

# CSCI 210 Systems Programming

Week 14

Sockets

The Linux Programming Interface (Ch. 56-59)

# Overview

- Introduction to sockets (Ch. 56)
  - The sockets API
  - Stream and datagram sockets
- UNIX domain sockets (Ch. 57)
  - For applications that run on the same host
- Introduction to TCP/IP Networks (Ch. 58)
- Internet domain sockets (Ch. 59)

# sockets

- Like pipes, they are a method of inter-process communication (IPC)
  - Unlike pipes, they also support communication between applications that are on different hosts connected by a network
- The common way to employ sockets is in a client-server architecture
  - Each application creates a socket, which is needed by both applications as a communication tool.
  - The server binds its socket to a well-known address (name), so that clients can locate it

# Creating a socket

- The `socket()` system call creates a new socket and returns a file descriptor to it that can be used in subsequent socket related system calls and using for using the familiar `read()/write()` system calls to receive/send data.
  - `fd = socket(domain, type, protocol)`
- Sockets exist in a *communication domain*
- The communication domain determines:
  - The “format” of the socket address
  - The range of communication: on the same host, or on different hosts

# Communication domains

- The UNIX domain
  - AF\_UNIX
- The IPv4 domain
  - AF\_INET
  - Most of the application on the Internet use this communication domain
- The IPv6 domain
  - AF\_INET6

# Communication domains

**Table 56-1:** Socket domains

Domain	Communication performed	Communication between applications	Address format	Address structure
AF_UNIX	within kernel	on same host	pathname	<i>sockaddr_un</i>
AF_INET	via IPv4	on hosts connected via an IPv4 network	32-bit IPv4 address + 16-bit port number	<i>sockaddr_in</i>
AF_INET6	via IPv6	on hosts connected via an IPv6 network	128-bit IPv6 address + 16-bit port number	<i>sockaddr_in6</i>

# Socket types

- Stream or datagram
- Stream socket semantics are the same as in pipes
  - Byte streams, no message boundaries
  - Need connections on both ends, i.e., connection oriented
- Datagram sockets
  - Message oriented, the data is read/written message by message
  - No connection needs to be established → can just specify a recipient
  - They are not reliable

**Table 56-2:** Socket types and their properties

Property	Socket type	
	Stream	Datagram
Reliable delivery?	Y	N
Message boundaries preserved?	N	Y
Connection-oriented?	Y	N

# Stream sockets

- Use SOCK\_STREAM as the socket type during creation
- Operate in connected pairs
  - peer socket → the socket on the other end
  - peer address → the address of the peer socket
  - peer application → the application on the other end
  - The term “remote” or “foreign” is also used synonymously with peer
- The term “local” is used for the application, socket, address for our end of the connection
- TCP/IP uses stream sockets
  - TCP socket → Internet domain stream socket

# Datagram sockets

- Use SOCK\_DGRAM as the socket type during creation
- Message, i.e., *datagram*, oriented
- Not reliable
  - Messages can come in any order
  - Messages can be duplicated
  - Messages may not arrive
- Connectionless
  - No need to establish connection
- UDP uses datagram sockets
  - UDP socket → Internet domain datagram socket

# socket system calls

- `socket()` : creates a new socket
- `bind()` : binds the socket to an address (file, IP address, etc.)
- `listen()` : allows a stream socket to accept incoming connections from other sockets, marks the socket as *passive*
- `accept()` : accepts a connection and creates a new file descriptor for it
- `connect()` : establishes a connection to a peer
- `read()/write()/send()/recv()/sendto()/recvfrom()` : sends/receives data

