EE3233 Systems Programming for Engrs

Reference: M. Kerrisk, The Linux Programming Interface

Lecture 18

Sockets

ELECTRICAL & COMPUTER ENGINEERING

Overview

- A method of IPC that exchanges data between applications either on the same host or on different host connected via a network
- Applications communicate using sockets as follows:
  - Each application creates a socket (apparatus) that allows communication, and both applications require one
  - The server binds its socket to a well-known address so that clients can locate it

Socket System Calls

- ***socket()***
  - creates a new socket
- ***bind()***
  - binds a socket to an address
- ***listen()***
  - allows a stream to accept incoming connections from other sockets
- ***accept()***
  - accepts a connection from a peer application
- ***connect()***
  - establishes a connection with another socket

socket()

```
fd = socket (domain, type, protocol);
```

- Communication domains, which determines:
  - method of identifying a socket (format of a socket "address")
  - range of communication (on the same host or different hosts)
- Modern OS supports following domains:
  - **AF_UNIX**: allows communication between applications on the same host (address structure: *sockaddr_un*)
  - **AF_INET**: allows communication between applications running on hosts connected via an Internet Protocol version 4 (IPv4) network (address structure: *sockaddr_in*)
  - **AF_INET6**: for IPv6 (address structure: *sockaddr_in6*)

socket()

fd = socket (domain, type, protocol);

- Socket type
  - stream (**SOCK_STREAM**)
    - ① Reliable: the transmitted data will arrive at the receiving application exactly as it was transmitted by sender
    - ② Bidirectional: transmitted in either direction
    - ③ Byte-stream: there is no concept of message boundaries
    - ④ Connection-Oriented: operates in connected pair

    : TCP
  - datagram (**SOCK_DGRAM**)
    - ① Data transmission is not reliable
    - ② Message may arrive out-of-order
    - ③ Message boundaries are preserved
    - ④ Connection-Less

    : UDP

socket()

```
fd = socket (domain, type, protocol);
```

- protocol values are always specified as 0 for the socket types that we describe

bind()

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

- *sockfd*
  - file descriptor obtained from a previous call to *socket()*
- *addr*
  - a pointer to a structure specifying the address to which this socket is to be bound
- *addrlen*
  - size of the address structure

bind()

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

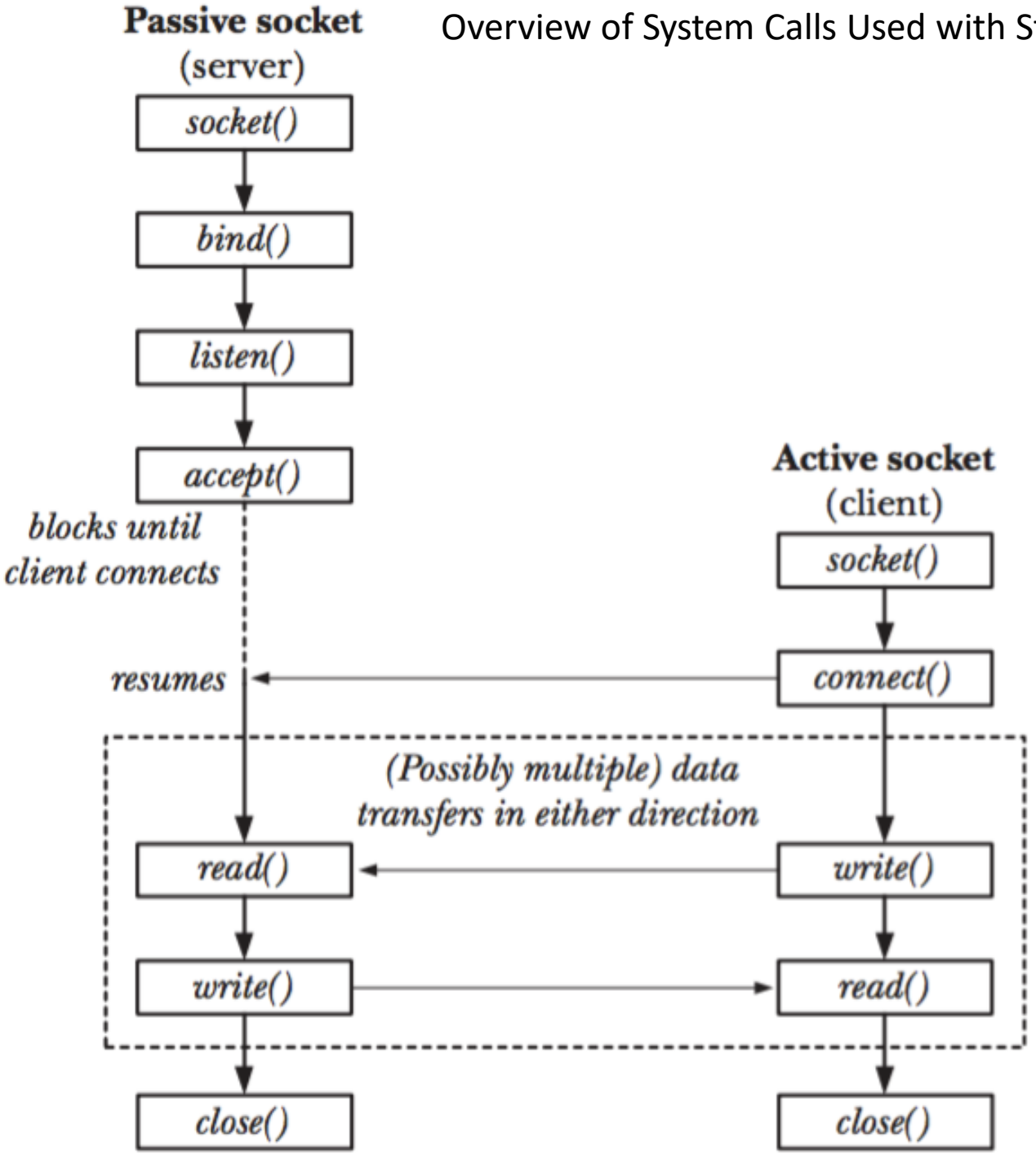
- Each socket domain uses a different address format
  - UNIX domain sockets use pathnames
  - Internet domain sockets use IP + port
- socket API defines a generic address structure

```
struct sockaddr {
    sa_family_t  sa_family;     /* Address family */
    char         sa_data[14];   /* Socket address */
};
```

Stream Sockets

- analogous to a telephone call:
  - ***socket()*** is installing a telephone(socket). For two persons(applications), each of telephone must be installed
  - ***bind()*** is having a known telephone number
  - ***listen()*** is ensuring that our telephone is turned on so that people can call us
  - ***accept()*** is picking up the phone when it rings
  - ***connect()*** is dialing someone's telephone number

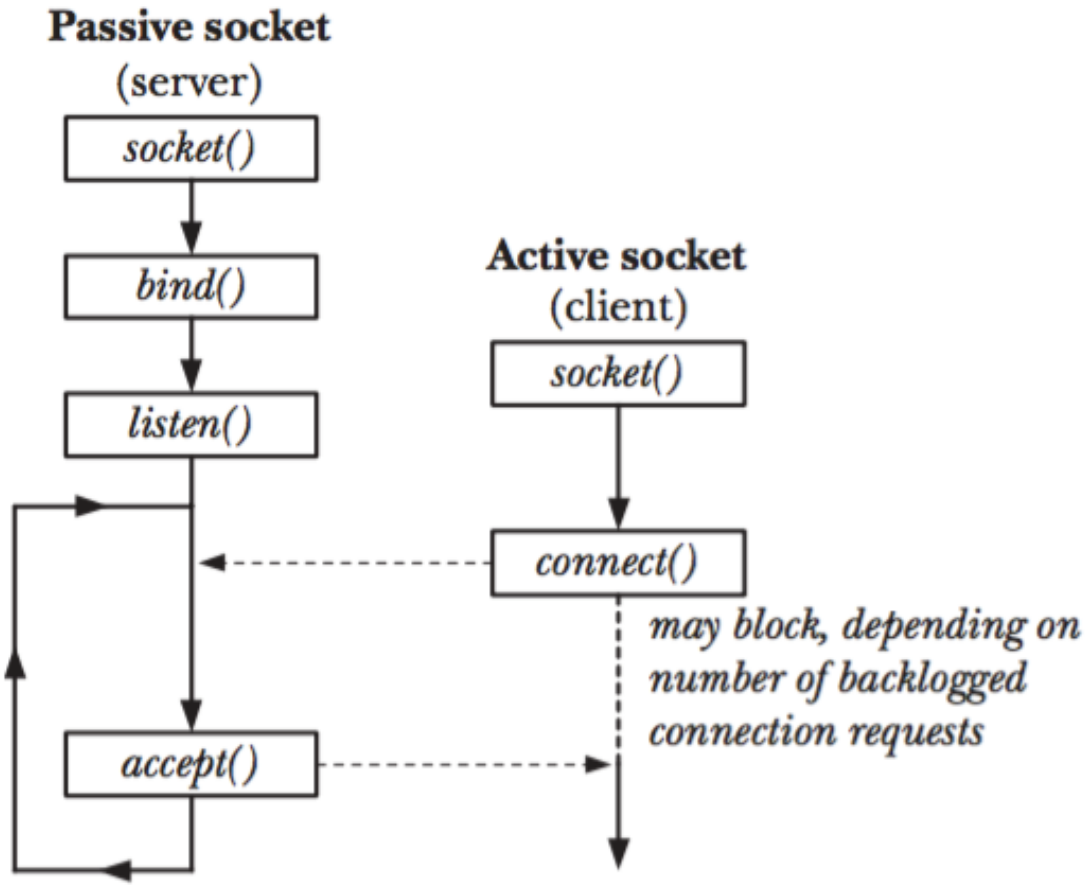
Overview of System Calls Used with Stream Sockets



listen()

int **listen** (int *sockfd*, int *backlog*);

