Appendix B

Sockets

The socket application programming interface (API) was developed at the University of California in Berkeley as part of the work on the BSD Unix system. The socket interface is a generic interface for interprocess communication using message passing. Sockets are abstract communication endpoints with a rather small number of associated function calls. The socket API distinguishes different types of sockets:

- A *stream socket* (`SOCK_STREAM`) is a bidirectional reliable communication endpoint. Data written to a local stream socket can be read from a remote stream socket without having to worry about transmission errors, fragmentation or reordering that might occur in the underlying network. While the order of the byte stream is not changed, the data block boundaries are not preserved.
- A *datagram socket* (`SOCK_DGRAM`) is a bidirectional unreliable communication endpoint which allows to exchange datagrams. Datagrams send over the local datagram socket may not be received by the remote datagram socket or they may be received multiple times. Furthermore, the ordering of the datagrams can change during the transmission. Note that datagram boundaries are preserved.
- A *raw socket* (`SOCK_RAW`) is a communication endpoint which allows to receive and send network or interface layer datagrams.
- A *reliable delivered message socket* (`SOCK_RDM`) is similar to a datagram socket but provides in addition reliable datagram delivery.
- A *sequenced packet socket* (`SOCK_SEQPACKET`) is similar to a stream socket but retains data block boundaries.

B.1 Socket Addresses

It is necessary to assign a name (an address) to a communication endpoint before it can be used. The socket API supports different name spaces with different address formats. The generic data structure for addresses (`struct sockaddr`) is defined as follows:

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

struct sockaddr {
    uint8_t     sa_len          /* address length (BSD) */
    sa_family_t sa_family;      /* address family */
    char        sa_data[...];   /* data of some size */
};

struct sockaddr_storage {
    uint8_t     ss_len;         /* address length (BSD) */
    sa_family_t ss_family;      /* address family */
    char        padding[...];   /* padding of some size */
};
```

Newer BSD systems support the (`sa_len`) field in the generic and the specific socket addresses which was not present in the older socket API. Other systems usually do not have this (`sa_len`) member (although it is generally a good idea to have this member). The currently most important name spaces are the name spaces for the Internet and a name space for local communication:

IPv4 Socket Addresses

Sockets that represent IPv4 communication endpoints use the address family `AF_INET` and the protocol family `PF_INET`. IPv4 transport addresses are represented by the structure `struct sockaddr_in`:

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

typedef ... sa_family_t;

#include <netinet/in.h>

typedef ... in_port_t;

struct in_addr {
    uint8_t  s_addr[4];         /* IPv4 address */
};

struct sockaddr_in {
    uint8_t     sin_len;        /* address length (BSD) */
    sa_family_t sin_family;     /* address family */
    in_port_t   sin_port;       /* transport layer port */
    struct in_addr sin_addr;    /* IPv4 address */
};
```

IPv6 Socket Addresses

Sockets that represent IPv6 communication endpoints use the address family `AF_INET6` and and the protocol family `PF_INET6`. IPv6 transport addresses are represented by the structure `struct sockaddr_in6`:

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

typedef ... sa_family_t;

#include <netinet/in.h>

typedef ... in_port_t;

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

struct sockaddr_in6 {
    uint8_t     sin6_len;       /* address length (BSD) */
    sa_family_t sin6_family;    /* address family */
    in_port_t   sin6_port;      /* transport layer port */
    uint32_t    sin6_flowinfo;  /* flow information */
    struct in6_addr sin6_addr;  /* IPv6 address */
    uint32_t    sin6_scope_id;  /* scope identifier */
};
```

Local Socket Addresses

Sockets that represent local communication endpoints use the address family `AF_LOCAL` and the protocol family `PF_LOCAL`. Local sockets are also widely known under the name Unix sockets with the address family `AF_UNIX` and the protocol family `PF_UNIX`. Local socket addresses are represented by the structure `struct sockaddr_un`:

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

typedef ... sa_family_t;

#include <sys/un.h>

struct sockaddr_un {
    uint8_t     sun_len;        /* address length (BSD) */
    sa_family_t sun_family;     /* address family */
    char        sun_path[108];  /* xxx Is 108 POSIX ? */
};
```

B.2 Communication Kinds

The socket API allows to realize different communication kinds in application programs. The most important two styles are connection-less datagram communication and connection-oriented data stream communication.