# Stream sockets

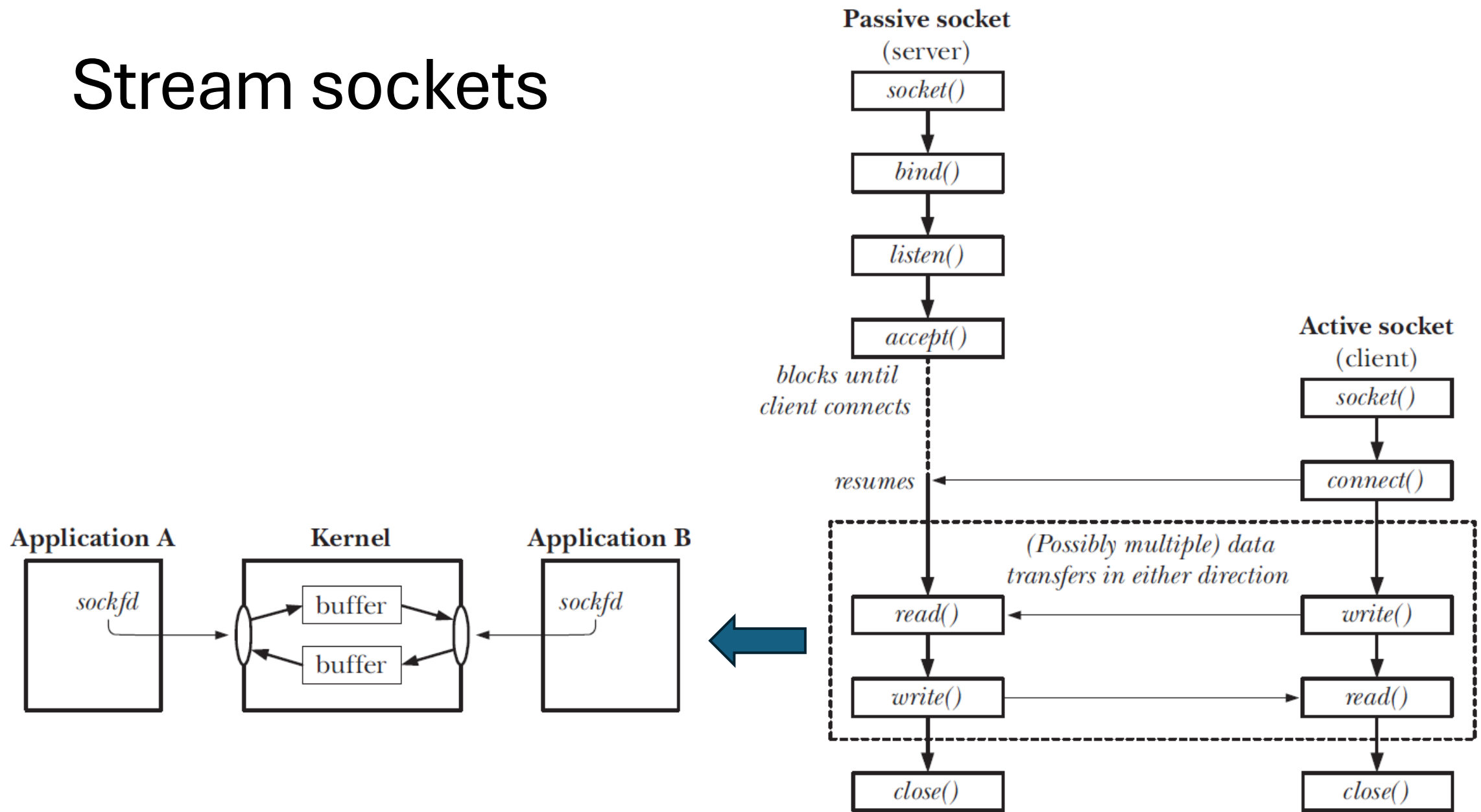


Figure 56-1: Overview of system calls used with stream sockets

# socket()

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

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

Returns file descriptor on success, or `-1` on error

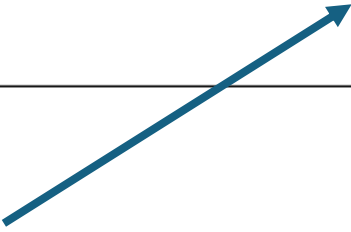
- Use 0 for protocol
- Example:

```
int sockfd;
```

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

# bind()

The address structure  
varies between domains



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

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

Returns 0 on success, or -1 on error

- Example:

```
int sfd;  
struct sockaddr_un addr;  
sfd = socket(AF_UNIX, SOCK_STREAM, 0);  
memset(&addr, 0, sizeof(struct sockaddr_un));  
addr.sun_family = AF_UNIX;  
strncpy(addr.sun_path, "/tmp/mysock/", sizeof(addr.sun_path) - 1);  
bind(sockfd, &addr, sizeof(struct sockaddr_un));
```

# listen()

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

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

Returns 0 on success, or -1 on error

- The listen() system call marks the stream socket as passive.
  - The socket will subsequently be used to accept connections from other (active) sockets.
- We can't apply listen() to a connected socket—that is, a socket on which a connect() has been successfully performed or a socket returned by a call to accept().
- Example:

```
listen(sfd, 20) ;
```

# accept()

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

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

Returns file descriptor on success, or `-1` on error

- Creates a new socket, and it is this new socket that is connected to the peer socket that performed the `connect()`.
- A file descriptor for the connected socket is returned as the function result of the `accept()` call.
- The listening socket (`sockfd`) remains open and can be used to accept further connections.
- Example: **`newfd = accept(sockfd, NULL, NULL);`**

