16

Network IPC: Sockets

16.1 Introduction

In the previous chapter, we looked at pipes, FIFOs, message queues, semaphores, and shared memory—the classical methods of IPC provided by various UNIX systems. These mechanisms allow processes running on the same computer to communicate with one another. In this chapter, we look at the mechanisms that allow processes running on different computers (connected to a common network) to communicate with one another—network IPC.

In this chapter, we describe the socket network IPC interface, which can be used by processes to communicate with other processes, regardless of where they are running—on the same machine or on different machines. Indeed, this was one of the design goals of the socket interface. The same interfaces can be used for both intermachine communication and intramachine communication. Although the socket interface can be used to communicate using many different network protocols, we will restrict our discussion to the TCP/IP protocol suite in this chapter, since it is the de facto standard for communicating over the Internet.

The socket API as specified by POSIX.1 is based on the 4.4BSD socket interface. Although minor changes have been made over the years, the current socket interface closely resembles the interface when it was originally introduced in 4.2BSD in the early 1980s.

This chapter is only an overview of the socket API. Stevens, Fenner, and Rudoff [2004] discuss the socket interface in detail in the definitive text on network programming in the UNIX System.

16.2 Socket Descriptors

A socket is an abstraction of a communication endpoint. Just as they would use file descriptors to access files, applications use socket descriptors to access sockets. Socket descriptors are implemented as file descriptors in the UNIX System. Indeed, many of the functions that deal with file descriptors, such as `read` and `write`, will work with a socket descriptor.

To create a socket, we call the `socket` function.

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

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

Returns: file (socket) descriptor if OK, –1 on error

The *domain* argument determines the nature of the communication, including the address format (described in more detail in the next section). Figure 16.1 summarizes the domains specified by POSIX.1. The constants start with `AF_` (for *address family*) because each domain has its own format for representing an address.

Domain	Description
`AF_INET`	IPv4 Internet domain
`AF_INET6`	IPv6 Internet domain (optional in POSIX.1)
`AF_UNIX`	UNIX domain
`AF_UNSPEC`	unspecified

Figure 16.1 Socket communication domains

We discuss the UNIX domain in Section 17.2. Most systems define the `AF_LOCAL` domain also, which is an alias for `AF_UNIX`. The `AF_UNSPEC` domain is a wildcard that represents "any" domain. Historically, some platforms provide support for additional network protocols, such as `AF_IPX` for the NetWare protocol family, but domain constants for these protocols are not defined by the POSIX.1 standard.

The *type* argument determines the type of the socket, which further determines the communication characteristics. The socket types defined by POSIX.1 are summarized in Figure 16.2, but implementations are free to add support for additional types.

Type	Description
`SOCK_DGRAM`	fixed-length, connectionless, unreliable messages
`SOCK_RAW`	datagram interface to IP (optional in POSIX.1)
`SOCK_SEQPACKET`	fixed-length, sequenced, reliable, connection-oriented messages
`SOCK_STREAM`	sequenced, reliable, bidirectional, connection-oriented byte streams

Figure 16.2 Socket types

The *protocol* argument is usually zero, to select the default protocol for the given domain and socket type. When multiple protocols are supported for the same domain and socket type, we can use the *protocol* argument to select a particular protocol. The default protocol for a `SOCK_STREAM` socket in the `AF_INET` communication domain is

TCP (Transmission Control Protocol). The default protocol for a SOCK_DGRAM socket in the AF_INET communication domain is UDP (User Datagram Protocol). Figure 16.3 lists the protocols defined for the Internet domain sockets.

Protocol	Description
IPPROTO_IP	IPv4 Internet Protocol
IPPROTO_IPV6	IPv6 Internet Protocol (optional in POSIX.1)
IPPROTO_ICMP	Internet Control Message Protocol
IPPROTO_RAW	Raw IP packets protocol (optional in POSIX.1)
IPPROTO_TCP	Transmission Control Protocol
IPPROTO_UDP	User Datagram Protocol

Figure 16.3 Protocols defined for Internet domain sockets

With a datagram (SOCK_DGRAM) interface, no logical connection needs to exist between peers for them to communicate. All you need to do is send a message addressed to the socket being used by the peer process.

A datagram, therefore, provides a connectionless service. A byte stream (SOCK_STREAM), in contrast, requires that, before you can exchange data, you set up a logical connection between your socket and the socket belonging to the peer with which you wish to communicate.

A datagram is a self-contained message. Sending a datagram is analogous to mailing someone a letter. You can mail many letters, but you can't guarantee the order of delivery, and some might get lost along the way. Each letter contains the address of the recipient, making the letter independent from all the others. Each letter can even go to different recipients.

In contrast, using a connection-oriented protocol for communicating with a peer is like making a phone call. First, you need to establish a connection by placing a phone call, but after the connection is in place, you can communicate bidirectionally with each other. The connection is a peer-to-peer communication channel over which you talk. Your words contain no addressing information, as a point-to-point virtual connection exists between both ends of the call, and the connection itself implies a particular source and destination.

A SOCK_STREAM socket provides a byte-stream service; applications are unaware of message boundaries. This means that when we read data from a SOCK_STREAM socket, it might not return the same number of bytes written by the sender. We will eventually get everything sent to us, but it might take several function calls.

A SOCK_SEQPACKET socket is just like a SOCK_STREAM socket except that we get a message-based service instead of a byte-stream service. This means that the amount of data received from a SOCK_SEQPACKET socket is the same amount as was written. The Stream Control Transmission Protocol (SCTP) provides a sequential packet service in the Internet domain.

A SOCK_RAW socket provides a datagram interface directly to the underlying network layer (which means IP in the Internet domain). Applications are responsible for building their own protocol headers when using this interface, because the transport protocols (TCP and UDP, for example) are bypassed. Superuser privileges are required to create a raw socket to prevent malicious applications from creating packets that might bypass established security mechanisms.

Calling `socket` is similar to calling `open`. In both cases, you get a file descriptor that can be used for I/O. When you are done using the file descriptor, you call `close` to relinquish access to the file or socket and free up the file descriptor for reuse.

Although a socket descriptor is actually a file descriptor, you can't use a socket descriptor with every function that accepts a file descriptor argument. Figure 16.4 summarizes most of the functions we've described so far that are used with file descriptors and describes how they behave when used with socket descriptors. Unspecified and implementation-defined behavior usually means that the function doesn't work with socket descriptors. For example, `lseek` doesn't work with sockets, since sockets don't support the concept of a file offset.

Function	Behavior with socket
`close` (Section 3.3)	deallocates the socket
`dup`, `dup2` (Section 3.12)	duplicates the file descriptor as normal
`fchdir` (Section 4.23)	fails with `errno` set to `ENOTDIR`
`fchmod` (Section 4.9)	unspecified
`fchown` (Section 4.11)	implementation defined
`fcntl` (Section 3.14)	some commands supported, including `F_DUPFD`, `F_DUPFD_CLOEXEC`, `F_GETFD`, `F_GETFL`, `F_GETOWN`, `F_SETFD`, `F_SETFL`, and `F_SETOWN`
`fdatasync`, `fsync` (Section 3.13)	implementation defined
`fstat` (Section 4.2)	some `stat` structure members supported, but how left up to the implementation
`ftruncate` (Section 4.13)	unspecified
`ioctl` (Section 3.15)	some commands work, depending on underlying device driver
`lseek` (Section 3.6)	implementation defined (usually fails with `errno` set to `ESPIPE`)
`mmap` (Section 14.8)	unspecified
`poll` (Section 14.4.2)	works as expected
`pread` and `pwrite` (Section 3.11)	fails with `errno` set to `ESPIPE`
`read` (Section 3.7) and `readv` (Section 14.6)	equivalent to `recv` (Section 16.5) without any flags
`select` (Section 14.4.1)	works as expected
`write` (Section 3.8) and `writev` (Section 14.6)	equivalent to `send` (Section 16.5) without any flags

Figure 16.4 How file descriptor functions act with sockets

Communication on a socket is bidirectional. We can disable I/O on a socket with the `shutdown` function.