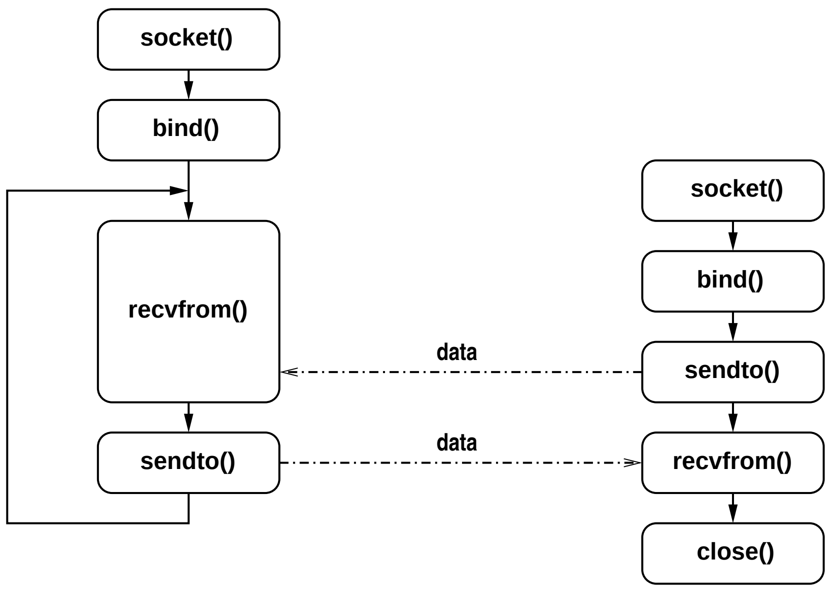

Figure B.1: Connection-less datagram communication

Figure B.1 shows how a server and a client make use of the socket primitives to provide and realize a connection-less datagram application protocol. After creating and binding a local socket, the processes use the `recvfrom()` and the `sendto()` primitives to receive and send datagrams.

Figure B.2 shows how a server and a client make use of the socket primitives to provide and realize a connection-oriented application protocol. The server creates a listening local socket which is used to accept incoming connections. Once a connection has been accepted, a new local file descriptor is returned which can be used to `read()` or `write()` data. The `close()` function is called to close the connection. On the client side, the `connect()` function is used to connected the local socket to a remote (server) socket. When the `connect()` function returns successfully, normal `read()` or `write()` functions can be used to exchange data. The `close()` function is again called to close the connection.

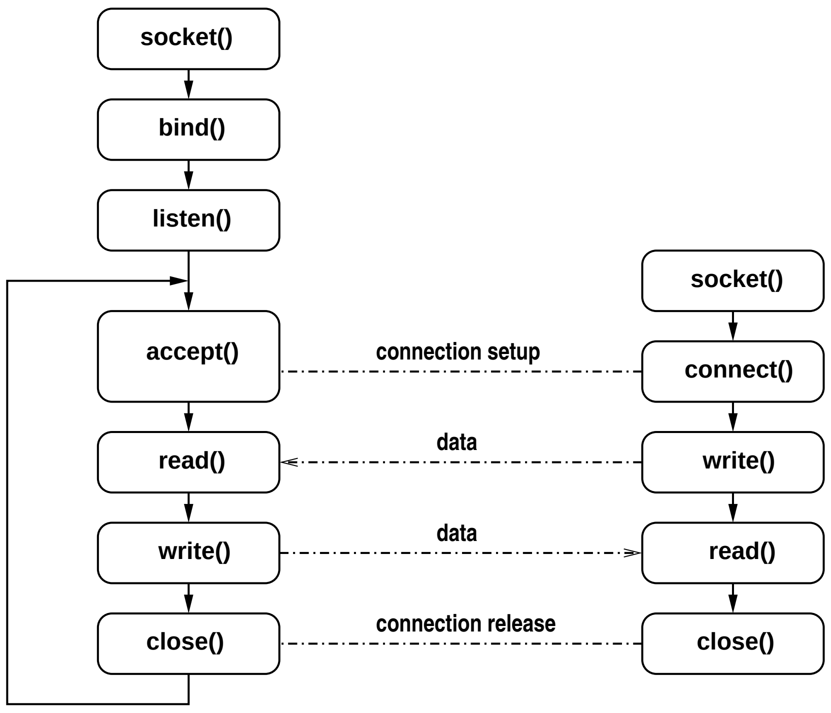

Figure B.2: Connection-oriented data stream communication

B.3 Socket API Overview

The following definitions summarize the socket API. For a more detailed description, please consult the corresponding Unix manual pages.

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

#define SOCK_STREAM    ...
#define SOCK_DGRAM     ...
#define SCOK_RAW       ...
#define SOCK_RDM       ...
#define SOCK_SEQPACKET ...

#define AF_LOCAL ...
#define AF_INET  ...
#define AF_INET6 ...