- notifying kernel of its willingness to accept incoming connections
- *sockfd*: a file descriptor
- *backlog:* number of pending connections
  - Connection requests up to this limit succeed immediately



accept()

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

returns *file descriptor* on success, or -1 on error

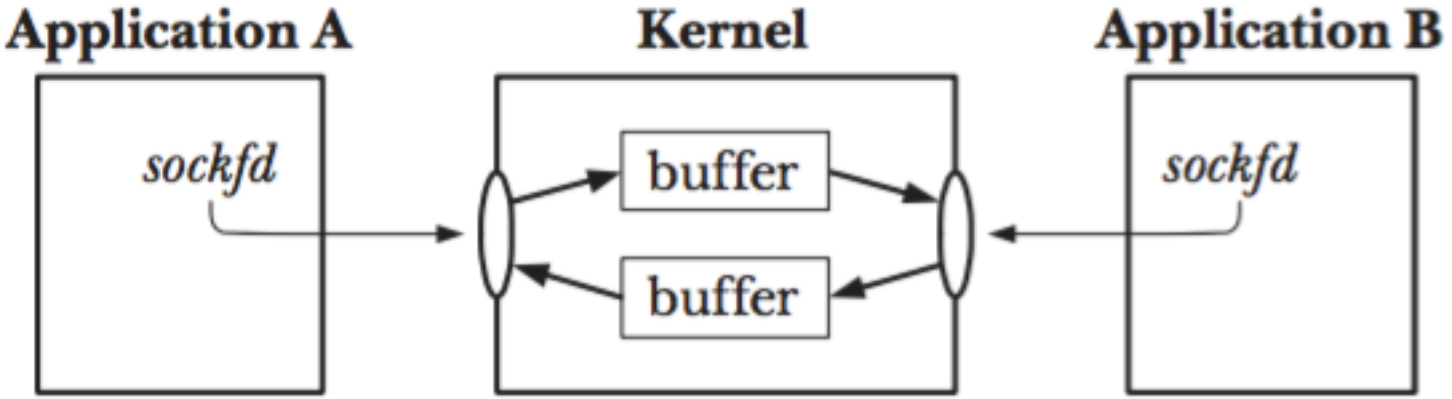
- *sockfd*: a file descriptor
- *addr:* returned address of the peer socket
- *addrlen*: that must be initialized (0) (prior to call) to the size of the buffer pointed to by *addr*

connect()

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

- If *connect()* fails and we wish to reattempt the connection, then close the socket, create a new socket, and reattempt the connection with the new socket

I/O on Stream Sockets

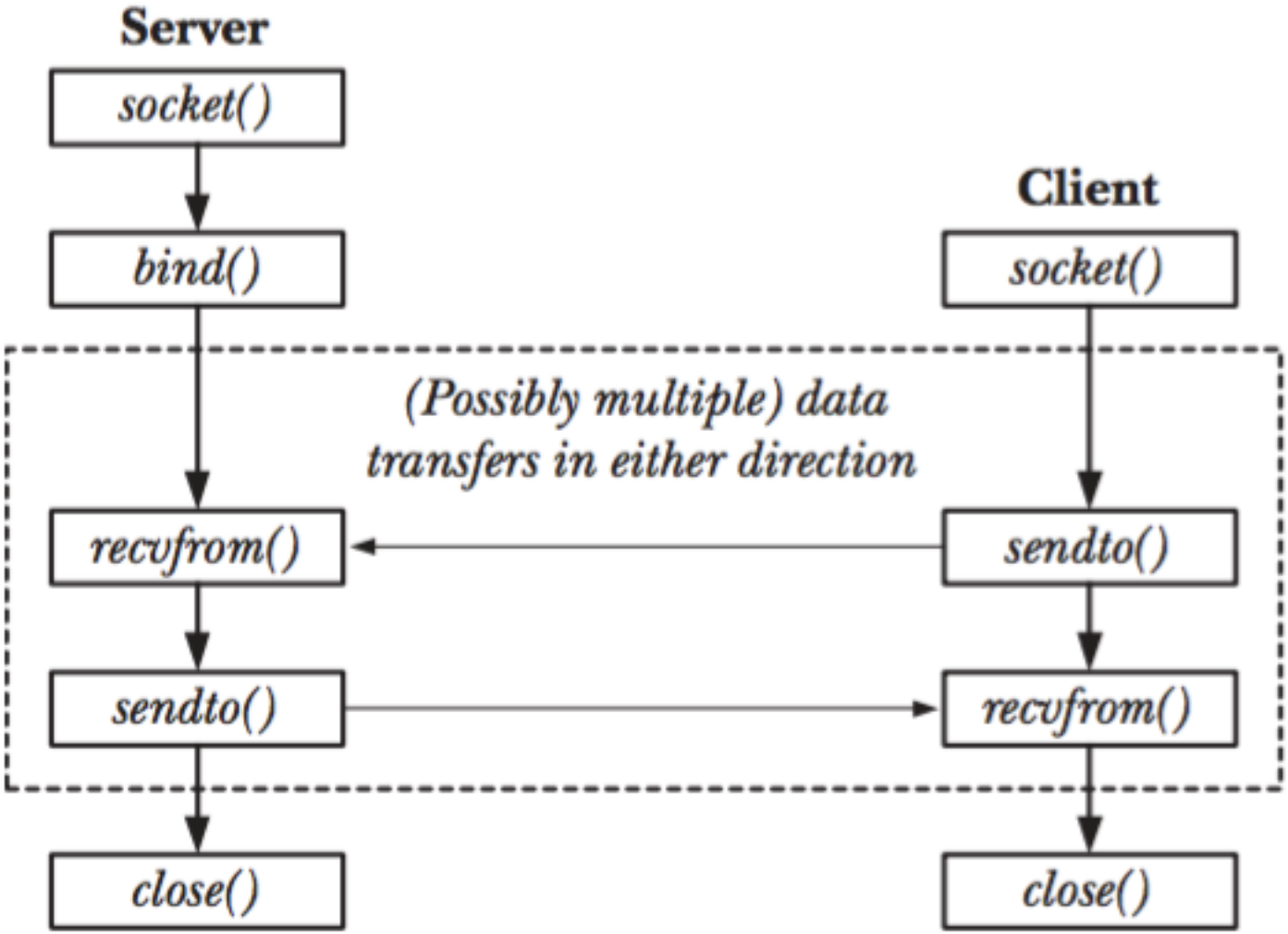


- To perform I/O, we use the *read()* and *write()* system calls
  - Since sockets are bidirectional, both calls may be used on each end of the connection
- A socket may be closed using the *close()* system call or as a consequence of the application terminating

Datagram Sockets

- Explained by analogy with the postal system:
  - ① *socket()* is setting up a mailbox
  - ② *bind()* is associating the mailbox with my address
    - a client sends datagrams (letters) to that address
  - ③ *sendto()* is putting the recipient's address on a letter and posting it
  - ④ *recvfrom()* is receiving a datagram (letter)

Overview of System Calls used with Datagram Sockets



Exchanging Datagrams: recvfrom(), sendto()

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

ssize_t recvfrom(int sockfd, void *buffer, size_t length, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Returns number of bytes received, 0 on EOF, or −1 on error

```
ssize_t sendto(int sockfd, const void *buffer, size_t length, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

Returns number of bytes sent, or −1 on error

- The first three arguments are the same as for *read()* and *write()*
- *flags*: a bit mask controlling socket-specific I/O features (will come to this)
- *src_addrs, dest_addr*: peer address

UNIX Domain Sockets

- communication between processes on the same host system
- UNIX Domain Socket Addresses:

