site by simply typing a domain name to the browser. The browser appends the missing '/' to the URL and passes it to the server, which expands the '/' to some default file name.

8.3 HTTP transactions

Since HTTP is is based on text lines transmitted over Internet connections, we can use the Linux `telnet` program to conduct transactions with any Web server on the Internet.[7] For example, Figure 41 uses `telnet` to request the home page from the AOL Web server.

```
 1 linux> telnet www.aol.com 80          Client: open connection to server
 2 Trying 205.188.146.23...              Telnet prints 3 lines to the terminal
 3 Connected to aol.com.
 4 Escape character is '^]'.
 5 GET / HTTP/1.1                        Client: request line
 6 host: www.aol.com                     Client: required HTTP/1.1 header
 7                                       Client: empty line terminates headers.
 8 HTTP/1.0 200 OK                       Server: response line
 9 MIME-Version: 1.0                     Server: followed by five response headers
10 Date: Mon, 08 Jan 2001 04:59:42 GMT
11 Server: NaviServer/2.0 AOLserver/2.3.3
12 Content-Type: text/html               Server: expect HTML in the response body
13 Content-Length: 42092                 Server: expect 42,092 bytes in the response body
14                                       Server: empty line terminates response headers
15 <html>                                Server: first HTML line in response body
16 ...                                   Server: 766 lines of HTML not shown.
17 </html>                               Server: last HTML line in response body
18 Connection closed by foreign host.    Server: closes connection
19 linux>                                Client: closes connection and terminates
```

Figure 41: **An HTTP transaction that serves static content.**

In line 1 we run `telnet` from a Linux shell and ask it to open a connection to the AOL Web server. `Telnet` prints three lines of output to the terminal, opens the connection, and then waits for us to enter text (line 5). Each time we enter a text line and hit the `enter` key, `telnet` reads the line, appends carriage return and line feed characters (`"\r\n"` in C notation), and sends the line to the server. This is consistent

[7] `telnet` is a handy tool for debugging servers that talk to clients with text lines over connections.

with the HTTP standard, which requires every text line to be terminated by a carriage return and line feed pair. To initiate the transaction, we enter an HTTP request (lines 5-7). The server replies with an HTTP response (lines 8-17) and then closes the connection (line 18).

HTTP requests

An HTTP request consists of a *request line* (line 5), followed by zero or more *request headers* (line 6), followed by an empty text line that terminates the list of headers (line 7). A request line has the form

```
<method> <uri> <version>
```

HTTP supports a number of different *methods*, including GET, POST, OPTIONS, HEAD, PUT, DELETE, and TRACE). We will only discuss the workhorse GET method, which according to one study accounts for over 99% of HTTP requests [8]. The GET method instructs the server to generate and return the content identified by the *URI* (Uniform Resource Identifier). The URI is the suffix of the corresponding URL that includes the file name and optional arguments. [8]

The `<version>` field in the request line indicates the HTTP version that the request conforms to. The current version is HTTP/1.1 [2]. HTTP/1.0 is a previous version from 1996 that is still in use [1]. HTTP/1.1 defines additional headers that provide support for advanced features such as caching and security, as well as a (seldom used) mechanism that allows a client and server to perform multiple transactions over the same *persistent connection.* In practice, the two versions are compatible because HTTP/1.0 clients and servers simply ignore unknown HTTP/1.1 headers.

In sum, the request line in line 5 asks the server to fetch and return the HTML file `/index.html`. It also informs the server that the remainder of the request will be in HTTP/1.1 format.

Request headers provide additional information to the server, such as the brand name of the browser or the MIME types that the browser understands. Request headers have the form

```
<header name>: <header data>
```

For our purposes, the only header we need to be concerned with is the `Host` header (line 5), which is required in HTTP/1.1 requests, but not in HTTP/1.0 requests. The `Host` header is only used by *proxy caches*, which sometimes serve as intermediaries between a browser and the *origin server* that manages the requested file. Multiple proxies can exist between a client and an origin server in a so-called *proxy chain.* The data in the `Host` header, which identifies the domain name of the origin server, allows a proxy in the middle of a proxy chain to determine if it might have a locally cached copy of the requested content.

Continuing with our example in Figure 41, the empty text line in line 6 (generated by hitting `enter` on our keyboard) terminates the headers and instructs the server to send the requested HTML file.