#define PF_LOCAL ...
#define PF_INET  ...
#define PF_INET6 ...

int socket(int domain, int type, int protocol);
int bind(int socket, struct sockaddr *addr,
         socklen_t addrlen);
int connect(int socket, struct sockaddr *addr,
            socklen_t addrlen);
int listen(int socket, int backlog);
int accept(int socket, struct sockaddr *addr,
           socklen_t *addrlen);

ssize_t write(int socket, void *buf, size_t count);
int send(int socket, void *msg, size_t len, int flags);
int sendto(int socket, void *msg, size_t len, int flags,
           struct sockaddr *addr, socklen_t addrlen);

ssize_t read(int socket, void *buf, size_t count);
int recv(int socket, void *buf, size_t len, int flags);
int recvfrom(int socket, void *buf, size_t len, int flags,
             struct sockaddr *addr, socklen_t *addrlen);

int shutdown(int socket, int how);
int close(int socket);

int getsockopt(int socket, int level, int optname,
               void *optval, socklen_t *optlen);
int setsockopt(int socket, int level, int optname,
               void *optval, socklen_t optlen);
int getsockname(int socket, struct sockaddr *addr,
                socklen_t *addrlen);
int getpeername(int socket, struct sockaddr *addr,
                socklen_t *addrlen);
```

B.4 Name Resolution

Numeric addresses are usually hard to memorize for humans. It thus useful to introduce more human friendly symbolic names. The Internet protocols use the Domain Name System (DNS) to map symbolic names to Internet addresses. Note that DNS supports IPv4 as well as IPv6 addresses. Furthermore, there is usually also a locally defined mapping of well-known port numbers to symbolic names (e.g., port number 80 has the well-known symbolic names http or www).

The name to address mapping is supported by the functions `getaddrinfo()` and `getnameinfo()` which are described below. Many older programs still use the functions `gethostbyname()` and `gethostbyaddr()` which have been deprecated.

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

#define AI_PASSIVE     ...
#define AI_CANONNAME   ...
#define AI_NUMERICHOST ...

struct addrinfo {
    int             ai_flags;
    int             ai_family;
    int             ai_socktype;
    int             ai_protocol;
    size_t          ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};

int getaddrinfo(const char *node,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
void freeaddrinfo(struct addrinfo *res);
const char *gai_strerror(int errcode);
```

The mapping of names to addresses is realized by the function `getaddrinfo()`. This function has three input parameters (`node`, `service`, `hints`) and returns a pointer to a list of `struct addrinfo` elements. This list must be released by calling `freeaddrinfo()` if it is not used anymore. In case of an error, `getaddrinfo()` returns a value unequal to 0 which can be passed to `gai_strerror()` in order to get a human readable error description.

One of the arguments `node` and `service` can be `NULL` thus requesting only a name resolution of the other element. The name resolution process can be further controlled by passing some `hints` to the function. Hints can be used, for example, to request addresses of a certain address family or socket type.

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

#define NI_NOFQDN       ...
#define NI_NUMERICHOST  ...
#define NI_NAMEREQD     ...
#define NI_NUMERICSERV  ...
#define NI_NUMERICSCOPE ...
#define NI_DGRAM        ...

int getnameinfo(const struct sockaddr *sa,
                socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen,
                int flags);
const char *gai_strerror(int errcode);
```

The inverse mapping of addresses to symbolic names is supported by the function `getnameinfo()`. The first two parameters (`sa`, `salen`) are input parameters. The result of the mapping is a host name and a service name which is written to the memory location `host` with the length `hostlen` and `serv` with the length `servlen`. Additional `flags` can be passed to the mapping function in order to control the details of the mapping process.

Example 3 *The following source code implements a client for a simple connection-oriented protocol which retrieves the date and time from a server.*

```
/*
 * daytime1/daytime.c --
 *
 * A simple TCP over IPv4/IPv6 daytime client.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>

static const char *progname = "daytime";
```

```
/*
 * Establish a connection to a remote TCP server. First get the list
 * of potential network layer addresses and transport layer port
 * numbers. Iterate through the returned address list until an attempt
 * to establish a TCP connection is successful (or no other
 * alternative exists).
 */

static int
tcp_connect(char *host, char *port)
{
    struct addrinfo hints, *ai_list, *ai;
    int n, fd = 0;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    n = getaddrinfo(host, port, &hints, &ai_list);
    if (n) {
        fprintf(stderr, "%s: getaddrinfo: %s\n",
                progname, gai_strerror(n));
        exit(EXIT_FAILURE);
    }