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

int shutdown(int sockfd, int how);

                                        Returns: 0 if OK, –1 on error
```

If *how* is `SHUT_RD`, then reading from the socket is disabled. If *how* is `SHUT_WR`, then we can't use the socket for transmitting data. We can use `SHUT_RDWR` to disable both data transmission and reception.

Given that we can `close` a socket, why is `shutdown` needed? There are several reasons. First, `close` will deallocate the network endpoint only when the last active reference is closed. If we duplicate the socket (with `dup`, for example), the socket won't be deallocated until we close the last file descriptor referring to it. The `shutdown` function allows us to deactivate a socket independently of the number of active file descriptors referencing it. Second, it is sometimes convenient to shut a socket down in one direction only. For example, we can shut a socket down for writing if we want the process we are communicating with to be able to tell when we are done transmitting data, while still allowing us to use the socket to receive data sent to us by the process.

16.3 Addressing

In the previous section, we learned how to create and destroy a socket. Before we learn to do something useful with a socket, we need to learn how to identify the process with which we wish to communicate. Identifying the process has two components. The machine's network address helps us identify the computer on the network we wish to contact, and the service, represented by a *port number*, helps us identify the particular process on the computer.

16.3.1 Byte Ordering

When communicating with processes running on the same computer, we generally don't have to worry about byte ordering. The byte order is a characteristic of the processor architecture, dictating how bytes are ordered within larger data types, such as integers. Figure 16.5 shows how the bytes within a 32-bit integer are numbered.

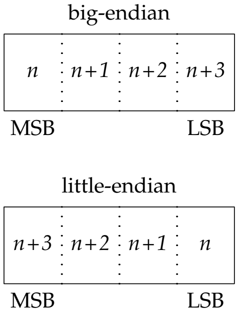

Figure 16.5 Byte order in a 32-bit integer

If the processor architecture supports *big-endian* byte order, then the highest byte address occurs in the least significant byte (LSB). *Little-endian* byte order is the opposite: the least significant byte contains the lowest byte address. Note that regardless of the byte ordering, the most significant byte (MSB) is always on the left, and the least significant byte is always on the right. Thus, if we were to assign a 32-bit integer the value `0x04030201`, the most significant byte would contain 4, and the least significant byte would contain 1, regardless of the byte ordering. If we were then to cast a character pointer (cp) to the address of the integer, we would see a difference from the byte ordering. On a little-endian processor, cp[0] would refer to the least significant byte and contain 1; cp[3] would refer to the most significant byte and contain 4. Compare that to a big-endian processor, where cp[0] would contain 4, referring to the most significant byte, and cp[3] would contain 1, referring to the least significant byte. Figure 16.6 summarizes the byte ordering for the four platforms discussed in this text.

Operating system	Processor architecture	Byte order
FreeBSD 8.0	Intel Pentium	little-endian
Linux 3.2.0	Intel Core i5	little-endian
Mac OS X 10.6.8	Intel Core 2 Duo	little-endian
Solaris 10	Sun SPARC	big-endian

Figure 16.6 Byte order for test platforms

> To confuse matters further, some processors can be configured for either little-endian or big-endian operation.

Network protocols specify a byte ordering so that heterogeneous computer systems can exchange protocol information without confusing the byte ordering. The TCP/IP protocol suite uses big-endian byte order. The byte ordering becomes visible to applications when they exchange formatted data. With TCP/IP, addresses are presented in network byte order, so applications sometimes need to translate them between the processor's byte order and the network byte order. This is common when printing an address in a human-readable form, for example.

Four functions are provided to convert between the processor byte order and the network byte order for TCP/IP applications.

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

uint32_t htonl(uint32_t hostint32);

                              Returns: 32-bit integer in network byte order

uint16_t htons(uint16_t hostint16);

                              Returns: 16-bit integer in network byte order

uint32_t ntohl(uint32_t netint32);

                                 Returns: 32-bit integer in host byte order

uint16_t ntohs(uint16_t netint16);

                                 Returns: 16-bit integer in host byte order
```

The h is for "host" byte order, and the n is for "network" byte order. The l is for "long" (i.e., 4-byte) integer, and the s is for "short" (i.e., 2-byte) integer. Although we include <arpa/inet.h> to use these functions, system implementations often declare these functions in other headers that are included by <arpa/inet.h>. It is also common for systems to implement these functions as macros.

16.3.2 Address Formats

An address identifies a socket endpoint in a particular communication domain. The address format is specific to the particular domain. So that addresses with different formats can be passed to the socket functions, the addresses are cast to a generic sockaddr address structure:

```
struct sockaddr {
  sa_family_t   sa_family;  /* address family */
  char          sa_data[];  /* variable-length address */
  ⋮
};
```

Implementations are free to add more members and define a size for the sa_data member. For example, on Linux, the structure is defined as

```
struct sockaddr {
  sa_family_t  sa_family;    /* address family */
  char         sa_data[14];  /* variable-length address */
};
```

But on FreeBSD, the structure is defined as

```
struct sockaddr {
  unsigned char  sa_len;       /* total length */
  sa_family_t    sa_family;    /* address family */
  char           sa_data[14];  /* variable-length address */
};
```

Internet addresses are defined in <netinet/in.h>. In the IPv4 Internet domain (AF_INET), a socket address is represented by a sockaddr_in structure:

```
struct in_addr {
  in_addr_t       s_addr;        /* IPv4 address */
};

struct sockaddr_in {
  sa_family_t    sin_family;    /* address family */
  in_port_t      sin_port;      /* port number */
  struct in_addr sin_addr;      /* IPv4 address */
};
```

The in_port_t data type is defined to be a uint16_t. The in_addr_t data type is defined to be a uint32_t. These integer data types specify the number of bits in the data type and are defined in <stdint.h>.

In contrast to the AF_INET domain, the IPv6 Internet domain (AF_INET6) socket address is represented by a sockaddr_in6 structure:

```
struct in6_addr {
  uint8_t        s6_addr[16];    /* IPv6 address */
};
```

```
struct sockaddr_in6 {
  sa_family_t     sin6_family;    /* address family */
  in_port_t       sin6_port;      /* port number */
  uint32_t        sin6_flowinfo;  /* traffic class and flow info */
  struct in6_addr sin6_addr;      /* IPv6 address */
  uint32_t        sin6_scope_id;  /* set of interfaces for scope */
};
```

These are the definitions required by the Single UNIX Specification. Individual implementations are free to add more fields. For example, on Linux, the `sockaddr_in` structure is defined as

```
struct sockaddr_in {
  sa_family_t     sin_family;    /* address family */
  in_port_t       sin_port;      /* port number */
  struct in_addr  sin_addr;      /* IPv4 address */
  unsigned char   sin_zero[8];   /* filler */
};
```

where the `sin_zero` member is a filler field that should be set to all-zero values.

Note that although the `sockaddr_in` and `sockaddr_in6` structures are quite different, they are both passed to the socket routines cast to a `sockaddr` structure. In Section 17.2, we will see that the structure of a UNIX domain socket address is different from both of the Internet domain socket address formats.

It is sometimes necessary to print an address in a format that is understandable by a person instead of a computer. The BSD networking software included the `inet_addr` and `inet_ntoa` functions to convert between the binary address format and a string in dotted-decimal notation (a.b.c.d). These functions, however, work only with IPv4 addresses. Two new functions—`inet_ntop` and `inet_pton`—support similar functionality and work with both IPv4 and IPv6 addresses.

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

const char *inet_ntop(int domain, const void *restrict addr,
                      char *restrict str, socklen_t size);
```

Returns: pointer to address string on success, NULL on error

```
int inet_pton(int domain, const char *restrict str,
              void *restrict addr);
```

Returns: 1 on success, 0 if the format is invalid, or −1 on error

The `inet_ntop` function converts a binary address in network byte order into a text string; `inet_pton` converts a text string into a binary address in network byte order. Only two *domain* values are supported: `AF_INET` and `AF_INET6`.

For `inet_ntop`, the *size* parameter specifies the size of the buffer (*str*) to hold the text string. Two constants are defined to make our job easier: `INET_ADDRSTRLEN` is large enough to hold a text string representing an IPv4 address, and `INET6_ADDRSTRLEN` is large enough to hold a text string representing an IPv6 address. For `inet_pton`, the *addr* buffer needs to be large enough to hold a 32-bit address if *domain* is `AF_INET` or large enough to hold a 128-bit address if *domain* is `AF_INET6`.

16.3.3 Address Lookup

Ideally, an application won't have to be aware of the internal structure of a socket address. If an application simply passes socket addresses around as `sockaddr` structures and doesn't rely on any protocol-specific features, then the application will work with many different protocols that provide the same type of service.

Historically, the BSD networking software has provided interfaces to access the various network configuration information. In Section 6.7, we briefly discussed the networking data files and the functions used to access them. In this section, we discuss them in a little more detail and introduce the newer functions used to look up addressing information.

The network configuration information returned by these functions can be kept in a number of places. This information can be kept in static files (e.g., `/etc/hosts`, `/etc/services`), or it can be managed by a name service, such as DNS (Domain Name System) or NIS (Network Information Service). Regardless of where the information is kept, the same functions can be used to access it.

The hosts known by a given computer system are found by calling `gethostent`.