[8] Actually, this is only true when a browser requests content. If a proxy server requests content, then the URI must be the complete URL.

HTTP responses

HTTP responses are similar to HTTP requests. An HTTP response consists of a *response line* (line 8), followed by zero or more *response headers* (lines 9-13), followed by an empty line that terminates the headers (line 14), followed by the *response body* (lines 15-17).

A response line has the form

```
<version>  <status code> <status message>
```

The version field describes the HTTP version that the response conforms to. The *status code* is a 3-digit positive integer that indicates the disposition of the request. The *status message* gives the English equivalent of the error code. Figure 42 lists some common status codes and their corresponding messages.

Status code	Status Message	Description
200	OK	Request was handled without error.
301	Moved permanently	Content has moved to the hostname in the Location header.
400	Bad request	Request could not be understood by the server.
403	Forbidden	Server lacks permission to access the requested file.
404	Not found	Server could not find the requested file.
501	Not implemented	Server does not support the request method.
505	HTTP version not supported	Server does not support version in request.

Figure 42: **Some HTTP status codes.**

The response headers in lines 9-13 provide additional information about the response. The two most important headers are `Content-Type` (line 12), which tells the client the MIME type of the content in the response body, and `Content-Length` (line 13), which indicates its size in byte.

The empty text line in line 11 that terminates the response headers is followed by the request body, which contains the requested content.

8.4 Serving dynamic content

If we stop to think for a moment how a server might provide dynamic content to a client, certain questions arise. For example, how does the client pass any program arguments to the server? How does the server pass these arguments to the child process that it creates? How does the server pass other information to the child that it might need to generate the content? Where does the child send its output? These questions are addressed by a de facto standard called CGI (Common Gateway Interface).

How does the client pass program arguments to the server?

Arguments for GET requests are passed in the URI.[9] Each argument is separated by a '&' character. Spaces are not allowed in arguments and must be denoted with the `%20` string. Similar encodings exist for other special characters.

[9] Arguments for POST requests are passed in the request body rather than the URI.

How does the server pass arguments to the child?

After a server receives a request such as

```
GET /cgi-bin/adder?15000&213 HTTP/1.1
```

it calls `fork` to creates a child process and calls `execve` to run the `/cgi-bin/adder` program in the context of the child. The `adder` program is often referred to as *CGI program* because it obeys the rules of the CGI standard. And since many CGI programs are written as Perl scripts, CGI programs are often called *CGI scripts.*

Before the call to `execve`, the child process sets the CGI environment variable QUERY_STRING to `15000&213`, which the `adder` program can reference at runtime using the Linux `getenv` function.

How does the server pass other information to the child?

CGI defines a number of other environment variables that a CGI program can expect to be set when it runs. Figure 43 shows a subset.

Environment variable	Description
SERVER_PORT	Port that the parent is listening on
REQUEST_METHOD	GET or POST
REMOTE_HOST	Domain name of client
REMOTE_ADDR	Dotted-decimal IP address of client
CONTENT_TYPE	POST only: MIME type of the request body
CONTENT_LENGTH	POST only: Size in bytes of the request body

Figure 43: **Examples of CGI environment variables.**

Where does the child send its output?

A CGI program prints dynamic content to the standard output. Before the child process loads and runs the CGI program, it uses the Linux `dup2` function to redirect standard output to the connected descriptor that is associated with the client. Thus, anything that the CGI program writes to standard output goes directly to the client.[10]

Notice that since the parent does not know the type or size of the content that the child generates, the child is responsible for generating the `Content-type` and `Content-length` response headers, as well as the empty line that terminates the headers.

Figure 44 shows a simple CGI program that sums its two arguments and returns an HTML file with the result to the client. Figure 45 shows an HTTP transaction that serves dynamic content from the `adder` program.

[10]For POST requests, the child would also redirect standard input to the connected descriptor and the CGI program would read the arguments in the request body from standard input.