    for (ai = ai_list; ai; ai = ai->ai_next) {
        fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (fd < 0) {
            continue;
        }
        if (connect(fd, ai->ai_addr, ai->ai_addrlen) == 0) {
            break;
        }
        close(fd);
    }

    freeaddrinfo(ai_list);

    if (ai == NULL) {
        fprintf(stderr, "%s: socket or connect: failed for %s port %s\n",
                progname, host, port);
        exit(EXIT_FAILURE);
    }

    return fd;
}

/*
 * Close a TCP connection. This function trivially calls close() on
 * POSIX systems, but might be more complicated on other systems.
 */

static int
```

```
tcp_close(int fd)
{
    return close(fd);
}

/*
 * Implement the daytime protocol, loosely modeled after RFC 867.
 */

static void
daytime(int fd)
{
    struct sockaddr_storage peer;
    socklen_t peerlen = sizeof(peer);
    char host[NI_MAXHOST];
    char serv[NI_MAXSERV];
    char message[128], *p;
    ssize_t n;

    /* Get the socket address of the remote end and convert it
     * into a human readable string (numeric format). */

    n = getpeername(fd, (struct sockaddr *) &peer, &peerlen);
    if (n) {
        fprintf(stderr, "%s: getpeername: %s\n",
                progname, strerror(errno));
        return;
    }

    n = getnameinfo((struct sockaddr *) &peer, peerlen,
                    host, sizeof(host), serv, sizeof(serv),
                    NI_NUMERICHOST | NI_NUMERICSERV);
    if (n) {
        fprintf(stderr, "%s: getnameinfo: %s\n",
                progname, gai_strerror(n));
        return;
    }

    while ((n = read(fd, message, sizeof(message) - 1)) > 0) {
        message[n] = '\0';
        p = strstr(message, "\r\n");
        if (p) *p = 0;
        printf("%s:%s\t %s\n", host, serv, message);
    }
}


int
main(int argc, char **argv)
{
    int fd;

    if (argc != 3) {
        fprintf(stderr, "usage: %s host port\n", progname);
        return EXIT_FAILURE;
    }

    fd = tcp_connect(argv[1], argv[2]);
    daytime(fd);
    tcp_close(fd);

    return EXIT_SUCCESS;
}
```

Example 4 *The following source code implements a server for a simple connection-oriented protocol which retrieves the date and time from a server.*

```
/*
 * daytimed1/daytimed.c --
 *
 * A simple TCP over IPv4/IPv6 daytime server. The server waits for
 * incoming connections, sends a daytime string as a reaction to
 * successful connection establishment and finally closes the
 * connection down again.
 */

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <syslog.h>
#include <string.h>
#include <time.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>

static const char *progname = "daytimed";

/*
 * Store the current date and time of the day using the local timezone
 * in the given buffer of the indicated size. Set the buffer to a zero
 * length string in case of errors.
 */

static void
daytime(char *buffer, size_t size)
{
    time_t ticks;
    struct tm *tm;

    ticks = time(NULL);
    tm = localtime(&ticks);
    if (tm == NULL) {
        buffer[0] = '\0';
        syslog(LOG_ERR, "localtime failed");
        return;
    }
    strftime(buffer, size, "%F %T\r\n", tm);
}

/*
 * Create a listening TCP endpoint. First get the list of potential
 * network layter addresses and transport layer port numbers. Iterate
 * through the returned address list until an attempt to create a
 * listening TCP endpoint is successful (or no other alternative
 * exists).
 */

static int
tcp_listen(char *port)
{
    struct addrinfo hints, *ai_list, *ai;
    int n, fd = 0, on = 1;

    memset(&hints, 0, sizeof(hints));
    hints.ai_flags = AI_PASSIVE;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    n = getaddrinfo(NULL, port, &hints, &ai_list);
    if (n) {
        fprintf(stderr, "%s: getaddrinfo failed: %s\n",
                progname, gai_strerror(n));
        return -1;
    }

    for (ai = ai_list; ai; ai = ai->ai_next) {
        fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (fd < 0) {
            continue;
        }

        setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
        if (bind(fd, ai->ai_addr, ai->ai_addrlen) == 0) {
            break;
        }
        close(fd);
    }

    freeaddrinfo(ai_list);

    if (ai == NULL) {
        fprintf(stderr, "%s: bind failed for port %s\n",
                progname, port);
        return -1;
    }

    if (listen(fd, 42) < 0) {
        fprintf(stderr, "%s: listen failed: %s\n",
                progname, strerror(errno));
close(fd);
        return -1;
    }

    return fd;
}

/*
 * Accept a new TCP connection and write a message about who was
 * accepted to the system log.
 */