```
#include <netdb.h>

struct hostent *gethostent(void);

                                        Returns: pointer if OK, NULL on error

void sethostent(int stayopen);

void endhostent(void);
```

If the host database file isn't already open, `gethostent` will open it. The `gethostent` function returns the next entry in the file. The `sethostent` function will open the file or rewind it if it is already open. When the *stayopen* argument is set to a nonzero value, the file remains open after calling `gethostent`. The `endhostent` function can be used to close the file.

When `gethostent` returns, we get a pointer to a `hostent` structure, which might point to a static data buffer that is overwritten each time we call `gethostent`. The `hostent` structure is defined to have at least the following members:

```
struct hostent {
  char   *h_name;       /* name of host */
  char  **h_aliases;    /* pointer to alternate host name array */
  int     h_addrtype;   /* address type */
  int     h_length;     /* length in bytes of address */
  char  **h_addr_list;  /* pointer to array of network addresses */
   .
   .
   .
};
```

The addresses returned are in network byte order.

Two additional functions—`gethostbyname` and `gethostbyaddr`—originally were included with the `hostent` functions, but are now considered to be obsolete. They were removed from Version 4 of the Single UNIX Specification. We'll see replacements for them shortly.

We can get network names and numbers with a similar set of interfaces.

```
#include <netdb.h>

struct netent *getnetbyaddr(uint32_t net, int type);

struct netent *getnetbyname(const char *name);

struct netent *getnetent(void);

                              All return: pointer if OK, NULL on error

void setnetent(int stayopen);

void endnetent(void);
```

The netent structure contains at least the following fields:

```
struct netent {
  char     *n_name;      /* network name */
  char    **n_aliases;   /* alternate network name array pointer */
  int       n_addrtype;  /* address type */
  uint32_t  n_net;       /* network number */
  ⋮
};
```

The network number is returned in network byte order. The address type is one of the address family constants (AF_INET, for example).

We can map between protocol names and numbers with the following functions.

```
#include <netdb.h>

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotoent(void);

                              All return: pointer if OK, NULL on error

void setprotoent(int stayopen);

void endprotoent(void);
```

The protoent structure as defined by POSIX.1 has at least the following members:

```
struct protoent {
  char   *p_name;     /* protocol name */
  char  **p_aliases;  /* pointer to alternate protocol name array */
  int     p_proto;    /* protocol number */
  ⋮
};
```

Services are represented by the port number portion of the address. Each service is offered on a unique, well-known port number. We can map a service name to a port number with `getservbyname`, map a port number to a service name with `getservbyport`, or scan the services database sequentially with `getservent`.

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

struct servent *getservent(void);

                              All return: pointer if OK, NULL on error

void setservent(int stayopen);

void endservent(void);
```

The `servent` structure is defined to have at least the following members:

```
struct servent {
  char   *s_name;     /* service name */
  char  **s_aliases;  /* pointer to alternate service name array */
  int     s_port;     /* port number */
  char   *s_proto;    /* name of protocol */
  ⋮
};
```

POSIX.1 defines several new functions to allow an application to map from a host name and a service name to an address, and vice versa. These functions replace the older `gethostbyname` and `gethostbyaddr` functions.

The `getaddrinfo` function allows us to map a host name and a service name to an address.

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

int getaddrinfo(const char *restrict host,
                const char *restrict service,
                const struct addrinfo *restrict hint,
                struct addrinfo **restrict res);

                        Returns: 0 if OK, nonzero error code on error

void freeaddrinfo(struct addrinfo *ai);
```

We need to provide the host name, the service name, or both. If we provide only one name, the other should be a null pointer. The host name can be either a node name or the host address in dotted-decimal notation.

The `getaddrinfo` function returns a linked list of `addrinfo` structures. We can use `freeaddrinfo` to free one or more of these structures, depending on how many structures are linked together using the `ai_next` field in the structures.

The `addrinfo` structure is defined to include at least the following members:

```
struct addrinfo {
  int              ai_flags;      /* customize behavior */
  int              ai_family;     /* address family */
  int              ai_socktype;   /* socket type */
  int              ai_protocol;   /* protocol */
  socklen_t        ai_addrlen;    /* length in bytes of address */
  struct sockaddr *ai_addr;       /* address */
  char            *ai_canonname;  /* canonical name of host */
  struct addrinfo *ai_next;       /* next in list */
  ⋮
};
```

We can supply an optional *hint* to select addresses that meet certain criteria. The hint is a template used for filtering addresses and uses only the `ai_family`, `ai_flags`, `ai_protocol`, and `ai_socktype` fields. The remaining integer fields must be set to 0, and the pointer fields must be null. Figure 16.7 summarizes the flags we can use in the `ai_flags` field to customize how addresses and names are treated.

Flag	Description
AI_ADDRCONFIG	Query for whichever address type (IPv4 or IPv6) is configured.
AI_ALL	Look for both IPv4 and IPv6 addresses (used only with AI_V4MAPPED).
AI_CANONNAME	Request a canonical name (as opposed to an alias).
AI_NUMERICHOST	The host address is specified in numeric format; don't try to translate it.
AI_NUMERICSERV	The service is specified as a numeric port number; don't try to translate it.
AI_PASSIVE	Socket address is intended to be bound for listening.
AI_V4MAPPED	If no IPv6 addresses are found, return IPv4 addresses mapped in IPv6 format.

Figure 16.7 Flags for `addrinfo` structure

If `getaddrinfo` fails, we can't use `perror` or `strerror` to generate an error message. Instead, we need to call `gai_strerror` to convert the error code returned into an error message.

```
#include <netdb.h>

const char *gai_strerror(int error);
```

Returns: a pointer to a string describing the error

The `getnameinfo` function converts an address into host and service names.

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

int getnameinfo(const struct sockaddr *restrict addr, socklen_t alen,
                char *restrict host, socklen_t hostlen,
                char *restrict service, socklen_t servlen, int flags);
```

Returns: 0 if OK, nonzero on error

The socket address (*addr*) is translated into a host name and a service name. If *host* is non-null, it points to a buffer *hostlen* bytes long that will be used to return the host name. Similarly, if *service* is non-null, it points to a buffer *servlen* bytes long that will be used to return the service name.

The *flags* argument gives us some control over how the translation is done. Figure 16.8 summarizes the supported flags.

Flag	Description
NI_DGRAM	The service is datagram based instead of stream based.
NI_NAMEREQD	If the host name can't be found, treat this as an error.
NI_NOFQDN	Return only the node name portion of the fully qualified domain name for local hosts.
NI_NUMERICHOST	Return the numeric form of the host address instead of the name.
NI_NUMERICSCOPE	For IPv6, return the numeric form of the scope ID instead of the name.
NI_NUMERICSERV	Return the numeric form of the service address (i.e., the port number) instead of the name.