../code/net/tiny/cgi-bin/adder.c

```
#include "ics.h"

int main(void) {
    char *buf, *p;
    char arg1[MAXLINE], arg2[MAXLINE], content[MAXLINE];
    int n1=0, n2=0;

    /* extract the two arguments */
    if ((buf = getenv("QUERY_STRING")) != NULL) {
        p = strchr(buf, '&');
        *p = '\0';
        strcpy(arg1, buf);
        strcpy(arg2, p+1);
        n1 = atoi(arg1);
        n2 = atoi(arg2);
    }

    /* make the response body */
    sprintf(content, "Welcome to add.com: ");
    sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
    sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
            content, n1, n2, n1 + n2);
    sprintf(content, "%sThanks for visiting!\r\n", content);

    /* generate the HTTP response */
    printf("Content-length: %d\r\n", strlen(content));
    printf("Content-type: text/html\r\n\r\n");
    printf("%s", content);
    fflush(stdout);
    exit(0);
}
```

../code/net/tiny/cgi-bin/adder.c

Figure 44: **CGI program that sums two integers.**

```
1 kittyhawk> telnet kittyhawk.cmcl.cs.cmu.edu 8000   Client: open connection
2 Trying 128.2.194.242...
3 Connected to kittyhawk.cmcl.cs.cmu.edu.
4 Escape character is '^]'.
5 GET /cgi-bin/adder?15000&213 HTTP/1.0   Client: request line
6                                         Client: empty line terminates headers
7 HTTP/1.0 200 OK                         Server: response line
8 Server: Tiny Web Server                 Server: identify server
9 Content-length: 115                     Adder: expect 115 bytes in response body
10 Content-type: text/html                Adder: expect HTML in response body
11                                        Adder: empty line terminates headers
12 Welcome to add.com: THE Internet addition portal.  Adder: first HTML line
13 <p>The answer is: 15000 + 213 = 15213  Adder: second HTML line in response body
14 <p>Thanks for visiting!                Adder: third HTML line in response body
15 Connection closed by foreign host.     Server: closes connection
16 kittyhawk>                             Client: closes connection and terminates
```

Figure 45: **An HTTP transaction that serves dynamic HTML content.**

Problem 8 [Category 1]:
In Section 5.8, we warned about the dangers of using the C standard I/O functions in servers. Yet the CGI program in Figure 44 is able to use standard I/O without any problems. Why?

8.5 The Tiny Web server

We will conclude our discussion of network programming by developing a small but functioning Web server called *Tiny*. Tiny is an interesting program. It combines many of the ideas that we have learned about concurrency, Linux I/O, the sockets interface, and HTTP in only 250 lines of code. While it lacks the functionality, robustness, and security of a real server, it is powerful enough to serve both static and dynamic content to real Web browsers. We encourage you to study it and implement it yourself. It is quite exciting (even for the authors!) to point a real browser at your own server and watch it display a complicated Web page with text and graphics.

The Tiny `main` routine

Figure 46 shows Tiny's main routine. Tiny is an iterative server that listens for connection requests on the port that is passed in the command line. After opening a listening socket (line 28) by calling the `open_listenfd` function from Figure 46, Tiny executes the typical infinite server loop, repeatedly accepting a connection request (line 31) and performing a transaction (line 32).

../code/net/tiny/tiny.c

```
/*
 * tiny.c - A simple HTTP/1.0 Web server that uses the GET method
 *          to serve static and dynamic content.
 */
#include "ics.h"

void doit(int fd);
void read_requesthdrs(int fd);
int parse_uri(char *uri, char *filename, char *cgiargs);
void serve_static(int fd, char *filename, int filesize);
void get_filetype(char *filename, char *filetype);
void serve_dynamic(int fd, char *filename, char *cgiargs);
void clienterror(int fd, char *cause, char *errnum,
                 char *shortmsg, char *longmsg);

int main(int argc, char **argv)
{
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;

    /* check command line args */
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(1);
    }
    port = atoi(argv[1]);

    listenfd = open_listenfd(port);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        doit(connfd);
        Close(connfd);
    }
}
```

../code/net/tiny/tiny.c

Figure 46: **The Tiny Web server.**

The `doit` function

The `doit` function in Figure 47 handles one HTTP transaction. First, we read and parse the request line (lines 9-10). Notice that we are using the robust `readline` function from Figure 16 to read the request line.

Tiny only supports the GET method. If the client requests another method (such as POST), we send it an error message and return to the main routine (lines 11-15), which then closes the connection and awaits the next connection request. Otherwise, we read and (as we shall see) ignore any request headers (line 16).