static int
tcp_accept(int listen)
{
    struct sockaddr_storage ss;
    socklen_t ss_len = sizeof(ss);
    char host[NI_MAXHOST];
    char serv[NI_MAXSERV];
    int n, fd;

    fd = accept(listen, (struct sockaddr *) &ss, &ss_len);
    if (fd == -1) {
        syslog(LOG_ERR, "accept failed: %s", strerror(errno));
        return -1;
    }

    n = getnameinfo((struct sockaddr *) &ss, ss_len,
                    host, sizeof(host), serv, sizeof(serv),
                    NI_NUMERICHOST);
    if (n) {
        syslog(LOG_ERR, "getnameinfo failed: %s", gai_strerror(n));
    } else {
        syslog(LOG_DEBUG, "connection from %s:%s", host, serv);
    }

    return fd;
}

/*
 * Close a TCP connection. This function trivially calls close() on
 * POSIX systems, but might be more complicated on other systems.
 */

static int
tcp_close(int fd)
{
    return close(fd);
}

/*
 * Implement the daytime protocol, loosely modeled after RFC 867.
 */
```

```
static void
tcp_daytime(int listenfd)
{
    size_t n;
    int client;
    char message[128];

    client = tcp_accept(listenfd);
    if (client == -1) {
        return;
    }

    daytime(message, sizeof(message));

    n = write(client, message, strlen(message));
    if (n != strlen(message)) {
        syslog(LOG_ERR, "write failed");
        return;
    }

    tcp_close(client);
}


int
main(int argc, char **argv)
{
    int tfd;

    if (argc != 2) {
        fprintf(stderr, "usage: %s port\n", progname);
        exit(EXIT_FAILURE);
    }

    openlog(progname, LOG_PID, LOG_DAEMON);

    tfd = tcp_listen(argv[1]);

    while (tfd != -1) {
        tcp_daytime(tfd);
    }

    tcp_close(tfd);

    closelog();

    return EXIT_SUCCESS;
}
```

Example 5 *The following source code implements a client for a simple connection-less protocol which retrieves the date and time from a server.*

```
/*
```

```
 * daytime2/daytime.c --
 *
 * A simple UDP over IPv4/IPv6 daytime client. This version uses a
 * connected UDP socket.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>

static const char *progname = "daytime";

/*
 * Establish a connected UDP endpoint. First get the list of potential
 * network layer addresses and transport layer port numbers. Iterate
 * through the returned address list until an attempt to create and
 * connect a UDP endpoint is successful (or no other alternative
 * exists).
 */

static int
udp_connect(char *host, char *port)
{
    struct addrinfo hints, *ai_list, *ai;
    int n, fd = 0;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;

    n = getaddrinfo(host, port, &hints, &ai_list);
    if (n) {
        fprintf(stderr, "%s: getaddrinfo: %s\n",
                progname, gai_strerror(n));
        exit(EXIT_FAILURE);
    }

    for (ai = ai_list; ai; ai = ai->ai_next) {
        fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (fd < 0) {
            continue;
        }
        if (connect(fd, ai->ai_addr, ai->ai_addrlen) == 0) {
            break;
        }
        close(fd);
    }
    freeaddrinfo(ai_list);

    if (ai == NULL) {
        fprintf(stderr, "%s: socket or connect: failed for %s port %s\n",
                progname, host, port);
        exit(EXIT_FAILURE);
    }

    return fd;
}

/*
 * Close a udp endpoint. This function trivially calls close() on
 * POSIX systems, but might be more complicated on other systems.
 */

static int
udp_close(int fd)
{
    return close(fd);
}

/*
 * Implement the daytime protocol, loosely modeled after RFC 867.
 */

static void
daytime(int fd)
{
    struct sockaddr_storage peer;
    socklen_t peerlen = sizeof(peer);
    char host[NI_MAXHOST];
    char serv[NI_MAXSERV];
    char message[128], *p;
    ssize_t n;

    n = send(fd, "", 0, 0);
    if (n == -1) {
        fprintf(stderr, "%s: send: %s\n",
                progname, strerror(errno));
        return;
    }

    n = recv(fd, message, sizeof(message) - 1, 0);
    if (n == -1) {
        fprintf(stderr, "%s: recv: %s\n",
                progname, strerror(errno));
        return;
    }
    message[n] = '\0';