Figure 16.8 Flags for the `getnameinfo` function

Example

Figure 16.9 illustrates the use of the `getaddrinfo` function.

```
#include "apue.h"
#if defined(SOLARIS)
#include <netinet/in.h>
#endif
#include <netdb.h>
#include <arpa/inet.h>
#if defined(BSD)
#include <sys/socket.h>
#include <netinet/in.h>
#endif

void
print_family(struct addrinfo *aip)
{
    printf(" family ");
    switch (aip->ai_family) {
    case AF_INET:
        printf("inet");
        break;
    case AF_INET6:
        printf("inet6");
        break;
    case AF_UNIX:
        printf("unix");
        break;
    case AF_UNSPEC:
        printf("unspecified");
        break;
    default:
        printf("unknown");
    }
}

void
print_type(struct addrinfo *aip)
{
    printf(" type ");
    switch (aip->ai_socktype) {
    case SOCK_STREAM:
        printf("stream");
        break;
    case SOCK_DGRAM:
        printf("datagram");
        break;
    case SOCK_SEQPACKET:
        printf("seqpacket");
        break;
    case SOCK_RAW:
        printf("raw");
        break;
    default:
        printf("unknown (%d)", aip->ai_socktype);
    }
}

void
print_protocol(struct addrinfo *aip)
{
    printf(" protocol ");
    switch (aip->ai_protocol) {
    case 0:
        printf("default");
        break;
    case IPPROTO_TCP:
        printf("TCP");
        break;
    case IPPROTO_UDP:
        printf("UDP");
        break;
    case IPPROTO_RAW:
        printf("raw");
        break;
    default:
        printf("unknown (%d)", aip->ai_protocol);
    }
}
```

```
void
print_flags(struct addrinfo *aip)
{
    printf("flags");
    if (aip->ai_flags == 0) {
        printf(" 0");
    } else {
        if (aip->ai_flags & AI_PASSIVE)
            printf(" passive");
        if (aip->ai_flags & AI_CANONNAME)
            printf(" canon");
        if (aip->ai_flags & AI_NUMERICHOST)
            printf(" numhost");
        if (aip->ai_flags & AI_NUMERICSERV)
            printf(" numserv");
        if (aip->ai_flags & AI_V4MAPPED)
            printf(" v4mapped");
        if (aip->ai_flags & AI_ALL)
            printf(" all");
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo     *ailist, *aip;
    struct addrinfo     hint;
    struct sockaddr_in  *sinp;
    const char          *addr;
    int                 err;
    char                abuf[INET_ADDRSTRLEN];

    if (argc != 3)
        err_quit("usage: %s nodename service", argv[0]);
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = 0;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], argv[2], &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        print_flags(aip);
        print_family(aip);
        print_type(aip);
        print_protocol(aip);
        printf("\n\thost %s", aip->ai_canonname?aip->ai_canonname:"-");
        if (aip->ai_family == AF_INET) {
            sinp = (struct sockaddr_in *)aip->ai_addr;
            addr = inet_ntop(AF_INET, &sinp->sin_addr, abuf,
                INET_ADDRSTRLEN);
            printf(" address %s", addr?addr:"unknown");
            printf(" port %d", ntohs(sinp->sin_port));
        }
        printf("\n");
    }
    exit(0);
}
```

Figure 16.9 Print host and service information

This program illustrates the use of the getaddrinfo function. If multiple protocols provide the given service for the given host, the program will print more than one entry. In this example, we print out the address information only for the protocols that work with IPv4 (ai_family equals AF_INET). If we wanted to restrict the output to the AF_INET protocol family, we could set the ai_family field in the hint.

When we run the program on one of the test systems, we get

```
$ ./a.out harry nfs
flags canon family inet type stream protocol TCP
     host harry address 192.168.1.99 port 2049
flags canon family inet type datagram protocol UDP
     host harry address 192.168.1.99 port 2049
```

□

16.3.4 Associating Addresses with Sockets

The address associated with a client's socket is of little interest, and we can let the system choose a default address for us. For a server, however, we need to associate a well-known address with the server's socket on which client requests will arrive. Clients need a way to discover the address to use to contact a server, and the simplest scheme is for a server to reserve an address and register it in /etc/services or with a name service.

We use the bind function to associate an address with a socket.

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Returns: 0 if OK, −1 on error

There are several restrictions on the address we can use:

- The address we specify must be valid for the machine on which the process is running; we can't specify an address belonging to some other machine.
- The address must match the format supported by the address family we used to create the socket.

- The port number in the address cannot be less than 1,024 unless the process has the appropriate privilege (i.e., is the superuser).
- Usually, only one socket endpoint can be bound to a given address, although some protocols allow duplicate bindings.

For the Internet domain, if we specify the special IP address INADDR_ANY (defined in <netinet/in.h>), the socket endpoint will be bound to all the system's network interfaces. This means that we can receive packets from any of the network interface cards installed in the system. We'll see in the next section that the system will choose an address and bind it to our socket for us if we call connect or listen without first binding an address to the socket.

We can use the getsockname function to discover the address bound to a socket.

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

int getsockname(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);

                                          Returns: 0 if OK, –1 on error
```

Before calling getsockname, we set *alenp* to point to an integer containing the size of the sockaddr buffer. On return, the integer is set to the size of the address returned. If the address won't fit in the buffer provided, the address is silently truncated. If no address is currently bound to the socket, the results are undefined.

If the socket is connected to a peer, we can find out the peer's address by calling the getpeername function.

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

int getpeername(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);

                                          Returns: 0 if OK, –1 on error
```

Other than returning the peer's address, the getpeername function is identical to the getsockname function.

16.4 Connection Establishment

If we're dealing with a connection-oriented network service (SOCK_STREAM or SOCK_SEQPACKET), then before we can exchange data, we need to create a connection between the socket of the process requesting the service (the client) and the process providing the service (the server). We use the connect function to create a connection.

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

int connect(int sockfd, const struct sockaddr *addr, socklen_t len);

                                          Returns: 0 if OK, –1 on error
```

The address we specify with connect is the address of the server with which we wish to communicate. If *sockfd* is not bound to an address, connect will bind a default address for the caller.

When we try to connect to a server, the connect request might fail for several reasons. For a connect request to succeed, the machine to which we are trying to connect must be up and running, the server must be bound to the address we are trying to contact, and there must be room in the server's pending connect queue (we'll learn more about this shortly). Thus, applications must be able to handle connect error returns that might be caused by transient conditions.

Example

Figure 16.10 shows one way to handle transient connect errors. These errors are likely with a server that is running on a heavily loaded system.

```c
#include "apue.h"
#include <sys/socket.h>

#define MAXSLEEP 128

int
connect_retry(int sockfd, const struct sockaddr *addr, socklen_t alen)
{
    int numsec;

    /*
     * Try to connect with exponential backoff.
     */
    for (numsec = 1; numsec <= MAXSLEEP; numsec <<= 1) {
        if (connect(sockfd, addr, alen) == 0) {
            /*
             * Connection accepted.
             */
            return(0);
        }

        /*
         * Delay before trying again.
         */
        if (numsec <= MAXSLEEP/2)
            sleep(numsec);
    }
    return(-1);
}
```

Figure 16.10 Connect with retry

This function shows what is known as an *exponential backoff* algorithm. If the call to connect fails, the process goes to sleep for a short time and then tries again, increasing the delay each time through the loop, up to a maximum delay of about 2 minutes.

There is a problem with the code shown in Figure 16.10: it isn't portable. This technique works on Linux and Solaris, but doesn't work as expected on FreeBSD and Mac OS X. If the first connection attempt fails, BSD-based socket implementations continue to fail successive connection attempts when the same socket descriptor is used with TCP. This is a case of a protocol-specific behavior leaking through the (protocol-independent) socket interface and becoming visible to applications. The reason for this is historical, and thus the Single UNIX Specification warns that the state of a socket is undefined if `connect` fails.

Because of this, portable applications need to close the socket if `connect` fails. If we want to retry, we have to open a new socket. This more portable technique is shown in Figure 16.11.

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

#define MAXSLEEP 128

int
connect_retry(int domain, int type, int protocol,
              const struct sockaddr *addr, socklen_t alen)
{
    int numsec, fd;

    /*
     * Try to connect with exponential backoff.
     */
    for (numsec = 1; numsec <= MAXSLEEP; numsec <<= 1) {
        if ((fd = socket(domain, type, protocol)) < 0)
            return(-1);
        if (connect(fd, addr, alen) == 0) {
            /*
             * Connection accepted.
             */
            return(fd);
        }
        close(fd);

        /*
         * Delay before trying again.
         */
        if (numsec <= MAXSLEEP/2)
            sleep(numsec);
    }
    return(-1);
}
```

Figure 16.11 Portable connect with retry

Note that because we might have to establish a new socket, it makes no sense to pass a socket descriptor to the `connect_retry` function. Instead of returning an indication of success, we now return a connected socket descriptor to the caller. □

If the socket descriptor is in nonblocking mode, which we discuss further in Section 16.8, connect will return −1 with errno set to the special error code EINPROGRESS if the connection can't be established immediately. The application can use either poll or select to determine when the file descriptor is writable. At this point, the connection is complete.