```
struct sockaddr_un {
    sa_family_t  sun_family;
    char  sun_path[108];
};
```

Binding a UNIX Domain Socket

- initialize a *sockaddr_un* structure, and then pass a (cast) pointer to this structure as the *addr* argument to *bind()*

```
const char *SOCKNAME = "/tmp/mysock";
int sfd;
struct sockaddr_un addr;

sfd = socket(AF_UNIX, SOCK_STREAM, 0);          /* Create socket */
if (sfd == -1)
    errExit("socket");

memset(&addr, 0, sizeof(struct sockaddr_un));   /* Clear structure */
addr.sun_family = AF_UNIX;                      /* UNIX domain address */
strncpy(addr.sun_path, SOCKNAME, sizeof(addr.sun_path) - 1);

if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
    errExit("bind");
```

Binding a UNIX Domain Socket

- memset()
  - ensures that all of the structure fields have the value 0
- strncpy()
  - copies "**SOCKNAME**" to *sun_path*
  - specifies one less than the size of the *sun_path* field ensuring that this field always has a terminating null byte

Binding a UNIX Domain Socket

- Usual to bind a socket to an *absolute* pathname
- A pathname can be bound to *only one* socket
- can't use *open()* to open a socket
- When the socket is no longer required, its pathname entry can be removed using *unlink()* or *remove()*

Stream Sockets in the UNIX Domain

- A simple client-server application that uses stream sockets in the UNIX domain
  - client: connects to the server, and uses the connection to transfer data from its standard input to the server
  - server: accepts client connections, and transfers all data sent on the connection by the client to standard output

```
#include <sys/un.h>
#include <sys/socket.h>
#include "tlpi_hdr.h"

#define SV_SOCK_PATH "/tmp/us_xfr"

#define BUF_SIZE 100
```

sockets/us_xfr_sv.c

```
#include "us_xfr.h"

#define BACKLOG 5

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        errExit("socket");

    /* Construct server socket address, bind socket to it,
       and make this a listening socket */

    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);

    if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");

    if (listen(sfd, BACKLOG) == -1)
        errExit("listen");
```

/* Remove any existing file with the same pathname as that to which we bind the socket */

```
    for (;;) {          /* Handle client connections iteratively */

        /* Accept a connection. The connection is returned on a new
           socket, 'cfd'; the listening socket ('sfd') remains open
           and can be used to accept further connections. */

        cfd = accept(sfd, NULL, NULL);
        if (cfd == -1)
            errExit("accept");

        /* Transfer data from connected socket to stdout until EOF */

        while ((numRead = read(cfd, buf, BUF_SIZE)) > 0)
            if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("partial/failed write");

        if (numRead == -1)
            errExit("read");


        if (close(cfd) == -1)
            errMsg("close");
    }
}
```

sockets/us_xfr_cl.c

```
#include "us_xfr.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);       /* Create client socket */
    if (sfd == -1)
        errExit("socket");

    /* Construct server address, and make the connection */

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);

    if (connect(sfd, (struct sockaddr *) &addr,
                sizeof(struct sockaddr_un)) == -1)
        errExit("connect");

    /* Copy stdin to socket */

    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
        if (write(sfd, buf, numRead) != numRead)
            fatal("partial/failed write");

    if (numRead == -1)
        errExit("read");

    exit(EXIT_SUCCESS);         /* Closes our socket; server sees EOF */
}
```

Example Code

```
$ ./us_xfr_sv > b &
[1] 9866
$ ls -lF /tmp/us_xfr                          Examine socket file with ls
srwxr-xr-x    1 mtk      users          0 Jul 18 10:48 /tmp/us_xfr=
```

We then create a test file to be used as input for the client, and run the client:

```
$ cat *.c > a
$ ./us_xfr_cl < a                             Client takes input from test file
```

At this point, the child has completed. Now we terminate the server as well, and check that the server's output matches the client's input:

```
$ kill %1                                     Terminate server
 [1]+  Terminated    ./us_xfr_sv >b           Shell sees server's termination
$ diff a b
$
```

Datagram Sockets in the UNIX Domain

- On Linux, you can send quite large datagram
  - Controlled via the SO_SNDBUF and various /proc files
  - Some other UNIX impose lower limits, 2048 bytes

sockets/ud_ucase.h

```
#include <sys/un.h>
#include <sys/socket.h>
#include <ctype.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 10             /* Maximum size of messages exchanged
                                   between client to server */

#define SV_SOCK_PATH "/tmp/ud_ucase"
```

sockets/ud_ucase_sv.c

```
#include "ud_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un svaddr, claddr;
    int sfd, j;
    ssize_t numBytes;
    socklen_t len;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_DGRAM, 0);       /* Create server socket */
    if (sfd == -1)
        errExit("socket");

    /* Construct well-known address and bind server socket to it */

    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);

    memset(&svaddr, 0, sizeof(struct sockaddr_un));
    svaddr.sun_family = AF_UNIX;
    strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(svaddr.sun_path) - 1);

    if (bind(sfd, (struct sockaddr *) &svaddr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");
```

```
    /* Receive messages, convert to uppercase, and return to client */

    for (;;) {
        len = sizeof(struct sockaddr_un);
        numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                            (struct sockaddr *) &claddr, &len);
        if (numBytes == -1)
            errExit("recvfrom");

        printf("Server received %ld bytes from %s\n", (long) numBytes,
                claddr.sun_path);

        for (j = 0; j < numBytes; j++)
            buf[j] = toupper((unsigned char) buf[j]);

        if (sendto(sfd, buf, numBytes, 0, (struct sockaddr *) &claddr, len) !=
                numBytes)
            fatal("sendto");
    }
}
```

sockets/ud_ucase_cl.c

```
#include "ud_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un svaddr, claddr;
    int sfd, j;
    size_t msgLen;
    ssize_t numBytes;
    char resp[BUF_SIZE];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s msg...\n", argv[0]);

    /* Create client socket; bind to unique pathname (based on PID) */

    sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sfd == -1)
        errExit("socket");

    memset(&claddr, 0, sizeof(struct sockaddr_un));
    claddr.sun_family = AF_UNIX;
    snprintf(claddr.sun_path, sizeof(claddr.sun_path),
            "/tmp/ud_ucase_cl.%ld", (long) getpid());

    if (bind(sfd, (struct sockaddr *) &claddr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");
```

sockets/ud_ucase_cl.c

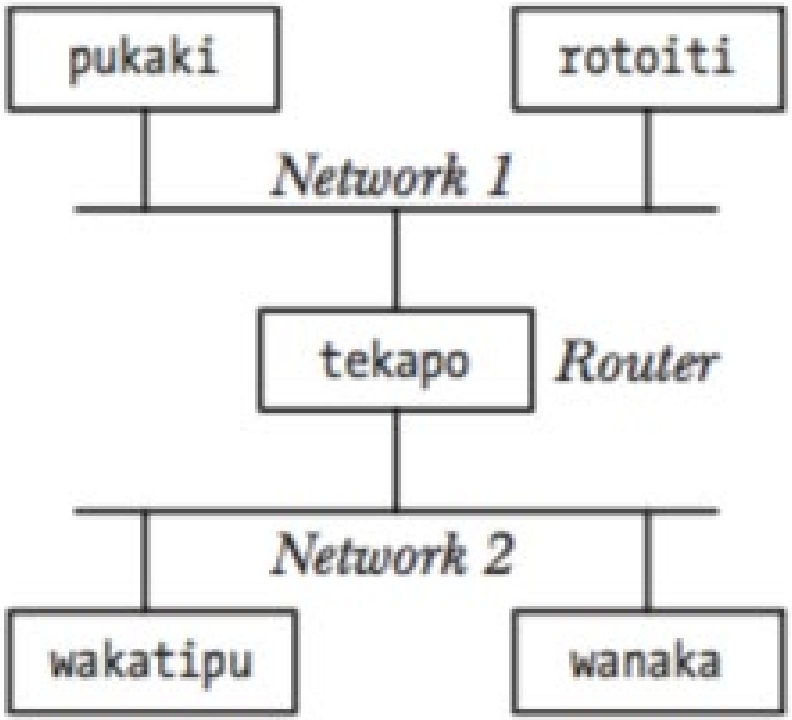
```
    /* Construct address of server */

    memset(&svaddr, 0, sizeof(struct sockaddr_un));
    svaddr.sun_family = AF_UNIX;
    strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(svaddr.sun_path) - 1);