    /* Get the socket address of the remote end and convert it
     * into a human readable string (numeric format). */
    n = getpeername(fd, (struct sockaddr *) &peer, &peerlen);
    if (n) {
        fprintf(stderr, "%s: getpeername: %s\n",
                progname, strerror(errno));
        return;
    }

    n = getnameinfo((struct sockaddr *) &peer, peerlen,
                    host, sizeof(host), serv, sizeof(serv),
                    NI_NUMERICHOST | NI_NUMERICSERV | NI_DGRAM);
    if (n) {
        fprintf(stderr, "%s: getnameinfo: %s\n",
                progname, gai_strerror(n));
        return;
    }

    p = strstr(message, "\r\n");
    if (p) *p = 0;
    printf("%s:%s\t %s\n", host, serv, message);
}

int
main(int argc, char **argv)
{
    int fd;

    if (argc != 3) {
        fprintf(stderr, "usage: %s host port\n", progname);
        return EXIT_FAILURE;
    }

    fd = udp_connect(argv[1], argv[2]);
    daytime(fd);
    udp_close(fd);

    return EXIT_SUCCESS;
}
```

Example 6 *The following source code implements a server for a simple connection-less protocol which retrieves the date and time from a server.*

```
/*
 * daytimed2/daytimed.c --
 *
 * A simple UDP over IPv4/IPv6 daytime server. The server waits for
 * incoming messages and sends a daytime string as a reaction to
 * received message.
 */

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <syslog.h>
```

```
#include <string.h>
#include <time.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>

static const char *progname = "daytimed";

/*
 * Store the current date and time of the day using the local timezone
 * in the given buffer of the indicated size. Set the buffer to a zero
 * length string in case of errors.
 */

static void
daytime(char *buffer, size_t size)
{
    time_t ticks;
    struct tm *tm;

    ticks = time(NULL);
    tm = localtime(&ticks);
    if (tm == NULL) {
        buffer[0] = '\0';
        syslog(LOG_ERR, "localtime failed");
        return;
    }
    strftime(buffer, size, "%F %T\r\n", tm);
}

/*
 * Create a named UDP endpoint. First get the list of potential
 * network layter addresses and transport layer port numbers. Iterate
 * through the returned address list until an attempt to create a UDP
 * endpoint is successful (or no other alternative exists).
 */

static int
udp_open(char *port)
{
    struct addrinfo hints, *ai_list, *ai;
    int n, fd = 0, on = 1;

    memset(&hints, 0, sizeof(hints));
    hints.ai_flags = AI_PASSIVE;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;

    n = getaddrinfo(NULL, port, &hints, &ai_list);
    if (n) {
        fprintf(stderr, "%s: getaddrinfo failed: %s\n",
                progname, gai_strerror(n));
        return -1;
    }

    for (ai = ai_list; ai; ai = ai->ai_next) {
        fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (fd < 0) {
            continue;
        }

        setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
        if (bind(fd, ai->ai_addr, ai->ai_addrlen) == 0) {
            break;
        }
        close(fd);
    }

    freeaddrinfo(ai_list);

    if (ai == NULL) {
        fprintf(stderr, "%s: bind failed for port %s\n", progname, port);
        return -1;
    }

    return fd;
}

/*
 * Close a udp socket. This function trivially calls close() on
 * POSIX systems, but might be more complicated on other systems.
 */

static int
udp_close(int fd)
{
    return close(fd);
}

/*
 * Implement the daytime protocol, loosely modeled after RFC 867.
 */

static void
udp_daytime(int fd)
{
    char message[128];
    char host[NI_MAXHOST];
    char serv[NI_MAXSERV];
    struct sockaddr_storage from;
    socklen_t fromlen = sizeof(from);
    int n;

    n = recvfrom(fd, message, sizeof(message), 0,
                 (struct sockaddr *) &from, &fromlen);
    if (n == -1) {
        syslog(LOG_ERR, "recvfrom failed: %s", strerror(errno));
        return;
    }

    n = getnameinfo((struct sockaddr *) &from, fromlen,
                    host, sizeof(host), serv, sizeof(serv),
                    NI_NUMERICHOST | NI_DGRAM);
    if (n) {
        syslog(LOG_ERR, "getnameinfo failed: %s", gai_strerror(n));
    } else {
        syslog(LOG_DEBUG, "request from %s:%s", host, serv);
    }

    daytime(message, sizeof(message));

    n = sendto(fd, message, strlen(message), 0,
               (struct sockaddr *) &from, fromlen);
    if (n == -1) {
        syslog(LOG_ERR, "sendto failed: %s", strerror(errno));
        return;
    }
}

int
main(int argc, char **argv)
{
    int ufd;

    if (argc != 2) {
        fprintf(stderr, "usage: %s port\n", progname);
        exit(EXIT_FAILURE);
    }

    openlog(progname, LOG_PID, LOG_DAEMON);

    ufd = udp_open(argv[1]);

    while (ufd != -1) {
        udp_daytime(ufd);
    }

    udp_close(ufd);

    closelog();

    return EXIT_SUCCESS;
}
```

B.5 Multiplexing

The examples discussed so far all had the property that the server or the client could block (freeze) in case of some communication errors. For example, the connection oriented server block incoming requests until a client has been served. One approach to address this deficiency is to use threads which of course requires thread-safe libraries. The other alternative is to avoid calling blocking functions by first checking whether a socket can be read or written.

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

typedef ... fd_set;

FD_ZERO(fd_set *set);
FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);

int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

int pselect(int n, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timespec *timeout,
            sigset_t sigmask);
```