The connect function can also be used with a connectionless network service (SOCK_DGRAM). This might seem like a contradiction, but it is an optimization instead. If we call connect with a SOCK_DGRAM socket, the destination address of all messages we send is set to the address we specified in the connect call, relieving us from having to provide the address every time we transmit a message. In addition, we will receive datagrams only from the address we've specified.

A server announces that it is willing to accept connect requests by calling the listen function.

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

int listen(int sockfd, int backlog);

                                        Returns: 0 if OK, −1 on error
```

The *backlog* argument provides a hint to the system regarding the number of outstanding connect requests that it should enqueue on behalf of the process. The actual value is determined by the system, but the upper limit is specified as SOMAXCONN in <sys/socket.h>.

> On Solaris, the SOMAXCONN value in <sys/socket.h> is ignored. The particular maximum depends on the implementation of each protocol. For TCP, the default is 128.

Once the queue is full, the system will reject additional connect requests, so the *backlog* value must be chosen based on the expected load of the server and the amount of processing it must do to accept a connect request and start the service.

Once a server has called listen, the socket used can receive connect requests. We use the accept function to retrieve a connect request and convert it into a connection.

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

int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict len);

                     Returns: file (socket) descriptor if OK, −1 on error
```

The file descriptor returned by accept is a socket descriptor that is connected to the client that called connect. This new socket descriptor has the same socket type and address family as the original socket (*sockfd*). The original socket passed to accept is not associated with the connection, but instead remains available to receive additional connect requests.

If we don't care about the client's identity, we can set the *addr* and *len* parameters to NULL. Otherwise, before calling accept, we need to set the *addr* parameter to a buffer large enough to hold the address and set the integer pointed to by *len* to the size of the buffer in bytes. On return, accept will fill in the client's address in the buffer and update the integer pointed to by *len* to reflect the size of the address.

If no connect requests are pending, accept will block until one arrives. If *sockfd* is in nonblocking mode, accept will return −1 and set errno to either EAGAIN or EWOULDBLOCK.

> All four platforms discussed in this text define EAGAIN to be the same as EWOULDBLOCK.

If a server calls accept and no connect request is present, the server will block until one arrives. Alternatively, a server can use either poll or select to wait for a connect request to arrive. In this case, a socket with pending connect requests will appear to be readable.

Example

Figure 16.12 shows a function we can use to allocate and initialize a socket for use by a server process.

```
#include "apue.h"
#include <errno.h>
#include <sys/socket.h>

int
initserver(int type, const struct sockaddr *addr, socklen_t alen,
  int qlen)
{
    int fd;
    int err = 0;

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (bind(fd, addr, alen) < 0)
        goto errout;
    if (type == SOCK_STREAM || type == SOCK_SEQPACKET) {
        if (listen(fd, qlen) < 0)
            goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(-1);
}
```

Figure 16.12 Initialize a socket endpoint for use by a server

We'll see that TCP has some strange rules regarding address reuse that make this example inadequate. Figure 16.22 shows a version of this function that bypasses these rules, solving the major drawback with this version. □

16.5 Data Transfer

Since a socket endpoint is represented as a file descriptor, we can use `read` and `write` to communicate with a socket, as long as it is connected. Recall that a datagram socket can be "connected" if we set the default peer address using the `connect` function. Using `read` and `write` with socket descriptors is significant, because it means that we can pass socket descriptors to functions that were originally designed to work with local files. We can also arrange to pass the socket descriptors to child processes that execute programs that know nothing about sockets.

Although we can exchange data using `read` and `write`, that is about all we can do with these two functions. If we want to specify options, receive packets from multiple clients, or send out-of-band data, we need to use one of the six socket functions designed for data transfer.

Three functions are available for sending data, and three are available for receiving data. First, we'll look at the ones used to send data.

The simplest one is `send`. It is similar to `write`, but allows us to specify flags to change how the data we want to transmit is treated.

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

ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
```

Returns: number of bytes sent if OK, –1 on error

Like `write`, the socket has to be connected to use `send`. The *buf* and *nbytes* arguments have the same meaning as they do with `write`.

Unlike `write`, however, `send` supports a fourth *flags* argument. Three flags are defined by the Single UNIX Specification, but it is common for implementations to support additional ones. They are summarized in Figure 16.13.

If `send` returns success, it doesn't necessarily mean that the process at the other end of the connection receives the data. All we are guaranteed is that when `send` succeeds, the data has been delivered to the network drivers without error.

With a protocol that supports message boundaries, if we try to send a single message larger than the maximum supported by the protocol, `send` will fail with `errno` set to `EMSGSIZE`. With a byte-stream protocol, `send` will block until the entire amount of data has been transmitted.

The `sendto` function is similar to `send`. The difference is that `sendto` allows us to specify a destination address to be used with connectionless sockets.

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

ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags,
               const struct sockaddr *destaddr, socklen_t destlen);
```

Returns: number of bytes sent if OK, –1 on error

With a connection-oriented socket, the destination address is ignored, as the destination is implied by the connection. With a connectionless socket, we can't use `send` unless the destination address is first set by calling `connect`, so `sendto` gives us an alternate way to send a message.

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`MSG_CONFIRM`	Provide feedback to the link layer to keep address mapping valid.			•		
`MSG_DONTROUTE`	Don't route packet outside of local network.		•	•	•	•
`MSG_DONTWAIT`	Enable nonblocking operation (equivalent to using `O_NONBLOCK`).		•	•	•	•
`MSG_EOF`	Shut the sender side of the socket down after sending data.		•		•	
`MSG_EOR`	Mark the end of the record if supported by protocol.	•	•	•	•	•
`MSG_MORE`	Delay sending the packet to allow more data to be written.			•		
`MSG_NOSIGNAL`	Don't generate `SIGPIPE` when writing to an unconnected socket.	•	•	•		
`MSG_OOB`	Send out-of-band data if supported by protocol (see Section 16.7).	•	•	•	•	•

Figure 16.13 Flags used with `send` socket calls

We have one more choice when transmitting data over a socket. We can call `sendmsg` with a `msghdr` structure to specify multiple buffers from which to transmit data, similar to the `writev` function (Section 14.6).

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

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Returns: number of bytes sent if OK, −1 on error

POSIX.1 defines the `msghdr` structure to have at least the following members:

```
struct msghdr {
  void          *msg_name;        /* optional address */
  socklen_t      msg_namelen;     /* address size in bytes */
  struct iovec  *msg_iov;         /* array of I/O buffers */
  int            msg_iovlen;      /* number of elements in array */
  void          *msg_control;     /* ancillary data */
  socklen_t      msg_controllen;  /* number of ancillary bytes */
  int            msg_flags;       /* flags for received message */
  ⋮
};
```

We saw the `iovec` structure in Section 14.6. We'll see the use of ancillary data in Section 17.4.

The recv function is similar to read, but allows us to specify some options to control how we receive the data.

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

ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

Returns: length of message in bytes,
0 if no messages are available and peer has done an orderly shutdown,
or −1 on error

The flags that can be passed to recv are summarized in Figure 16.14. Only three are defined by the Single UNIX Specification.

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
MSG_CMSG_CLOEXEC	Set the close-on-exec flag for file descriptors received over a UNIX domain socket (see Section 17.4).			•		
MSG_DONTWAIT	Enable nonblocking operation (equivalent to using O_NONBLOCK).		•	•		•
MSG_ERRQUEUE	Receive error information as ancillary data.			•		
MSG_OOB	Retrieve out-of-band data if supported by protocol (see Section 16.7).	•	•	•	•	•
MSG_PEEK	Return packet contents without consuming the packet.	•	•	•	•	•
MSG_TRUNC	Request that the real length of the packet be returned, even if it was truncated.			•		
MSG_WAITALL	Wait until all data is available (SOCK_STREAM only).	•	•	•	•	•

Figure 16.14 Flags used with recv socket calls

When we specify the MSG_PEEK flag, we can peek at the next data to be read without actually consuming it. The next call to read or one of the recv functions will return the same data we peeked at.