Next, we parse the URI into a filename and a possibly empty CGI argument string, and we set a flag that indicates whether the request is for static or dynamic content (line 19). If the file does not exist on disk, we immediately send an error message to the client and return (lines 20-24).

Finally, if the request is for static content (lines 26), we verify that the file is a regular file (i.e., not a directory file or a FIFO) and that we have read permission (line 27). If so, we serve the static content (line 32) to the client. Similarly, if the request is for dynamic content (line 34), we verify that the file is executable (line 35), and if so we go ahead and serve the dynamic content (line 40).

The `clienterror` function

Tiny lacks many of the robustness features of a real server. However it does check for some obvious errors and reports them to the client. The `clienterror` function in Figure 48 sends an HTTP response to the client with the appropriate status code and status message in the response line, along with an HTML file in the response body that explains the error to the browser's user.

Recall that an HTML response should indicate the size and type the content in the body. Thus, we have opted to build the HTML content as a single string (lines 7-11) so that we can easily determine its size (line 18). Also, notice that we are using the robust `writen` function from Figure 15 for all output.

The `read_requesthdrs` function

Tiny does not use any of the information in the request headers. It simply reads and ignores them by calling the `read_requesthdrs` function in Figure 49. Notice that the empty text line that terminates the request headers consists of a carriage return and line feed pair, which we check for in line 6.

The `parse_uri` function

Tiny assumes that the home directory for static content is the current Linux directory `'.'`, and that the home directory for executables is `./cgi-bin`. Any URI that contains the string `cgi-bin` is assumed to denote a request for dynamic content. The default file is `./index.html`.

The `parse_uri` function in Figure 50 implements these policies. It parses the URI into a filename and an optional CGI argument string. If the request is for static content (line 5) we clear the CGI argument string (line 6), and then convert the URI into a relative Linux pathname such as `./index.html` (lines 7-8). If the URI ends with a `'/'` character (line 9), then we append the default file name (lines 9). On the other

../code/net/tiny/tiny.c

```
1  void doit(int fd)
2  {
3      int is_static;
4      struct stat sbuf;
5      char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
6      char filename[MAXLINE], cgiargs[MAXLINE];
7
8      /* read request line and headers */
9      Readline(fd, buf, MAXLINE);
10     sscanf(buf, "%s %s %s\n", method, uri, version);
11     if (strcasecmp(method, "GET")) {
12        clienterror(fd, method, "501", "Not Implemented",
13                 "Tiny does not implement this method");
14         return;
15     }
16     read_requesthdrs(fd);
17
18     /* parse URI from GET request */
19     is_static = parse_uri(uri, filename, cgiargs);
20     if (stat(filename, &sbuf) < 0) {
21         clienterror(fd, filename, "404", "Not found",
22                     "Tiny couldn't find this file");
23         return;
24     }
25
26     if (is_static) { /* serve static content */
27         if (!(S_ISREG(sbuf.st_mode)) || !(S_IRUSR & sbuf.st_mode)) {
28             clienterror(fd, filename, "403", "Forbidden",
29                         "Tiny couldn't read the file");
30             return;
31         }
32         serve_static(fd, filename, sbuf.st_size);
33     }
34     else { /* serve dynamic content */
35         if (!(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode)) {
36             clienterror(fd, filename, "403", "Forbidden",
37                         "Tiny couldn't run the CGI program");
38             return;
39         }
40         serve_dynamic(fd, filename, cgiargs);
41     }
42 }
```

../code/net/tiny/tiny.c

Figure 47: **Tiny `doit`: Handles one HTTP transaction.**

../code/net/tiny/tiny.c

```
1 void clienterror(int fd, char *cause, char *errnum,
2                  char *shortmsg, char *longmsg)
3 {
4     char buf[MAXLINE], body[MAXBUF];
5
6     /* build the HTTP response body */
7     sprintf(body, "<html><title>Tiny Error</title>");
8     sprintf(body, "%s<body bgcolor=""ffffff"">\r\n", body);
9     sprintf(body, "%s%s: %s\r\n", body, errnum, shortmsg);
10    sprintf(body, "%s<p>%s: %s\r\n", body, longmsg, cause);
11    sprintf(body, "%s<hr><em>The Tiny Web server</em>\r\n", body);
12
13    /* print the HTTP response */
14    sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
15    Writen(fd, buf, strlen(buf));
16    sprintf(buf, "Content-type: text/html\r\n");
17    Writen(fd, buf, strlen(buf));
18    sprintf(buf, "Content-length: %d\r\n\r\n", strlen(body));
19    Writen(fd, buf, strlen(buf));
20    Writen(fd, body, strlen(body));
21 }
```