    /* Send messages to server; echo responses on stdout */

    for (j = 1; j < argc; j++) {
        msgLen = strlen(argv[j]);       /* May be longer than BUF_SIZE */
        if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
                sizeof(struct sockaddr_un)) != msgLen)
            fatal("sendto");

        numBytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
        if (numBytes == -1)
            errExit("recvfrom");
        printf("Response %d: %.*s\n", j, (int) numBytes, resp);
    }

    remove(claddr.sun_path);            /* Remove client socket pathname */
    exit(EXIT_SUCCESS);
}
```

Results

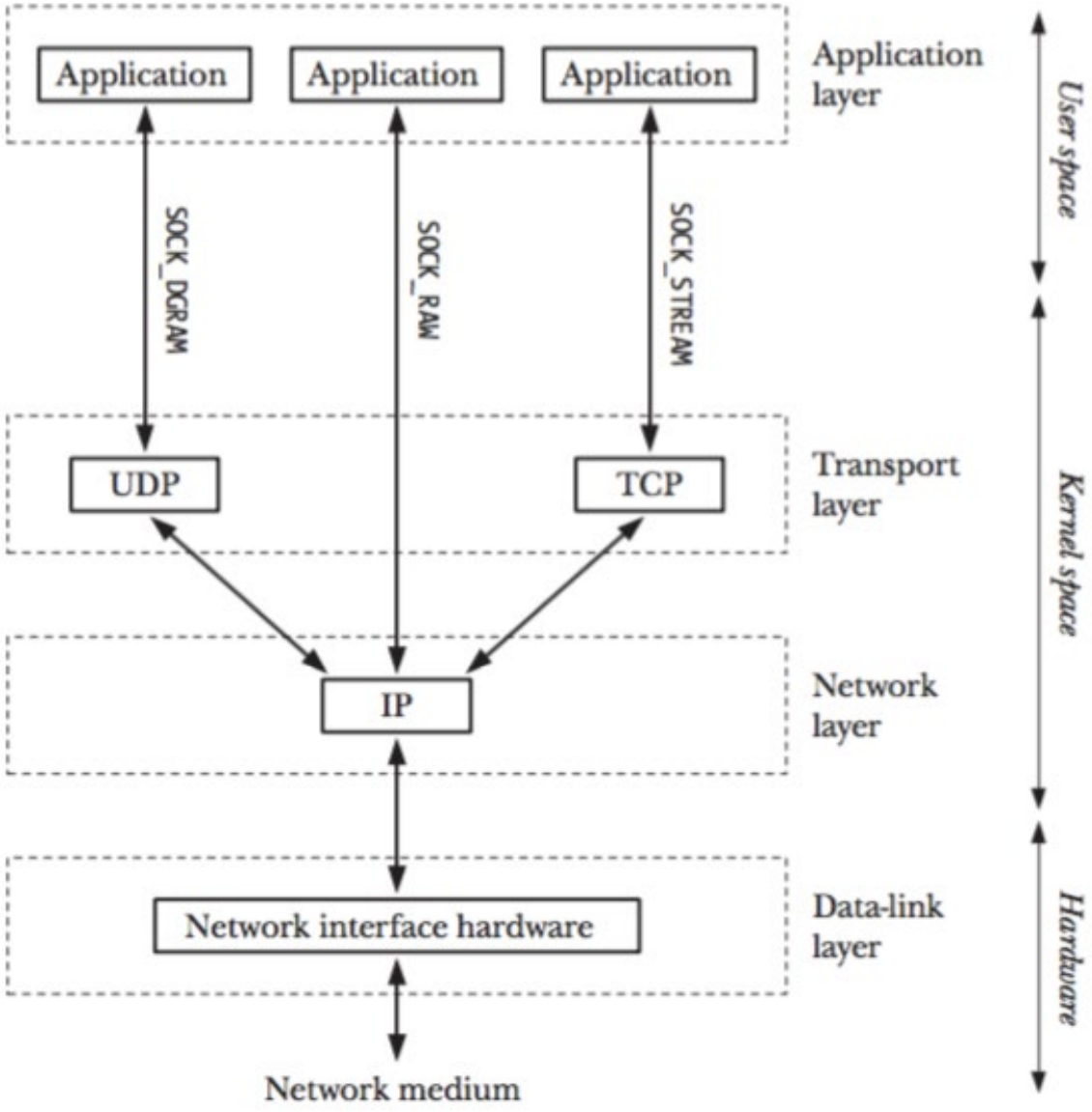
```
$ ./ud_ucase_sv &
[1] 20113
$ ./ud_ucase_cl hello world                    Send 2 messages to server
Server received 5 bytes from /tmp/ud_ucase_cl.20150
Response 1: HELLO
Server received 5 bytes from /tmp/ud_ucase_cl.20150
Response 2: WORLD
$ ./ud_ucase_cl 'long message'                 Send 1 longer message to server
Server received 10 bytes from /tmp/ud_ucase_cl.20151
Response 1: LONG MESSA
$ kill %1                                      Terminate server
```

Sockets: TCP/IP Networks

- TCP/IP is dominant protocol



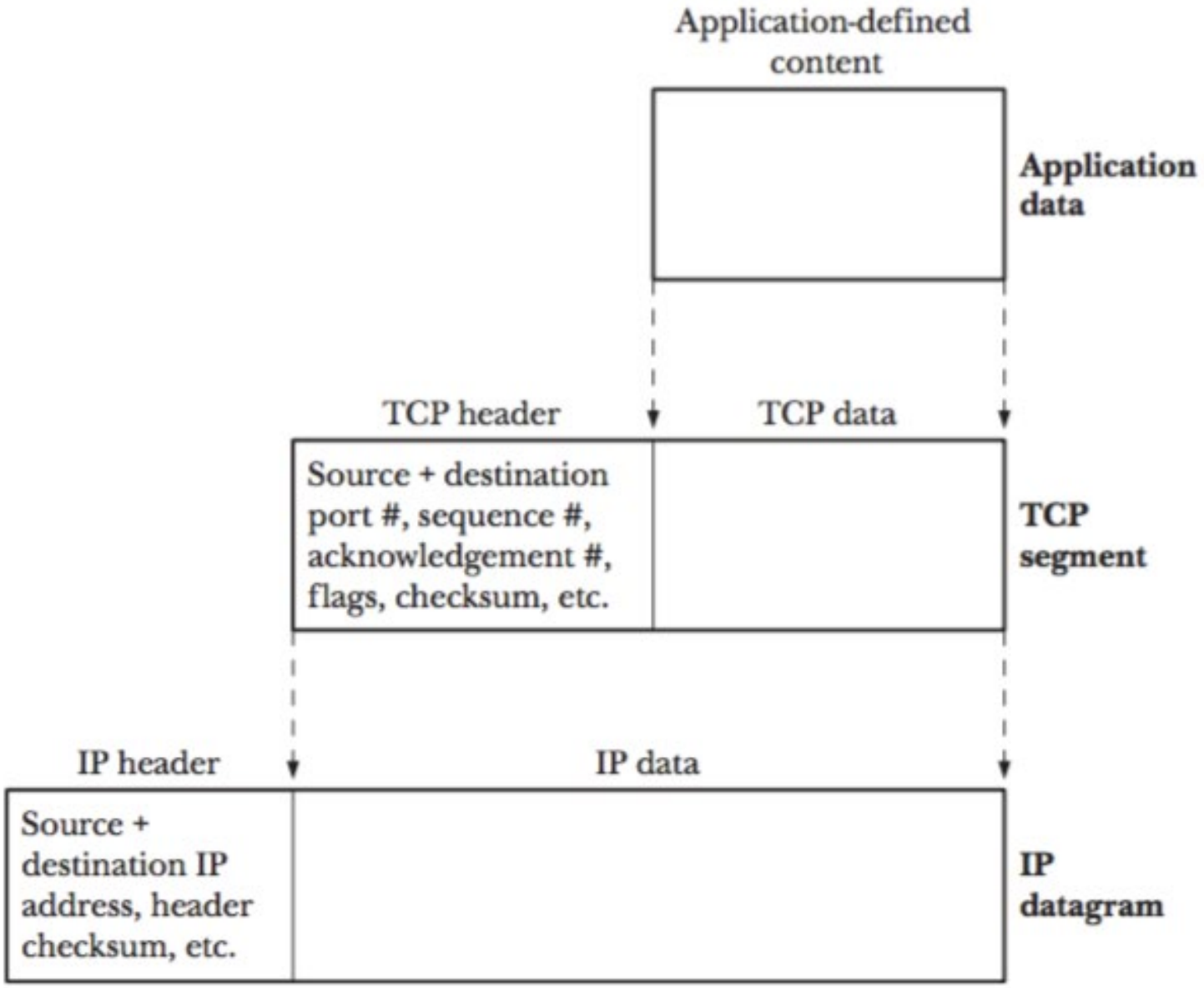
Networking Protocol and Layers



Encapsulation

- The lower layer makes no attempt to interpret information sent from the upper layer
  - and adds its own layer-specific header before passing the packet down to the next lower layer
- When data is passed up from a lower layer to a higher layer, a converse unpacking process takes place

Encapsulation



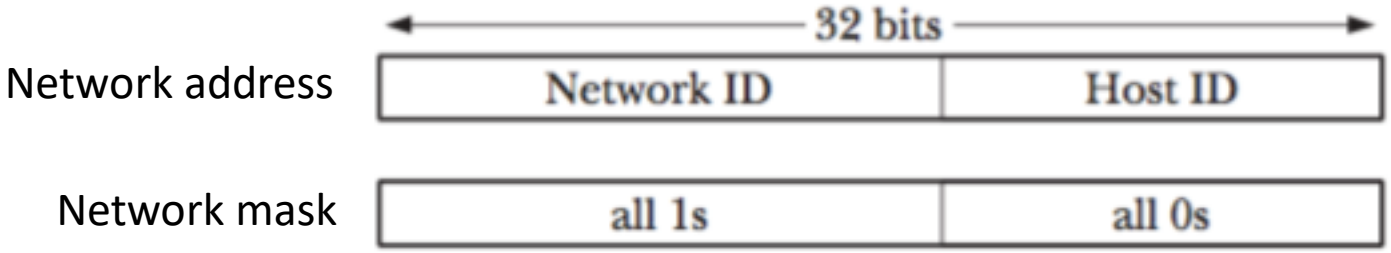
Data-Link Layer

- consists of device driver and H/W interface (network card)
- concerned with transferring data across a physical link in a network
- encapsulates datagrams into frames
- includes a header containing destination address and frame size
- performs error detection, retransmission, flow control
  - Some data-link layers split large network packets into multiple frames and reassemble them at the receiver

Network Layer: IP

- concerned with delivering packets from source to destination host
- Tasks
  - breaks data into fragments small enough for transmission
  - routs data across the internet
  - provides services to the transport layer
- IP transmits data in the form of datagram (packets)
- Header contains
  - address of the target host
  - originating address of the packet

IP Address



- Network ID
  - specifies the network on which a host resides
- Host ID
  - identifies the host within that network
- network mask
  - 1s indicates which part of the address contains the assigned network ID
  - 0s to assign unique host IDs on its network

  e.g., 204.152.189.0/24 : /24 indicates that network ID part of the assigned address consists of the leftmost 24 bits (network mask is 255.255.255.0)

IP Address

- Loopback address