With SOCK_STREAM sockets, we can receive less data than we requested. The MSG_WAITALL flag inhibits this behavior, preventing recv from returning until all the data we requested has been received. With SOCK_DGRAM and SOCK_SEQPACKET sockets, the MSG_WAITALL flag provides no change in behavior, because these message-based socket types already return an entire message in a single read.

If the sender has called shutdown (Section 16.2) to end transmission, or if the network protocol supports orderly shutdown by default and the sender has closed the socket, then recv will return 0 when we have received all the data.

If we are interested in the identity of the sender, we can use `recvfrom` to obtain the source address from which the data was sent.

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

ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                 struct sockaddr *restrict addr,
                 socklen_t *restrict addrlen);
```

Returns: length of message in bytes,
0 if no messages are available and peer has done an orderly shutdown,
or −1 on error

If *addr* is non-null, it will contain the address of the socket endpoint from which the data was sent. When calling `recvfrom`, we need to set the *addrlen* parameter to point to an integer containing the size in bytes of the socket buffer to which *addr* points. On return, the integer is set to the actual size of the address in bytes.

Because it allows us to retrieve the address of the sender, `recvfrom` is typically used with connectionless sockets. Otherwise, `recvfrom` behaves identically to `recv`.

To receive data into multiple buffers, similar to `readv` (Section 14.6), or if we want to receive ancillary data (Section 17.4), we can use `recvmsg`.

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

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

Returns: length of message in bytes,
0 if no messages are available and peer has done an orderly shutdown,
or −1 on error

The `msghdr` structure (which we saw used with `sendmsg`) is used by `recvmsg` to specify the input buffers to be used to receive the data. We can set the *flags* argument to change the default behavior of `recvmsg`. On return, the `msg_flags` field of the `msghdr` structure is set to indicate various characteristics of the data received. (The `msg_flags` field is ignored on entry to `recvmsg`.) The possible values on return from `recvmsg` are summarized in Figure 16.15. We'll see an example that uses `recvmsg` in Chapter 17.

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`MSG_CTRUNC`	Control data was truncated.	•	•	•	•	•
`MSG_EOR`	End of record was received.	•	•	•	•	•
`MSG_ERRQUEUE`	Error information was received as ancillary data.			•		
`MSG_OOB`	Out-of-band data was received.	•	•	•	•	•
`MSG_TRUNC`	Normal data was truncated.	•	•	•	•	•

Figure 16.15 Flags returned in `msg_flags` by `recvmsg`

Example — Connection-Oriented Client

Figure 16.16 shows a client command that communicates with a server to obtain the output from a system's uptime command. We call this service "remote uptime" (or "ruptime" for short).

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define BUFLEN      128

extern int connect_retry(int, int, int, const struct sockaddr *,
    socklen_t);

void
print_uptime(int sockfd)
{
    int     n;
    char    buf[BUFLEN];

    while ((n = recv(sockfd, buf, BUFLEN, 0)) > 0)
        write(STDOUT_FILENO, buf, n);
    if (n < 0)
        err_sys("recv error");
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err;

    if (argc != 2)
        err_quit("usage: ruptime hostname");
    memset(&hint, 0, sizeof(hint));
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = connect_retry(aip->ai_family, SOCK_STREAM, 0,
          aip->ai_addr, aip->ai_addrlen)) < 0) {
            err = errno;
        } else {
            print_uptime(sockfd);
            exit(0);
        }
    }
    err_exit(err, "can’t connect to %s", argv[1]);
}
```

Figure 16.16 Client command to get uptime from server

This program connects to a server, reads the string sent by the server, and prints the string on the standard output. Since we’re using a SOCK_STREAM socket, we can’t be guaranteed that we will read the entire string in one call to recv, so we need to repeat the call until it returns 0.

The getaddrinfo function might return more than one candidate address for us to use if the server supports multiple network interfaces or multiple network protocols. We try each one in turn, giving up when we find one that allows us to connect to the service. We use the connect_retry function from Figure 16.11 to establish a connection with the server. □

Example — Connection-Oriented Server

Figure 16.17 shows the server that provides the uptime command’s output to the client program from Figure 16.16.

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUFLEN  128
#define QLEN 10

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, const struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int     clfd;
    FILE    *fp;
    char    buf[BUFLEN];

    set_cloexec(sockfd);
    for (;;) {
        if ((clfd = accept(sockfd, NULL, NULL)) < 0) {
            syslog(LOG_ERR, "ruptimed: accept error: %s",
              strerror(errno));
            exit(1);
        }
        set_cloexec(clfd);
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "error: %s\n", strerror(errno));
            send(clfd, buf, strlen(buf), 0);
        } else {
            while (fgets(buf, BUFLEN, fp) != NULL)
                send(clfd, buf, strlen(buf), 0);
            pclose(fp);
        }
        close(clfd);
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("usage: ruptimed");
    if ((n = sysconf(_SC_HOST_NAME_MAX)) < 0)
        n = HOST_NAME_MAX;  /* best guess */
    if ((host = malloc(n)) == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    memset(&hint, 0, sizeof(hint));
    hint.ai_flags = AI_CANONNAME;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
          aip->ai_addrlen, QLEN)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

Figure 16.17 Server program to provide system uptime

To find its address, the server needs to get the name of the host on which it is running. If the maximum host name length is indeterminate, we use HOST_NAME_MAX instead. If the system doesn't define HOST_NAME_MAX, we define it ourselves. POSIX.1 requires the maximum host name length to be at least 255 bytes, not including the terminating null, so we define HOST_NAME_MAX to be 256 to include the terminating null.

The server gets the host name by calling gethostname and looks up the address for the remote uptime service. Multiple addresses can be returned, but we simply choose the first one for which we can establish a passive socket endpoint (i.e., one used only to listen for connect requests). Handling multiple addresses is left as an exercise.

We use the initserver function from Figure 16.12 to initialize the socket endpoint on which we will wait for connect requests to arrive. (Actually, we use the version from Figure 16.22; we'll see why when we discuss socket options in Section 16.6.) □

Example — Alternative Connection-Oriented Server

Previously, we stated that using file descriptors to access sockets was significant, because it allowed programs that knew nothing about networking to be used in a networked environment. The version of the server shown in Figure 16.18 illustrates this point. Instead of reading the output of the uptime command and sending it to the client, the server arranges to have the standard output and standard error of the uptime command be the socket endpoint connected to the client.

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define QLEN 10

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, const struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int     clfd, status;
    pid_t   pid;

    set_cloexec(sockfd);
    for (;;) {
        if ((clfd = accept(sockfd, NULL, NULL)) < 0) {
            syslog(LOG_ERR, "ruptimed: accept error: %s",
              strerror(errno));
            exit(1);
        }
        if ((pid = fork()) < 0) {
            syslog(LOG_ERR, "ruptimed: fork error: %s",
              strerror(errno));
            exit(1);
        } else if (pid == 0) {  /* child */
            /*
             * The parent called daemonize (Figure 13.1), so
             * STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO
             * are already open to /dev/null.  Thus, the call to
             * close doesn’t need to be protected by checks that
             * clfd isn’t already equal to one of these values.
             */
            if (dup2(clfd, STDOUT_FILENO) != STDOUT_FILENO ||
              dup2(clfd, STDERR_FILENO) != STDERR_FILENO) {
                syslog(LOG_ERR, "ruptimed: unexpected error");
                exit(1);
            }
            close(clfd);
            execl("/usr/bin/uptime", "uptime", (char *)0);
            syslog(LOG_ERR, "ruptimed: unexpected return from exec: %s",
              strerror(errno));
        } else {        /* parent */
            close(clfd);
            waitpid(pid, &status, 0);
        }
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("usage: ruptimed");
    if ((n = sysconf(_SC_HOST_NAME_MAX)) < 0)
        n = HOST_NAME_MAX;  /* best guess */
    if ((host = malloc(n)) == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    memset(&hint, 0, sizeof(hint));
    hint.ai_flags = AI_CANONNAME;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
          aip->ai_addrlen, QLEN)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

Figure 16.18 Server program illustrating command writing directly to socket

Instead of using popen to run the uptime command and reading the output from the pipe connected to the command's standard output, we use fork to create a child process and then use dup2 to arrange that the child's copy of STDIN_FILENO is open to /dev/null and that both STDOUT_FILENO and STDERR_FILENO are open to the socket endpoint. When we execute uptime, the command writes the results to its standard output, which is connected to the socket, and the data is sent back to the ruptime client command.