../code/net/tiny/tiny.c

Figure 48: **Tiny `clienterror`: Sends an error message to the client.**

../code/net/tiny/tiny.c

```
1 void read_requesthdrs(int fd)
2 {
3     char buf[MAXLINE];
4
5     Readline(fd, buf, MAXLINE);
6     while(strcmp(buf, "\r\n"))
7         Readline(fd, buf, MAXLINE);
8     return;
9 }
```

../code/net/tiny/tiny.c

Figure 49: **Tiny `read_requesthdrs`: Reads and ignores request headers.**

hand, if the request is for dynamic content (line 13), we extract any CGI arguments (line 14-20) and convert the remaining portion of the URI to a relative Linux file name (lines 21-22).

../code/net/tiny/tiny.c

```
1  int parse_uri(char *uri, char *filename, char *cgiargs)
2  {
3      char *ptr;
4
5      if (!strstr(uri, "cgi-bin")) {  /* static content */
6          strcpy(cgiargs, "");
7          strcpy(filename, ".");
8          strcat(filename, uri);
9          if (uri[strlen(uri)-1] == '/')
10             strcat(filename, "index.html");
11         return 1;
12     }
13     else {  /* dynamic content */
14         ptr = index(uri, '?');
15         if (ptr) {
16             strcpy(cgiargs, ptr+1);
17             *ptr = '\0';
18         }
19         else
20             strcpy(cgiargs, "");
21         strcpy(filename, ".");
22         strcat(filename, uri);
23         return 0;
24     }
25 }
```

../code/net/tiny/tiny.c

Figure 50: **Tiny `parse_uri`: Parses an HTTP URI.**

The `serve_static` function

Tiny serves 4 different types of static content: HTML files, unformatted text files, and images encoded in GIF and JPG formats. These file types account for the majority of static content served over the Web.

The `serve_static` function in Figure 51 sends an HTTP response whose body contains the contents of a local file. First, we determine the file type by inspecting the suffix in the filename (line 7), and then send the response line and response headers to the client (lines 6-12). Notice that we are using the `writen` function from Figure 15 for all output on the descriptor. Notice also that a blank line terminates the headers (line 12).

Next, we send the response body by copying the contents of the requested file to the connected descriptor `fd` (lines 15-19). The code here is somewhat subtle and needs to be studied carefully.

../code/net/tiny/tiny.c

```
1  void serve_static(int fd, char *filename, int filesize)
2  {
3      int srcfd;
4      char *srcp, filetype[MAXLINE], buf[MAXBUF];
5
6      /* send response headers to client */
7      get_filetype(filename, filetype);
8      sprintf(buf, "HTTP/1.0 200 OK\r\n");
9      sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
10     sprintf(buf, "%sContent-length: %d\n", buf, filesize);
11     sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
12     Writen(fd, buf, strlen(buf));
13
14     /* send response body to client */
15     srcfd = Open(filename, O_RDONLY, 0);
16     srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
17     Close(srcfd);
18     Writen(fd, srcp, filesize);
19     Munmap(srcp, filesize);
20 }
21
22 /*
23  * get_filetype - derive file type from file name
24  */
25 void get_filetype(char *filename, char *filetype)
26 {
27     if (strstr(filename, ".html"))
28         strcpy(filetype, "text/html");
29     else if (strstr(filename, ".gif"))
30         strcpy(filetype, "image/gif");
31     else if (strstr(filename, ".jpg"))
32         strcpy(filetype, "image/jpg");
33     else
34         strcpy(filetype, "text/plain");
35 }
```

../code/net/tiny/tiny.c

Figure 51: **Tiny `serve_static`: Serves static content to a client.**

Line 15 opens `filename` for reading and gets its descriptor. In line 16, the Linux `mmap` function maps the requested file to a virtual memory area. Recall from our discussion of `mmap` in Section **??** that the call to `mmap` maps the first `filesize` bytes of file `srcfd` to a private read-only area of virtual memory that starts at address `srcp`.