  - special address 127.0.0.1 : conventionally assigned to the hostname localhost
  - A datagram sent to this address never actually reaches the network for testing client/server programs on the same host
  - **INADDR_LOOPBACK** is defined for this address
- Wildcard address
  - INADDR_ANY
  - useful for applications that bind Internet domain socket on multi-homed hosts (more than one network interface)

Transport Layer

- Two widely used protocols
  - UDP (User Datagram Protocol) : for datagram sockets
  - TCP (Transmission Control Protocol) : for stream sockets
- Task is to provide an end-to-end communication service to applications residing on different hosts
  - requires a method to differentiate the applications
  - this differentiation is provided by a 16-bit *port* number

Port Numbers

- Well-known port numbers
  - ssh (secure shell) : 22
  - HTTP : 80
  - HTTPS: 443
  - assigned 0 ~ 1023
  - Normally privileged (**CAP_NET_BIND_SERVICE**)
- IANA (Internet Assigned Numbers Authority) records registered ports
  - 1024 ~ 49151

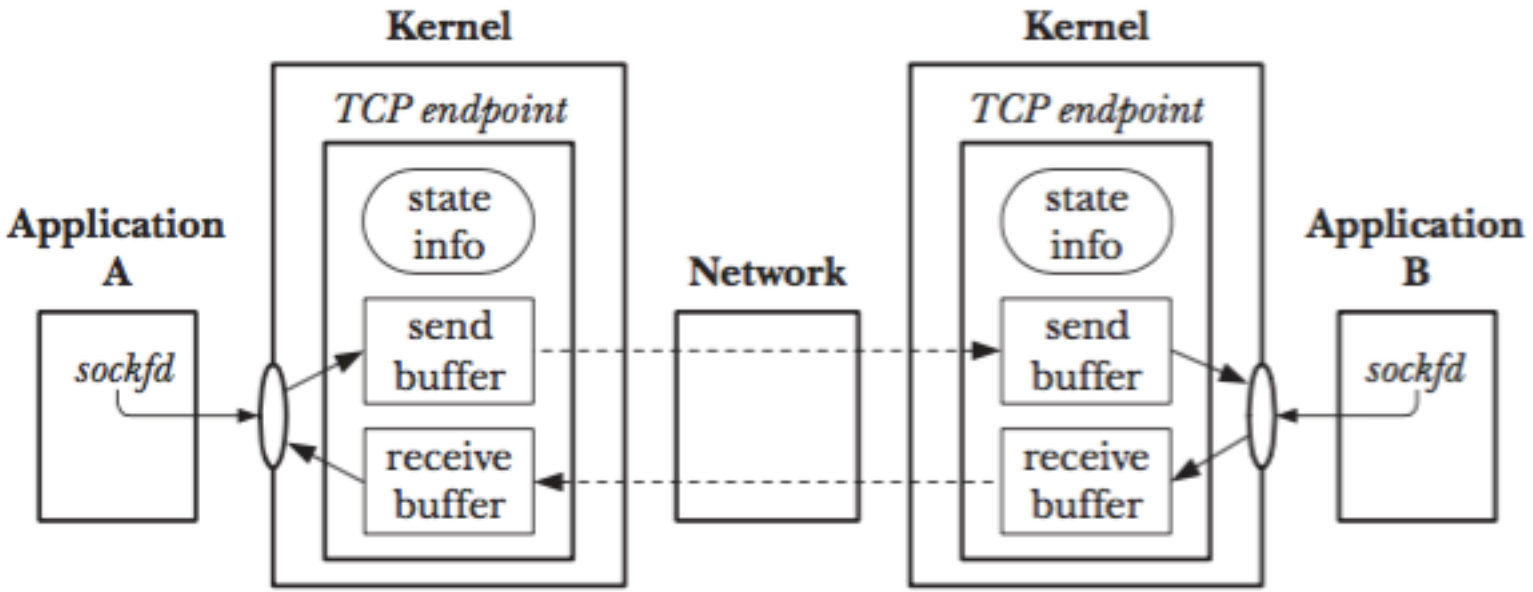
UDP

- adds two features
  - port number
  - data checksum (since connectionless)
- data checksum
  - 16 bits long
  - for error detection

TCP

- Acknowledgements
  - When a TCP segment arrives at its destination without errors, the receiving TCP sends a positive acknowledgement to the sender
- Sequencing
  - Each byte transmitted over a TCP is assigned a logical sequence number
  - This number indicates the position in data stream

TCP

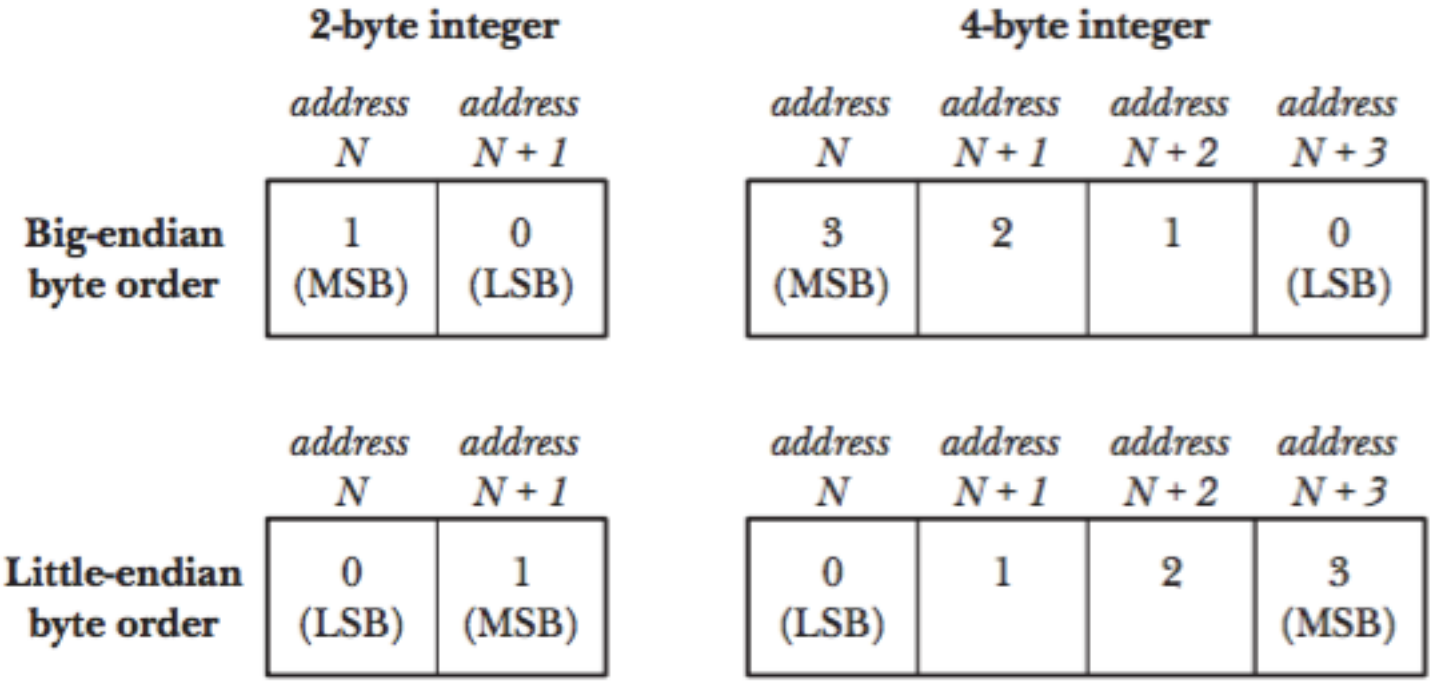

Kernel
TCP endpoint
state info
send buffer
receive buffer
Application A
sockfd
Network
Kernel
TCP endpoint
state info
send buffer
receive buffer
Application B
sockfd

Network Byte Order

- IP addresses and port numbers are integer values
- When passing these numbers across a network, different architectures store integer bytes differently
- Thus, standard ordering must be used
  - this ordering is called network byte order
  - can be stored in a *socket address structure*

Network Byte Order

- Big-endian: stores integers with the most significant byte first
- Little-endian: least significant byte first



Host Byte Order

- convention of the host machine
- should be converted to network byte order before storing them in socket address structures

htons(), htonl(), ntohs(), ntohl()

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

uint16_t htons(uint16_t host_uint16);
                                Returns host_uint16 converted to network byte order
uint32_t htonl(uint32_t host_uint32);
                                Returns host_uint32 converted to network byte order
uint16_t ntohs(uint16_t net_uint16);
                                Returns net_uint16 converted to host byte order
uint32_t ntohl(uint32_t net_uint32);
                                Returns net_uint32 converted to host byte order
```