The parent can safely close the file descriptor connected to the client, because the child still has it open. The parent waits for the child to complete before proceeding, so that the child doesn't become a zombie. Since it shouldn't take too long to run the uptime command, the parent can afford to wait for the child to exit before accepting the next connect request. This strategy might not be appropriate if the child takes a long time, however. □

The previous examples have used connection-oriented sockets. But how do we choose the appropriate type? When do we use a connection-oriented socket, and when do we use a connectionless socket? The answer depends on how much work we want to do and how much tolerance we have for errors.

With a connectionless socket, packets can arrive out of order, so if we can't fit all our data in one packet, we will have to worry about ordering in our application. The maximum packet size is a characteristic of the communication protocol. Also, with a connectionless socket, the packets can be lost. If our application can't tolerate this loss, we should use connection-oriented sockets.

Tolerating packet loss means that we have two choices. If we intend to have reliable communication with our peer, we have to number our packets and request retransmission from the peer application when we detect a missing packet. We also have to identify duplicate packets and discard them, since a packet might be delayed and appear to be lost, but show up after we have requested retransmission.

The other choice we have is to deal with the error by letting the user retry the command. For simple applications this might be adequate, but for complex applications it usually isn't a viable alternative. Thus, it is generally better to use connection-oriented sockets in this case.

The drawbacks to connection-oriented sockets are that more work and time are needed to establish a connection, and each connection consumes more resources from the operating system.

Example — Connectionless Client

The program in Figure 16.19 is a version of the uptime client command that uses the datagram socket interface.

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define BUFLEN      128
#define TIMEOUT     20

void
sigalrm(int signo)
{
}

void
print_uptime(int sockfd, struct addrinfo *aip)
{
    int     n;
    char    buf[BUFLEN];

    buf[0] = 0;
    if (sendto(sockfd, buf, 1, 0, aip->ai_addr, aip->ai_addrlen) < 0)
        err_sys("sendto error");
    alarm(TIMEOUT);
    if ((n = recvfrom(sockfd, buf, BUFLEN, 0, NULL, NULL)) < 0) {
        if (errno != EINTR)
            alarm(0);
        err_sys("recv error");
    }
    alarm(0);
    write(STDOUT_FILENO, buf, n);
}

int
main(int argc, char *argv[])
{
    struct addrinfo     *ailist, *aip;
    struct addrinfo     hint;
    int                 sockfd, err;
    struct sigaction    sa;

    if (argc != 2)
        err_quit("usage: ruptime hostname");
    sa.sa_handler = sigalrm;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGALRM, &sa, NULL) < 0)
        err_sys("sigaction error");
    memset(&hint, 0, sizeof(hint));
    hint.ai_socktype = SOCK_DGRAM;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));

    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = socket(aip->ai_family, SOCK_DGRAM, 0)) < 0) {
            err = errno;
        } else {
            print_uptime(sockfd, aip);
            exit(0);
        }
    }

    fprintf(stderr, "can’t contact %s: %s\n", argv[1], strerror(err));
    exit(1);
}
```

Figure 16.19 Client command using datagram service

The `main` function for the datagram-based client is similar to the one for the connection-oriented client, with the addition of installing a signal handler for `SIGALRM`. We use the `alarm` function to avoid blocking indefinitely in the call to `recvfrom`.

With the connection-oriented protocol, we needed to connect to the server before exchanging data. The arrival of the connect request was enough for the server to determine that it needed to provide service to a client. But with the datagram-based protocol, we need a way to notify the server that we want it to perform its service on our behalf. In this example, we simply send the server a 1-byte message. The server will receive it, get our address from the packet, and use this address to transmit its response. If the server offered multiple services, we could use this request message to indicate the service we want, but since the server does only one thing, the content of the 1-byte message doesn’t matter.

If the server isn’t running, the client will block indefinitely in the call to `recvfrom`. With the connection-oriented example, the `connect` call will fail if the server isn’t running. To avoid blocking indefinitely, we set an alarm clock before calling `recvfrom`. □

Example — Connectionless Server

The program in Figure 16.20 is the datagram version of the uptime server.

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUFLEN      128
#define MAXADDRLEN  256

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, const struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int             n;
    socklen_t       alen;
    FILE            *fp;
    char            buf[BUFLEN];
    char            abuf[MAXADDRLEN];
    struct sockaddr *addr = (struct sockaddr *)abuf;

    set_cloexec(sockfd);
    for (;;) {
        alen = MAXADDRLEN;
        if ((n = recvfrom(sockfd, buf, BUFLEN, 0, addr, &alen)) < 0) {
            syslog(LOG_ERR, "ruptimed: recvfrom error: %s",
              strerror(errno));
            exit(1);
        }
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "error: %s\n", strerror(errno));
            sendto(sockfd, buf, strlen(buf), 0, addr, alen);
        } else {
            if (fgets(buf, BUFLEN, fp) != NULL)
                sendto(sockfd, buf, strlen(buf), 0, addr, alen);
            pclose(fp);
        }
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;
    if (argc != 1)
        err_quit("usage: ruptimed");
    if ((n = sysconf(_SC_HOST_NAME_MAX)) < 0)
        n = HOST_NAME_MAX;  /* best guess */
    if ((host = malloc(n)) == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    memset(&hint, 0, sizeof(hint));
    hint.ai_flags = AI_CANONNAME;
    hint.ai_socktype = SOCK_DGRAM;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_DGRAM, aip->ai_addr,
          aip->ai_addrlen, 0)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

Figure 16.20 Server providing system uptime over datagrams

The server blocks in `recvfrom` for a request for service. When a request arrives, we save the requester's address and use `popen` to run the `uptime` command. We send the output back to the client using the `sendto` function, with the destination address set to the requester's address. □

16.6 Socket Options

The socket mechanism provides two socket-option interfaces for us to control the behavior of sockets. One interface is used to set an option, and another interface allows us to query the state of an option. We can get and set three kinds of options:

1. Generic options that work with all socket types
2. Options that are managed at the socket level, but depend on the underlying protocols for support
3. Protocol-specific options unique to each individual protocol

The Single UNIX Specification defines only the socket-layer options (the first two option types in the preceding list).

We can set a socket option with the `setsockopt` function.

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

int setsockopt(int sockfd, int level, int option, const void *val,
               socklen_t len);

                                          Returns: 0 if OK, −1 on error
```

The *level* argument identifies the protocol to which the option applies. If the option is a generic socket-level option, then *level* is set to `SOL_SOCKET`. Otherwise, *level* is set to the number of the protocol that controls the option. Examples are `IPPROTO_TCP` for TCP options and `IPPROTO_IP` for IP options. Figure 16.21 summarizes the generic socket-level options defined by the Single UNIX Specification.

Option	Type of *val* argument	Description
`SO_ACCEPTCONN`	`int`	Return whether a socket is enabled for listening (`getsockopt` only).
`SO_BROADCAST`	`int`	Broadcast datagrams if `*val` is nonzero.
`SO_DEBUG`	`int`	Debugging in network drivers enabled if `*val` is nonzero.
`SO_DONTROUTE`	`int`	Bypass normal routing if `*val` is nonzero.
`SO_ERROR`	`int`	Return and clear pending socket error (`getsockopt` only).
`SO_KEEPALIVE`	`int`	Periodic keep-alive messages enabled if `*val` is nonzero.
`SO_LINGER`	`struct linger`	Delay time when unsent messages exist and socket is closed.
`SO_OOBINLINE`	`int`	Out-of-band data placed inline with normal data if `*val` is nonzero.
`SO_RCVBUF`	`int`	The size in bytes of the receive buffer.
`SO_RCVLOWAT`	`int`	The minimum amount of data in bytes to return on a receive call.
`SO_RCVTIMEO`	`struct timeval`	The timeout value for a socket receive call.
`SO_REUSEADDR`	`int`	Reuse addresses in `bind` if `*val` is nonzero.
`SO_SNDBUF`	`int`	The size in bytes of the send buffer.
`SO_SNDLOWAT`	`int`	The minimum amount of data in bytes to transmit in a send call.
`SO_SNDTIMEO`	`struct timeval`	The timeout value for a socket send call.
`SO_TYPE`	`int`	Identify the socket type (`getsockopt` only).