Once we have mapped the file to memory, we no longer need its descriptor, so we close the file (line 17). Failing to do this would introduce a potentially fatal memory leak.

Line 18 performs the actual transfer of the file to the client. The `writen` function copies the `filesize` bytes starting at location `srcp` (which of course is mapped to the requested file) to the client's connected descriptor. Finally, line 19 frees the mapped virtual memory area. This is important to avoid a potentially fatal memory leak.

The `serve_dynamic` function

Tiny serves any type of dynamic content by forking a child process, and then running a CGI program in the context of the child.

The `serve_dynamic` function in Figure 52 begins by sending a response line indicating success to the client (lines 6-7), along with an informational `Server` header (lines 8-9). The CGI program is responsible for sending the rest of the response. Notice that this is not as robust as we might wish, since it doesn't allow for the possibility that the CGI program might encounter some error.

../code/net/tiny/tiny.c

```
1  void serve_dynamic(int fd, char *filename, char *cgiargs)
2  {
3      char buf[MAXLINE];
4
5      /* return first part of HTTP response */
6      sprintf(buf, "HTTP/1.0 200 OK\r\n");
7      Writen(fd, buf, strlen(buf));
8      sprintf(buf, "Server: Tiny Web Server\r\n");
9      Writen(fd, buf, strlen(buf));
10
11     if (Fork() == 0) { /* child */
12         /* real server would set all CGI vars here */
13         setenv("QUERY_STRING", cgiargs, 1);
14         Dup2(fd, STDOUT_FILENO);         /* redirect output to client */
15         Execve(filename, NULL, environ); /* run CGI program */
16     }
17     Wait(NULL); /* parent reaps child */
18 }
```

../code/net/tiny/tiny.c

Figure 52: **Tiny `serve_dynamic`: Serves dynamic content to a client.**

After sending the first part of the response, we fork a new child process (line 11). The child initializes the

QUERY_STRING environment variable with the CGI arguments from the request URI (line 13). Notice that a real server would set the other CGI environment variables here as well. For brevity, we have omitted this step.

Next, the child redirects the child's standard output to the connected file descriptor (line 13), and then loads and runs the CGI program (line 14). Since the CGI program runs in the context of the child, it has access to the same open descriptors and environment variables that existed before the call to the `execve` function. Thus, everything that the CGI program writes to standard output goes directly to the client process, without any intervention from the parent process.

Meanwhile, the parent blocks in a call to `wait`, waiting to reap the child when it terminates (line 17).

Problem 9 [Category 1]:

A. Is the Tiny `doit` routine reentrant? Why or why not?

B. If not, how would you make it reentrant?

Problem 10 [Category 2]:

A. Modify Tiny so that it echos every request line and request header.

B. Use your favorite browser to make a request to Tiny for static content. Capture the output from Tiny in a file.

C. Inspect the output from Tiny to determine the the version of HTTP your browser uses.

D. Consult the HTTP/1.1 standard in RFC 2616 to determine the meaning of each header in the HTTP request from your browser. You can obtain RFC 2616 from `www.rfc-editor.org/rfc.html`.

Problem 11 [Category 2]:
Extend Tiny to so that it serves MPG video files. Check your work using a real browser.

Problem 12 [Category 2]:
Modify Tiny so that its reaps CGI children inside a SIGCHLD handler instead of explicitly waiting for them to terminate.

Problem 13 [Category 2]:
Modify Tiny so that when it serves static content, it copies the requested file to the connected descriptor using `malloc`, `read`, and `write`, instead of `mmap` and `write`.

Problem 14 [Category 2]:

A. Write an HTML form for the CGI `adder` function in Figure 44. Your form should include two text boxes that users will fill in with the two numbers they want to add together. Your form should also request content using the GET method.

B. Check your work by using a real browser to request the form from Tiny, submit the filled in form to Tiny, and then display the the dynamic content generated by `adder`.

Problem 15 [Category 2]:
Extend Tiny to support the HTTP HEAD method. Check your work using `telnet` as a Web client.