The functions `select()` and `pselect()` can be used to test whether socket descriptors are "ready" so that a subsequent socket library call does not block. The `select()` call distinguishes three sets of socket descriptors:

1. The set `readfds` contains descriptors which will be watched to see if a subsequent read operation will not block.
2. The set `writefds` contains descriptors which will be watched to see if a subsequent write operation will not block.
3. The set `exceptfds` contains the descriptors which will be watched for excetions.

The macros `FD_ZERO()`, `FD_SET()`, `FD_CLR()` and `FD_ISSET()` can be used to manipulate the sets. The `timeout` is an upper bound on the amount of time elapsed before the select function returns. The parameter `n` contains the highest-numbered file descriptor in any of the three sets plus 1.

The function `pselect()` allows the correct handling of situations where a program wants to wait for socket descriptors as well as software signals.

Example 7 *The following source code combines the connection-less server with the connection-oriented server. The main loop uses the `select()` function to wait for incoming requests.*

```
/*
 * daytimed3/daytimed.c --
 *
 * A simple UDP and TCP over IPv4/IPv6 daytime server. The server
 * waits for incoming messages and sends a daytime string as a
 * reaction to received message. This version uses select() to handle
 * incoming TCP and UDP requests.
 */

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <syslog.h>
#include <string.h>
#include <time.h>

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

```
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/select.h>

static const char *progname = "daytimed";

/*
 * Store the current date and time of the day using the local timezone
 * in the given buffer of the indicated size. Set the buffer to a zero
 * length string in case of errors.
 */

static void
daytime(char *buffer, size_t size)
{
    time_t ticks;
    struct tm *tm;

    ticks = time(NULL);
    tm = localtime(&ticks);
    if (tm == NULL) {
        buffer[0] = '\0';
        syslog(LOG_ERR, "localtime failed");
        return;
    }
    strftime(buffer, size, "%F %T\r\n", tm);
}

/*
 * Create a named UDP endpoint. First get the list of potential
 * network layter addresses and transport layer port numbers. Iterate
 * through the returned address list until an attempt to create a UDP
 * endpoint is successful (or no other alternative exists).
 */

static int
udp_open(char *port)
{
    struct addrinfo hints, *ai_list, *ai;
    int n, fd = 0, on = 1;

    memset(&hints, 0, sizeof(hints));
    hints.ai_flags = AI_PASSIVE;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;

    n = getaddrinfo(NULL, port, &hints, &ai_list);
    if (n) {
        fprintf(stderr, "%s: getaddrinfo failed: %s\n",
                progname, gai_strerror(n));
        return -1;
    }

    for (ai = ai_list; ai; ai = ai->ai_next) {
        fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (fd < 0) {
            continue;
        }

        setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
        if (bind(fd, ai->ai_addr, ai->ai_addrlen) == 0) {
            break;
        }
        close(fd);
    }

    freeaddrinfo(ai_list);

    if (ai == NULL) {
        fprintf(stderr, "%s: bind failed for port %s\n", progname, port);
        return -1;
    }

    return fd;
}

/*
 * Close a udp socket. This function trivially calls close() on
 * POSIX systems, but might be more complicated on other systems.
 */

static int
udp_close(int fd)
{
    return close(fd);
}

/*
 * Implement the daytime protocol, loosely modeled after RFC 867.
 */

static void
udp_daytime(int fd)
{
    char message[128];
    char host[NI_MAXHOST];
    char serv[NI_MAXSERV];
    struct sockaddr_storage from;
    socklen_t fromlen = sizeof(from);
    int n;

    n = recvfrom(fd, message, sizeof(message), 0,
                 (struct sockaddr *) &from, &fromlen);
    if (n == -1) {
        syslog(LOG_ERR, "recvfrom failed: %s", strerror(errno));
        return;
    }
    n = getnameinfo((struct sockaddr *) &from, fromlen,
                    host, sizeof(host), serv, sizeof(serv),
                    NI_NUMERICHOST | NI_DGRAM);
    if (n) {
        syslog(LOG_ERR, "getnameinfo failed: %s", gai_strerror(n));
    } else {
        syslog(LOG_DEBUG, "request from %s:%s", host, serv);
    }

    daytime(message, sizeof(message));

    n = sendto(fd, message, strlen(message), 0,
               (struct sockaddr *) &from, fromlen);
    if (n == -1) {
        syslog(LOG_ERR, "sendto failed: %s", strerror(errno));
        return;
    }
}