- convert between host byte order and network byte order

readline()

```
#include "read_line.h"

ssize_t readLine(int fd, void *buffer, size_t n);
```

Returns number of bytes copied into *buffer* (excluding terminating null byte), or 0 on end-of-file, or −1 on error

- reads bytes from the file referred to by the file descriptor, *fd* until a newline is encountered

Reading data a line at a time

sockets/read_line.c

```
#include <unistd.h>
#include <errno.h>
#include "read_line.h"                  /* Declaration of readLine() */

ssize_t
readLine(int fd, void *buffer, size_t n)
{
    ssize_t numRead;                    /* # of bytes fetched by last read() */
    size_t totRead;                     /* Total bytes read so far */
    char *buf;
    char ch;

    if (n <= 0 || buffer == NULL) {
        errno = EINVAL;
        return -1;
    }

    buf = buffer;                       /* No pointer arithmetic on "void *" */

    totRead = 0;
```

```
    for (;;) {
        numRead = read(fd, &ch, 1);

        if (numRead == -1) {
            if (errno == EINTR)         /* Interrupted --> restart read() */
                continue;
            else
                return -1;              /* Some other error */

        } else if (numRead == 0) {      /* EOF */
            if (totRead == 0)           /* No bytes read; return 0 */
                return 0;
            else                        /* Some bytes read; add '\0' */
                break;

        } else {                        /* 'numRead' must be 1 if we get here */
            if (totRead < n - 1) {      /* Discard > (n - 1) bytes */
                totRead++;
                *buf++ = ch;
            }

            if (ch == '\n')
                break;
        }
    }

    *buf = '\0';
    return totRead;
}
```

Internet Socket Addresses

```
struct in_addr {                    /* IPv4 4-byte address */
    in_addr_t s_addr;               /* Unsigned 32-bit integer */
};

struct sockaddr_in {                /* IPv4 socket address */
    sa_family_t    sin_family;      /* Address family (AF_INET) */
    in_port_t      sin_port;        /* Port number */
    struct in_addr sin_addr;        /* IPv4 address */
    unsigned char  __pad[X];        /* Pad to size of 'sockaddr'
                                       structure (16 bytes) */
};
```

- IPv4 socket addresses: *struct sockaddr_in*
- *sin_family* field is always set to **AF_INET**

Internet Socket Addresses

```
struct in6_addr {                       /* IPv6 address structure */
    uint8_t s6_addr[16];                /* 16 bytes == 128 bits */
};

struct sockaddr_in6 {                   /* IPv6 socket address */
    sa_family_t sin6_family;            /* Address family (AF_INET6) */
    in_port_t   sin6_port;              /* Port number */
    uint32_t    sin6_flowinfo;          /* IPv6 flow information */
    struct in6_addr sin6_addr;          /* IPv6 address */
    uint32_t    sin6_scope_id;          /* Scope ID (new in kernel 2.4) */
};
```

- IPv6 socket addresses: *struct sockaddr_in6*
- IPv6 loopback address is (***::1***)

inet_pton(), inet_ntop()

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

int inet_pton(int domain, const char *src_str, void *addrptr);
```

Returns 1 on successful conversion, 0 if *src_str* is not in presentation format, or −1 on error

```
const char *inet_ntop(int domain, const void *addrptr, char *dst_str, size_t len);
```

Returns pointer to *dst_str* on success, or NULL on error

- converts between binary form and dotted-decimal notation
- *p* : presentation (human readable)
  - 204.111.122.114 (IPv4 dotted-decimal address)
  - ::1 (an IPv6 colon-separated hexadecimal address)
- *n* : network (binary form)

Example (**Datagram Sockets**)

- header file

sockets/i6d_ucase.h

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <ctype.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 10                     /* Maximum size of messages exchanged
                                           between client and server */

#define PORT_NUM 50002                  /* Server port number */
```

sockets/i6d_ucase_sv.c

```
#include "i6d_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_in6 svaddr, claddr;
    int sfd, j;
    ssize_t numBytes;
    socklen_t len;
    char buf[BUF_SIZE];
    char claddrStr[INET6_ADDRSTRLEN];

    sfd = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sfd == -1)
        errExit("socket");

    memset(&svaddr, 0, sizeof(struct sockaddr_in6));
    svaddr.sin6_family = AF_INET6;
    svaddr.sin6_addr = in6addr_any;                    /* Wildcard address */
    svaddr.sin6_port = htons(PORT_NUM);

    if (bind(sfd, (struct sockaddr *) &svaddr,
             sizeof(struct sockaddr_in6)) == -1)
        errExit("bind");
```

```
    /* Receive messages, convert to uppercase, and return to client */

    for (;;) {
        len = sizeof(struct sockaddr_in6);
        numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                            (struct sockaddr *) &claddr, &len);
        if (numBytes == -1)
            errExit("recvfrom");

        if (inet_ntop(AF_INET6, &claddr.sin6_addr, claddrStr,
                      INET6_ADDRSTRLEN) == NULL)
            printf("Couldn't convert client address to string\n");
        else
            printf("Server received %ld bytes from (%s, %u)\n",
                    (long) numBytes, claddrStr, ntohs(claddr.sin6_port));

        for (j = 0; j < numBytes; j++)
            buf[j] = toupper((unsigned char) buf[j]);

        if (sendto(sfd, buf, numBytes, 0, (struct sockaddr *) &claddr, len) !=
                numBytes)
            fatal("sendto");
    }
}
```

sockets/i6d_ucase_cl.c

```
#include "i6d_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_in6 svaddr;
    int sfd, j;
    size_t msgLen;
    ssize_t numBytes;
    char resp[BUF_SIZE];

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s host-address msg...\n", argv[0]);

    sfd = socket(AF_INET6, SOCK_DGRAM, 0);      /* Create client socket */
    if (sfd == -1)
        errExit("socket");

    memset(&svaddr, 0, sizeof(struct sockaddr_in6));
    svaddr.sin6_family = AF_INET6;
    svaddr.sin6_port = htons(PORT_NUM);
    if (inet_pton(AF_INET6, argv[1], &svaddr.sin6_addr) <= 0)
        fatal("inet_pton failed for address '%s'", argv[1]);
```

```
    /* Send messages to server; echo responses on stdout */

    for (j = 2; j < argc; j++) {
        msgLen = strlen(argv[j]);
        if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
                   sizeof(struct sockaddr_in6)) != msgLen)
            fatal("sendto");

        numBytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
        if (numBytes == -1)
            errExit("recvfrom");

        printf("Response %d: %.*s\n", j - 1, (int) numBytes, resp);
    }

    exit(EXIT_SUCCESS);
}
```