Problem 16 [Category 3]:
Extend Tiny so that it serves dynamic contest requested by the HTTP POST method. Check your work using your favorite Web browser.

Problem 17 [Category 3]:
Build a concurrent Tiny server based on processes.

Problem 18 [Category 3]:
Build a concurrent Tiny server based on threads.

9 Summary

In this chapter we have learned some basic concepts about network applications. Network applications use the client-server model, where servers perform services on behalf of their clients. The Internet provides network applications with two key mechanisms: (1) A unique name for each Internet host, and (2) a mechanism for establishing a connection to a server running on any of those hosts. Clients and servers establish connections by using the sockets interface, and they communicate over these connections using standard Linux file I/O functions.

There a two basic design options for servers. An iterative server handles one request at a time. A concurrent server can handle multiple requests concurrently. We investigated two designs for concurrent servers, one that forks a new process for each request, the other that creates a new thread for each request. Other designs are possible, such as using the Linux `select` function to explicitly manage the concurrency, or avoiding the per-connection overhead by pre-forking a set of child processes to handle connection requests.

Finally, we studied the design and implementation of a simple but functional Web server. In a few lines of code, it ties together many important systems concepts such as Linux I/O, memory mapping, concurrency, the sockets interface, and the HTTP protocol.

Bibliographic notes

The official source information for the Internet is contained in a set of freely-available numbered documents known as *RFCs* (Requests for Comments). A searchable index of RFCs is available from

```
http://www.rfc-editor.org/rfc.html
```

RFCs are typically written for developers of Internet infrastructure, and thus are usually too detailed for the casual reader. However, for authoritative information, there is no better source.

There are many good texts that cover basic concepts of computer networking [3, 4, 11].

The great technical writer W. Richard Stevens developed a whole series of classic texts on such topics as advanced Unix programming [5], the Internet protocols [6, 7, 8], and Unix network programming [10, 9]. Serious students of Linux systems programming will want to study all of them. Tragically, Stevens died in 1999. His contributions will be greatly missed.

The authoritative list of MIME types is maintained at

```
ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types
```

The HTTP/1.1 protocol is documented in RFC 2616.

References

[1] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol - HTTP/1.0. RFC 1945, 1996.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol - HTTP/1.1. RFC 2616, 1999.

[3] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet.* Addison-Wesley, 2000.

[4] L. Peterson and B. Davies. *Computer Networks: A Systems Approach, Third Edition.* Morgan Kaufmann, 1999.

[5] W. Richard Stevens. *Advanced Programming in the Unix Environment.* Addison-Wesley, 1992.

[6] W. Richard Stevens. *TCP/IP Illustrated: The Protocols*, volume 1. Addison Wesley, 1994.

[7] W. Richard Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley, 1995.

[8] W. Richard Stevens. *TCP/IP Illustrated: TCP for Transactions, HTTP, NNTP and the Unix domain protocols*, volume 3. Addison-Wesley, 1996.

[9] W. Richard Stevens. *Unix Network Programming: Interprocess Communications (Second Edition)*, volume 2. Prentice Hall, 1998.

[10] W. Richard Stevens. *Unix Network Programming: Networking APIs (Second Edition)*, volume 1. Prentice Hall, 1998.

[11] A. Tannenbaum. *Computer Networks, Third Edition.* Prentice-Hall, 1996.

Problem solutions

Problem 1 Solution:

Hex address	Dotted decimal address
0x0	0.0.0.0
0xffffffff	255.255.255.255
0x7f000001	127.0.0.1
0xcdbca079	205.188.160.121
0x400c950d	64.12.149.13
0xcdbc9217	205.188.146.23

Problem 2 Solution:

../code/net/hex2dd.c

```
 1 #include "ics.h"
 2
 3 int main(int argc, char **argv)
 4 {
 5     struct in_addr inaddr;  /* addr in network byte order */
 6     unsigned int   addr;    /* addr in host byte order */
 7
 8     if (argc != 2) {
 9         fprintf(stderr, "usage: %s <hex number>\n", argv[0]);
10         exit(0);
11     }
12     sscanf(argv[1], "%x", &addr);
13     inaddr.s_addr = htonl(addr);
14     printf("%s\n", inet_ntoa(inaddr));
15
16     exit(0);
17 }
```