/*
 * Create a listening TCP endpoint. First get the list of potential
 * network layter addresses and transport layer port numbers. Iterate
 * through the returned address list until an attempt to create a
 * listening TCP endpoint is successful (or no other alternative
 * exists).
 */

static int
tcp_listen(char *port)
{
    struct addrinfo hints, *ai_list, *ai;
    int n, fd = 0, on = 1;

    memset(&hints, 0, sizeof(hints));
    hints.ai_flags = AI_PASSIVE;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    n = getaddrinfo(NULL, port, &hints, &ai_list);
    if (n) {
        fprintf(stderr, "%s: getaddrinfo failed: %s\n",
                progname, gai_strerror(n));
        return -1;
    }

    for (ai = ai_list; ai; ai = ai->ai_next) {
        fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (fd < 0) {
            continue;
        }

        setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
        if (bind(fd, ai->ai_addr, ai->ai_addrlen) == 0) {
            break;
        }
        close(fd);
    }

    freeaddrinfo(ai_list);

    if (ai == NULL) {
        fprintf(stderr, "%s: bind failed for port %s\n",
                progname, port);
        return -1;
    }

    if (listen(fd, 42) < 0) {
        fprintf(stderr, "%s: listen failed: %s\n",
                progname, strerror(errno));
        return -1;
    }

    return fd;
}

/*
 * Accept a new TCP connection and write a message about who was
 * accepted to the system log.
 */

static int
tcp_accept(int listen)
{
    struct sockaddr_storage ss;
    socklen_t ss_len = sizeof(ss);
    char host[NI_MAXHOST];
    char serv[NI_MAXSERV];
    int n, fd;

    fd = accept(listen, (struct sockaddr *) &ss, &ss_len);
    if (fd == -1) {
        syslog(LOG_ERR, "accept failed: %s", strerror(errno));
        return -1;
    }

    n = getnameinfo((struct sockaddr *) &ss, ss_len,
                    host, sizeof(host), serv, sizeof(serv),
                    NI_NUMERICHOST);
    if (n) {
        syslog(LOG_ERR, "getnameinfo failed: %s", gai_strerror(n));
    } else {
        syslog(LOG_DEBUG, "connection from %s:%s", host, serv);
    }
    return fd;
}

/*
 * Close a TCP connection. This function trivially calls close() on
 * POSIX systems, but might be more complicated on other systems.
 */

static int
tcp_close(int fd)
{
    return close(fd);
}

/*
 * Implement the daytime protocol, loosely modeled after RFC 867.
 */

static void
tcp_daytime(int listenfd)
{
    size_t n;
    int client;
    char message[128];

    client = tcp_accept(listenfd);
    if (client == -1) {
        return;
    }

    daytime(message, sizeof(message));

    n = write(client, message, strlen(message));
    if (n != strlen(message)) {
        syslog(LOG_ERR, "write failed");
        return;
    }

    tcp_close(client);
}

int
main(int argc, char **argv)
{
    int tfd, ufd;
    fd_set fdset;

    if (argc != 2) {
        fprintf(stderr, "usage: %s port\n", progname);
        exit(EXIT_FAILURE);
    }

    openlog(progname, LOG_PID, LOG_DAEMON);

    ufd = udp_open(argv[1]);
    tfd = tcp_listen(argv[1]);

    while (ufd != -1 || tfd != -1) {
        FD_ZERO(&fdset);
        if (ufd > -1) FD_SET(ufd, &fdset);
        if (tfd > -1) FD_SET(tfd, &fdset);

        if (select(1 + (ufd > tfd ? ufd : tfd),
                   &fdset, NULL, NULL, NULL) == -1) {
            fprintf(stderr, "%s: select failed: %s\n",
                    progname, strerror(errno));
            return EXIT_FAILURE;
        }

        if (tfd > -1 && FD_ISSET(tfd, &fdset)) {
            tcp_daytime(tfd);
        }
        if (ufd > -1 && FD_ISSET(ufd, &fdset)) {
            udp_daytime(ufd);
        }
    }

    if (ufd != -1) udp_close(ufd);
    if (tfd != -1) tcp_close(tfd);

    closelog();

    return EXIT_SUCCESS;
}
```