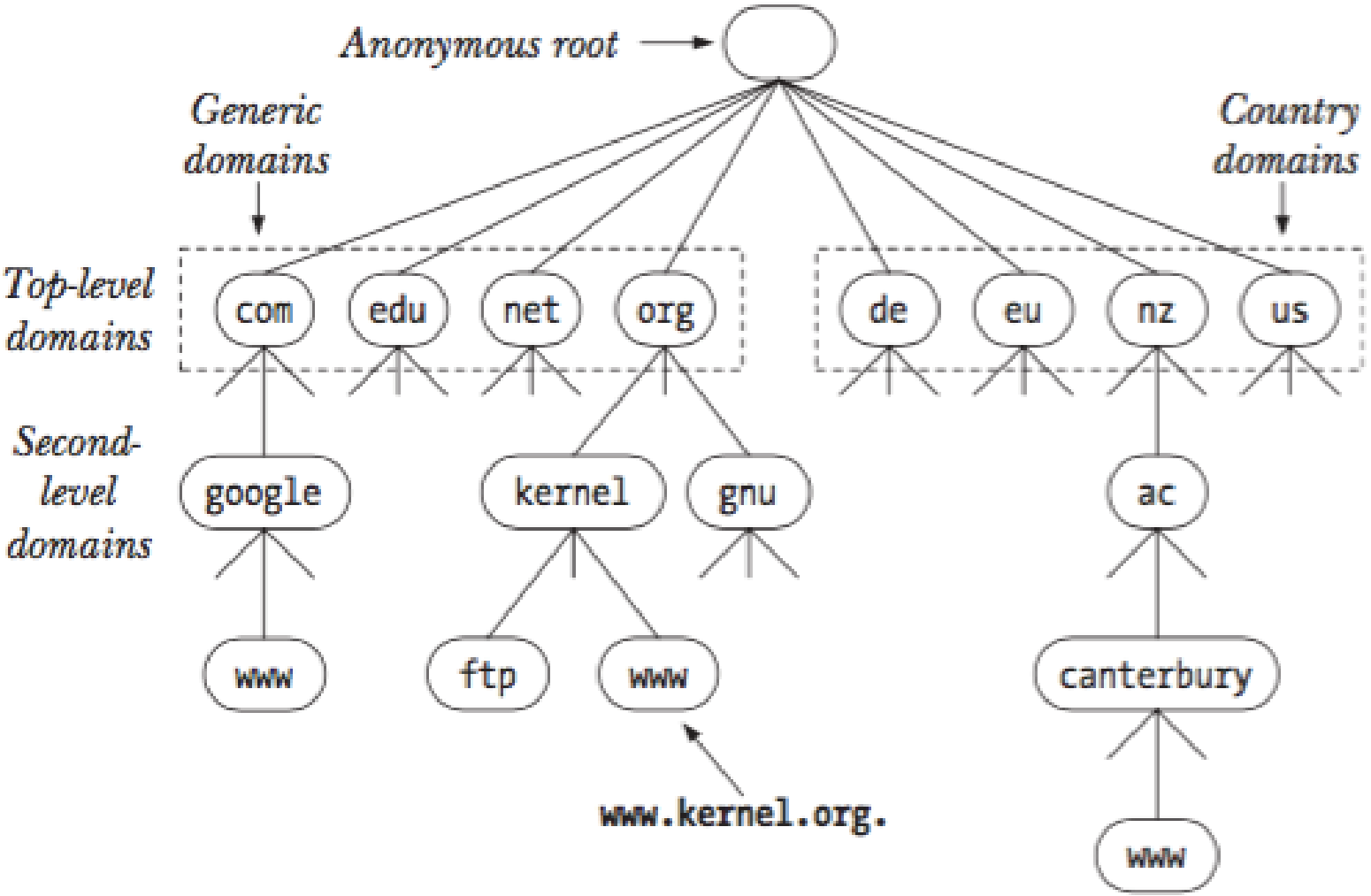
```
$ ./i6d_ucase_sv &
[1] 31047
$ ./i6d_ucase_cl ::1 ciao
Server received 4 bytes from (::1, 32770)
Response 1: CIAO
```

Send to server on local host

DNS

```
$ cat /etc/hosts
# IP-address   canonical hostname    [aliases]
127.0.0.1      localhost
```

A subset of DNS hierarchy



getaddrinfo()

- converts *host* (e.g., www.google.com) and *service* names (e.g., https) to *IP* and *port*
  - returns a list of socket address structures, each of which contains an IP address and port number

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

int getaddrinfo(const char *host, const char *service,
                const struct addrinfo *hints, struct addrinfo **result);
```

Returns 0 on success, or nonzero on error

- service: either service name or decimal port number
- hints: addrinfo structure
  - that specifies further criteria for selecting the socket address structures
- result: returns a list of structures

getaddrinfo()

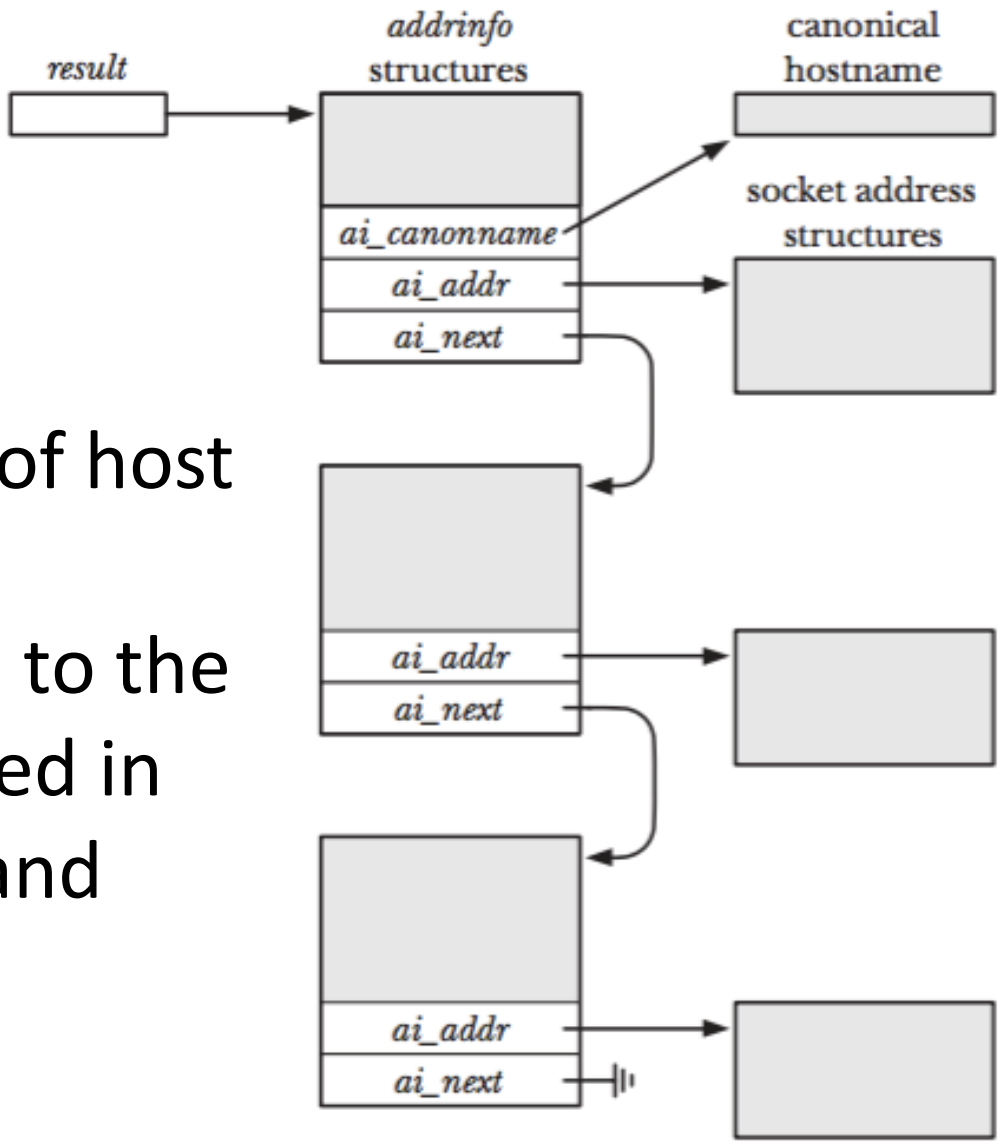
- When a program calls *getaddrinfo()* to obtain the IP address for a domain name, *getaddrinfo()* employs a suite of library functions that communicate with local DNS server
  - if this server can't supply the required information, then it communicates with other DNS servers

getaddrinfo()