../code/net/hex2dd.c

Problem 3 Solution:

../code/net/dd2hex.c

```
 1 #include "ics.h"
 2
 3 int main(int argc, char **argv)
 4 {
 5     struct in_addr inaddr;  /* addr in network byte order */
 6     unsigned int   addr;    /* addr in host byte order */
 7
 8     if (argc != 2) {
 9         fprintf(stderr, "usage: %s <dotted-decimal>\n", argv[0]);
10         exit(0);
11     }
12
13     if (inet_aton(argv[1], &inaddr) == 0)
14         app_error("inet_aton error");
15     addr = ntohl(inaddr.s_addr);
16     printf("0x%x\n", addr);
17
18     exit(0);
19 }
```

../code/net/dd2hex.c

Problem 4 Solution:
Each time we request the host entry for `aol.com`, the list of corresponding Internet addresses is returned in a different, round-robin order.

```
kittyhawk> hostinfo aol.com
official hostname: aol.com
address: 205.188.146.23
address: 205.188.160.121
address: 64.12.149.13

kittyhawk> hostinfo aol.com
official hostname: aol.com
address: 64.12.149.13
address: 205.188.146.23
address: 205.188.160.121

kittyhawk> hostinfo aol.com
official hostname: aol.com
address: 205.188.146.23
address: 205.188.160.121
address: 64.12.149.13
```

The different ordering of the addresses in different DNS queries is known as *DNS round-robin*. It can be used to load-balance requests to a heavily used domain name.

Problem 5 Solution:

../code/net/cpstdinbuf.c

```
 1 #include "ics.h"
 2
 3 int main(void)
 4 {
 5     char buf[MAXBUF];
 6     int n;
 7
 8     /* copy stdin to stdout MAXBUF bytes at a time */
 9     while((n = Readn(STDIN_FILENO, buf, MAXBUF)) != 0)
10         Writen(STDOUT_FILENO, buf, n);
11     exit(0);
12 }
```

../code/net/cpstdinbuf.c

Problem 6 Solution:
When the parent the forks the child, it gets a copy of the connected descriptor and the reference count for the associated file table in incremented from 1 to 2. When the parent closes its copy of the descriptor, the reference count is decremented from 2 to 1. Since the kernel will not close a file until the reference counter in its file table goes to zero, the child's end of the connection stays open.

Problem 7 Solution:
When a process terminates for any reason, the kernel closes all open descriptors. Thus, the child's copy of the connected file descriptor will be closed automatically when the child exits.

Problem 8 Solution:
The reason that standard I/O works in CGI programs is that we never have to explicitly close the standard input and output streams. When the child exits, the kernel will close streams and their associated file descriptors automatically.

Problem 9 Solution:

A. The `doit` function is not reentrant, because it and its subfunctions use the non-reentrant `readline` function.

B. To make Tiny reentrant, we must replace all calls to `readline` with its reentrant counterpart `readline_r`, being carefull to call `readline_rinit` in `doit` before the first call to `readline_r`.

Problem 10 Solution:
There is no unique solution. The problem has several purposes. First, we want to make sure you can compile and run Tiny. Second, we want you to see what a real browser request looks like and what the information contained in it means.

Problem 11 Solution:
Solution outline: This sounds like it might be difficult, but it is really very simple. To a Web server, all content is just a stream of bytes. Simply add the MIME type `video/mpg` (CHECK THIS!) to the `get_filetype` function in Figure 51.

Problem 12 Solution:
Solution outline: Install a SIGCHLD handler in the main routine and delete the call to `wait` in `serve_dynamic`.

Problem 13 Solution:
Solution outline: Allocate a buffer, read the requested file into the buffer, write the buffer to the descriptor, and then free the buffer.

Problem 14 Solution:
No solution yet.

Problem 15 Solution:
Solution outline: HEAD is identical to GET, except that it does not return the response body.

Problem 16 Solution:
No solution yet.

Problem 17 Solution:
Solution outline: The general approach is identical to the the concurrent echo server in Figure 34. Replace the call to `echo` with a call to `doit`.

Problem 18 Solution:
Solution outline: The general approach is identical to the the threaded echo server in Figure 35. As with the threaded echo echo, the non-reentrant `readline` function must be replaced by its reentrant `readline_r` counterpart.