Figure 16.21 Socket options

The *val* argument points to a data structure or an integer, depending on the option. Some options are on/off switches. If the integer is nonzero, then the option is enabled. If the integer is zero, then the option is disabled. The *len* argument specifies the size of the object to which *val* points.

We can find out the current value of an option with the `getsockopt` function.

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

int getsockopt(int sockfd, int level, int option, void *restrict val,
               socklen_t *restrict lenp);

                                          Returns: 0 if OK, −1 on error
```

The *lenp* argument is a pointer to an integer. Before calling `getsockopt`, we set the integer to the size of the buffer where the option is to be copied. If the actual size of the option is greater than this size, the option is silently truncated. If the actual size of the option is less than this size, then the integer is updated with the actual size on return.

Example

The function in Figure 16.12 fails to operate properly when the server terminates and we try to restart it immediately. Normally, the implementation of TCP will prevent us from binding the same address until a timeout expires, which is usually on the order of several minutes. Luckily, the `SO_REUSEADDR` socket option allows us to bypass this restriction, as illustrated in Figure 16.22.

```
#include "apue.h"
#include <errno.h>
#include <sys/socket.h>

int
initserver(int type, const struct sockaddr *addr, socklen_t alen,
  int qlen)
{
    int fd, err;
    int reuse = 1;

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &reuse,
      sizeof(int)) < 0)
        goto errout;
    if (bind(fd, addr, alen) < 0)
        goto errout;
    if (type == SOCK_STREAM || type == SOCK_SEQPACKET)
        if (listen(fd, qlen) < 0)
            goto errout;
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(-1);
}
```

Figure 16.22 Initialize a socket endpoint for use by a server with address reuse

To enable the `SO_REUSEADDR` option, we set an integer to a nonzero value and pass the address of the integer as the *val* argument to `setsockopt`. We set the *len* argument to the size of an integer to indicate the size of the object to which *val* points. □

16.7 Out-of-Band Data

Out-of-band data is an optional feature supported by some communication protocols, allowing higher-priority delivery of data than normal. Out-of-band data is sent ahead of any data that is already queued for transmission. TCP supports out-of-band data, but UDP doesn't. The socket interface to out-of-band data is heavily influenced by TCP's implementation of out-of-band data.

TCP refers to out-of-band data as ''urgent'' data. TCP supports only a single byte of urgent data, but allows urgent data to be delivered out of band from the normal data delivery mechanisms. To generate urgent data, we specify the `MSG_OOB` flag to any of the three `send` functions. If we send more than one byte with the `MSG_OOB` flag, the last byte will be treated as the urgent-data byte.

When urgent data is received, we are sent the `SIGURG` signal if we have arranged for signal generation by the socket. In Sections 3.14 and 14.5.2, we saw that we could use the `F_SETOWN` command to `fcntl` to set the ownership of a socket. If the third argument to `fcntl` is positive, it specifies a process ID. If it is a negative value other than −1, it represents the process group ID. Thus, we can arrange that our process receive signals from a socket by calling

```
fcntl(sockfd, F_SETOWN, pid);
```

The `F_GETOWN` command can be used to retrieve the current socket ownership. As with the `F_SETOWN` command, a negative value represents a process group ID and a positive value represents a process ID. Thus, the call

```
owner = fcntl(sockfd, F_GETOWN, 0);
```

will return with `owner` equal to the ID of the process configured to receive signals from the socket if `owner` is positive and with the absolute value of `owner` equal to the ID of the process group configured to receive signals from the socket if `owner` is negative.

TCP supports the notion of an *urgent mark*: the point in the normal data stream where the urgent data would go. We can choose to receive the urgent data inline with the normal data if we use the `SO_OOBINLINE` socket option. To help us identify when we have reached the urgent mark, we can use the `sockatmark` function.

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

int sockatmark(int sockfd);
```

Returns: 1 if at mark, 0 if not at mark, −1 on error

When the next byte to be read is at the urgent mark, `sockatmark` will return 1.

When out-of-band data is present in a socket's read queue, the `select` function (Section 14.4.1) will return the file descriptor as having an exception condition pending. We can choose to receive the urgent data inline with the normal data, or we can use the `MSG_OOB` flag with one of the `recv` functions to receive the urgent data ahead of any other queue data. TCP queues only one byte of urgent data. If another urgent byte arrives before we receive the current one, the existing one is discarded.

16.8 Nonblocking and Asynchronous I/O

Normally, the `recv` functions will block when no data is immediately available. Similarly, the `send` functions will block when there is not enough room in the socket's output queue to send the message. This behavior changes when the socket is in nonblocking mode. In this case, these functions will fail instead of blocking, setting `errno` to either `EWOULDBLOCK` or `EAGAIN`. When this happens, we can use either `poll` or `select` to determine when we can receive or transmit data.

The Single UNIX Specification includes support for a general asynchronous I/O mechanism (recall Section 14.5). The socket mechanism has its own way of handling asynchronous I/O, but this isn't standardized in the Single UNIX Specification. Some texts refer to the classic socket-based asynchronous I/O mechanism as "signal-based I/O" to distinguish it from the general asynchronous I/O mechanism found in the Single UNIX Specification.

With socket-based asynchronous I/O, we can arrange to be sent the `SIGIO` signal when we can read data from a socket or when space becomes available in a socket's write queue. Enabling asynchronous I/O is a two-step process.

1. Establish socket ownership so signals can be delivered to the proper processes.
2. Inform the socket that we want it to signal us when I/O operations won't block.

We can accomplish the first step in three ways.

1. Use the `F_SETOWN` command with `fcntl`.
2. Use the `FIOSETOWN` command with `ioctl`.
3. Use the `SIOCSPGRP` command with `ioctl`.

To accomplish the second step, we have two choices.

1. Use the `F_SETFL` command with `fcntl` and enable the `O_ASYNC` file flag.
2. Use the `FIOASYNC` command with `ioctl`.

We have several options, but they are not universally supported. Figure 16.23 summarizes the support for these options provided by the platforms discussed in this text.

Mechanism	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`fcntl(fd, F_SETOWN, pid)`	•	•	•	•	•
`ioctl(fd, FIOSETOWN, pid)`		•	•	•	•
`ioctl(fd, SIOCSPGRP, pid)`		•	•	•	•
`fcntl(fd, F_SETFL, flags|O_ASYNC)`		•	•	•	
`ioctl(fd, FIOASYNC, &n);`		•	•	•	•

Figure 16.23 Socket asynchronous I/O management commands

16.9 Summary

In this chapter, we looked at the IPC mechanisms that allow processes to communicate with other processes on different machines as well as within the same machine. We discussed how socket endpoints are named and how we can discover the addresses to use when contacting servers.

We presented examples of clients and servers that use connectionless (i.e., datagram-based) sockets and connection-oriented sockets. We briefly discussed asynchronous and nonblocking socket I/O and the interfaces used to manage socket options.

In the next chapter, we will look at some advanced IPC topics, including how we can use sockets to pass file descriptors between processes running on the same machine.

Exercises

16.1 Write a program to determine your system's byte ordering.

16.2 Write a program to print out which `stat` structure members are supported for sockets on at least two different platforms, and describe how the results differ.

16.3 The program in Figure 16.17 provides service on only a single endpoint. Modify the program to support service on multiple endpoints (each with a different address) at the same time.

16.4 Write a client program and a server program to return the number of processes currently running on a specified host computer.

16.5 In the program in Figure 16.18, the server waits for the child to execute the `uptime` command and exit before accepting the next connect request. Redesign the server so that the time to service one request doesn't delay the processing of incoming connect requests.

16.6 Write two library routines: one to enable asynchronous (signal-based) I/O on a socket and one to disable asynchronous I/O on a socket. Use Figure 16.23 to make sure that the functions work on all platforms with as many socket types as possible.