```
struct addrinfo {
    int    ai_flags;          /* Input flags (AI_* constants) */
    int    ai_family;         /* Address family */
    int    ai_socktype;       /* Type: SOCK_STREAM, SOCK_DGRAM */
    int    ai_protocol;       /* Socket protocol */
    size_t ai_addrlen;        /* Size of structure pointed to by ai_addr */
    char  *ai_canonname;      /* Canonical name of host */
    struct sockaddr *ai_addr; /* Pointer to socket address structure */
    struct addrinfo *ai_next; /* Next structure in linked list */
};
```

Structures allocated and returned by *getaddrinfo()*

- There may be multiple combinations of host and service corresponding to the criteria specified in *host*, *service*, and *hints*



getnameinfo()

- converse of getaddrinfo()
- addr: a pointer to the socket address structure that is to be converted
- host, service: resulting host and service names returned in the buffers pointed to by

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

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

Returns 0 on success, or nonzero on error

Example (**Stream Sockets**)

sockets/is_seqnum.h

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <signal.h>
#include "read_line.h"          /* Declaration of readLine() */
#include "tlpi_hdr.h"

#define PORT_NUM "50000"        /* Port number for server */

#define INT_LEN 30              /* Size of string able to hold largest
                                   integer (including terminating '\n') */
```

sockets/is_seqnum.h

sockets/is_seqnum_sv.c

```
#define _BSD_SOURCE             /* To get definitions of NI_MAXHOST and
                                   NI_MAXSERV from <netdb.h> */
#include <netdb.h>
#include "is_seqnum.h"

#define BACKLOG 50

int
main(int argc, char *argv[])
{
    uint32_t seqNum;
    char reqLenStr[INT_LEN];            /* Length of requested sequence */
    char seqNumStr[INT_LEN];            /* Start of granted sequence */
    struct sockaddr_storage claddr;
    int lfd, cfd, optval, reqLen;
    socklen_t addrlen;
    struct addrinfo hints;
    struct addrinfo *result, *rp;
#define ADDRSTRLEN (NI_MAXHOST + NI_MAXSERV + 10)
    char addrStr[ADDRSTRLEN];
    char host[NI_MAXHOST];
    char service[NI_MAXSERV];

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [init-seq-num]\n", argv[0]);

①   seqNum = (argc > 1) ? getInt(argv[1], 0, "init-seq-num") : 0;
```

```
②      if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
           errExit("signal");

       /* Call getaddrinfo() to obtain a list of addresses that
          we can try binding to */

       memset(&hints, 0, sizeof(struct addrinfo));
       hints.ai_canonname = NULL;
       hints.ai_addr = NULL;
       hints.ai_next = NULL;
       hints.ai_socktype = SOCK_STREAM;
       hints.ai_family = AF_UNSPEC;        /* Allows IPv4 or IPv6 */
③      hints.ai_flags = AI_PASSIVE | AI_NUMERICSERV;
                           /* Wildcard IP address; service name is numeric */
④      if (getaddrinfo(NULL, PORT_NUM, &hints, &result) != 0)
           errExit("getaddrinfo");

       /* Walk through returned list until we find an address structure
          that can be used to successfully create and bind a socket */

       optval = 1;
⑤      for (rp = result; rp != NULL; rp = rp->ai_next) {
           lfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
           if (lfd == -1)
               continue;                   /* On error, try next address */
```

```
⑥          if (setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval))
                   == -1)
               errExit("setsockopt");

⑦          if (bind(lfd, rp->ai_addr, rp->ai_addrlen) == 0)
               break;                          /* Success */

           /* bind() failed: close this socket and try next address */

           close(lfd);
       }

       if (rp == NULL)
           fatal("Could not bind socket to any address");

⑧      if (listen(lfd, BACKLOG) == -1)
           errExit("listen");

       freeaddrinfo(result);

⑨      for (;;) {                  /* Handle clients iteratively */

           /* Accept a client connection, obtaining client's address */

           addrlen = sizeof(struct sockaddr_storage);
⑩          cfd = accept(lfd, (struct sockaddr *) &claddr, &addrlen);
           if (cfd == -1) {
               errMsg("accept");
               continue;
           }
```

```
⑪          if (getnameinfo((struct sockaddr *) &claddr, addrlen,
                        host, NI_MAXHOST, service, NI_MAXSERV, 0) == 0)
               snprintf(addrStr, ADDRSTRLEN, "(%s, %s)", host, service);
           else
               snprintf(addrStr, ADDRSTRLEN, "(?UNKNOWN?)");
           printf("Connection from %s\n", addrStr);

           /* Read client request, send sequence number back */

⑫          if (readLine(cfd, reqLenStr, INT_LEN) <= 0) {
               close(cfd);
               continue;                   /* Failed read; skip request */
           }

⑬          reqLen = atoi(reqLenStr);
           if (reqLen <= 0) {              /* Watch for misbehaving clients */
               close(cfd);
               continue;                   /* Bad request; skip it */
           }

⑭          snprintf(seqNumStr, INT_LEN, "%d\n", seqNum);
           if (write(cfd, &seqNumStr, strlen(seqNumStr)) != strlen(seqNumStr))
               fprintf(stderr, "Error on write");
⑮          seqNum += reqLen;               /* Update sequence number */

           if (close(cfd) == -1)           /* Close connection */
               errMsg("close");
       }
   }
```

sockets/is_seqnum_sv.c

sockets/is_seqnum_cl.c

```
#include <netdb.h>
#include "is_seqnum.h"

int
main(int argc, char *argv[])
{
    char *reqLenStr;                    /* Requested length of sequence */
    char seqNumStr[INT_LEN];            /* Start of granted sequence */
    int cfd;
    ssize_t numRead;
    struct addrinfo hints;
    struct addrinfo *result, *rp;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s server-host [sequence-len]\n", argv[0]);

    /* Call getaddrinfo() to obtain a list of addresses that
       we can try connecting to */

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    hints.ai_family = AF_UNSPEC;                /* Allows IPv4 or IPv6 */
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_NUMERICSERV;

①   if (getaddrinfo(argv[1], PORT_NUM, &hints, &result) != 0)
        errExit("getaddrinfo");
```

```
        /* Walk through returned list until we find an address structure
           that can be used to successfully connect a socket */

②      for (rp = result; rp != NULL; rp = rp->ai_next) {
③          cfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
           if (cfd == -1)
               continue;                       /* On error, try next address */

④          if (connect(cfd, rp->ai_addr, rp->ai_addrlen) != -1)
               break;                          /* Success */

            /* Connect failed: close this socket and try next address */

            close(cfd);
        }

        if (rp == NULL)
            fatal("Could not connect socket to any address");

        freeaddrinfo(result);

        /* Send requested sequence length, with terminating newline */

⑤      reqLenStr = (argc > 2) ? argv[2] : "1";
        if (write(cfd, reqLenStr, strlen(reqLenStr)) !=  strlen(reqLenStr))
            fatal("Partial/failed write (reqLenStr)");
        if (write(cfd, "\n", 1) != 1)
            fatal("Partial/failed write (newline)");
```

```
        /* Read and display sequence number returned by server */

⑥      numRead = readLine(cfd, seqNumStr, INT_LEN);
        if (numRead == -1)
            errExit("readLine");
        if (numRead == 0)
            fatal("Unexpected EOF from server");

⑦      printf("Sequence number: %s", seqNumStr);          /* Includes '\n' */

        exit(EXIT_SUCCESS);                                /* Closes 'cfd' */
    }
```

sockets/is_seqnum_cl.c

Results

```
$ ./is_seqnum_sv &
[1] 4075
$ ./is_seqnum_cl localhost
Connection from (localhost, 33273)
Sequence number: 0
$ ./is_seqnum_cl localhost 10
Connection from (localhost, 33274)
Sequence number: 1
$ ./is_seqnum_cl localhost
Connection from (localhost, 33275)
Sequence number: 11
```

Client 1: requests 1 sequence number
Server displays client address + port
Client displays returned sequence number
Client 2: requests 10 sequence numbers

Client 3: requests 1 sequence number

```
$ telnet localhost 50000

Trying 127.0..0.1...
Connection from (localhost, 33276)
Connected to localhost.
Escape character is '^]'.
1
12
Connection closed by foreign host.
```

Our server uses this port number
Empty line printed by telnet

Enter length of requested sequence
telnet displays sequence number and
detects that server closed connection