# connect()

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

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

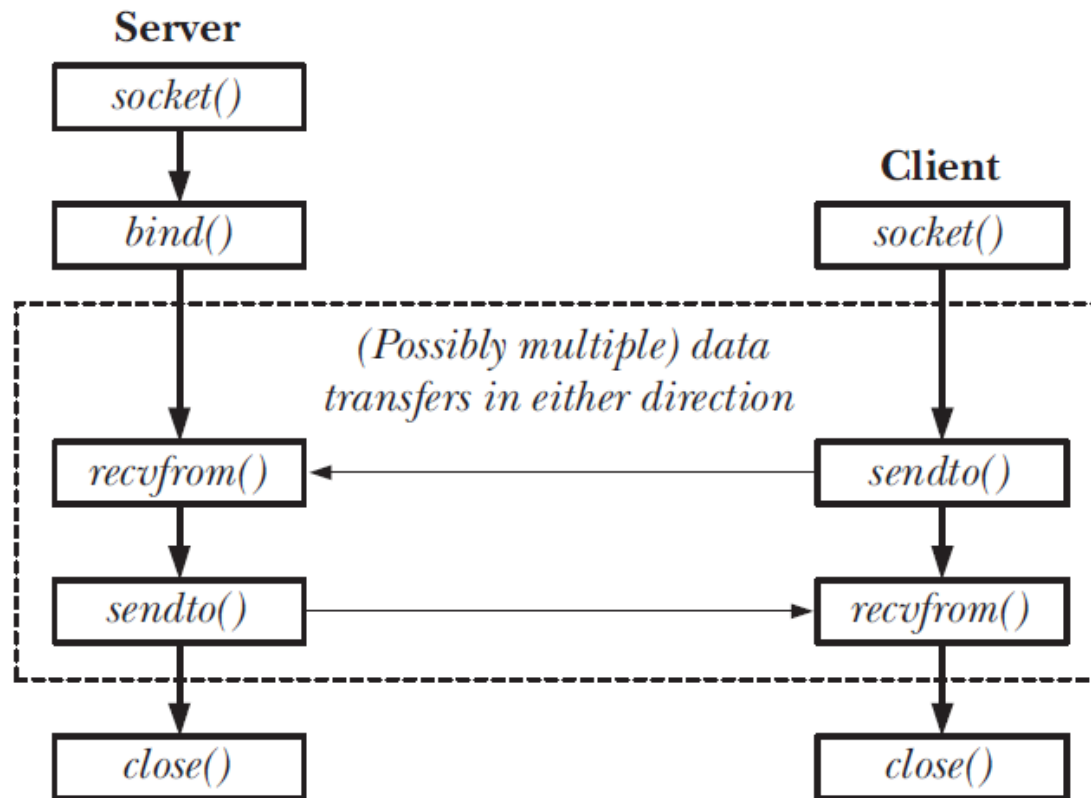
Returns 0 on success, or -1 on error

- Example:

```
int sfd;  
struct sockaddr_un addr;  
sfd = socket(AF_UNIX, SOCK_STREAM, 0);  
memset(&addr, 0, sizeof(struct sockaddr_un));  
addr.sun_family = AF_UNIX;  
strncpy(addr.sun_path, "/tmp/mysock/", sizeof(addr.sun_path) - 1);  
connect(sfd, &addr, sizeof(struct sockaddr_un));
```

# Datagram sockets

- No `listen()`, `accept()`, `connect()` calls are needed



**Figure 56-4:** Overview of system calls used with datagram sockets

# sendto()/recvfrom()

```
#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

# Unix Domain Sockets

- Similar to named pipes
  - Use a file on a known path to establish communication between two applications
- Unlike named pipes, you can establish a message (datagram) oriented connection
- Unix domain addresses:

```
struct sockaddr_un {  
    sa_family_t sun_family; /* Always AF_UNIX */  
    char sun_path[108]; /* Null-terminated socket pathname */  
};
```

# Binding Unix domain sockets

- The file is created on the file
  - Need access rights to the directories on the path
- Cannot be an existing file
- File type has the character “s” (as opposed to “p” for pipes)
- Cannot use `open()` to open a socket.
- When the socket is no longer required, its pathname entry can (and generally should) be removed using `unlink()` (or `remove()`).
  - It is not removed automatically by `close()`
- Creating sockets in the `/tmp` directory is easy
  - But may not be secure

# Example server

```
#include <sys/un.h>
#include <sys/socket.h>
int main(int argc, char *argv[]) {
    struct sockaddr_un addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[100];
    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, "/tmp/mysock", sizeof(addr.sun_path) - 1);
    bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un));
    listen(sfd, 5);
    for (;;) { /* Handle client connections iteratively */
        cfd = accept(sfd, NULL, NULL);
        while ((numRead = read(cfd, buf, 100)) > 0)
            write(1, buf, numRead);
        close(cfd)
    }
}
```

# Example client

```
#include <sys/un.h>
#include <sys/socket.h>
int main(int argc, char *argv[]) {
    struct sockaddr_un addr;
    int sfd;
    ssize_t numRead;
    char buf[100];
    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, "/tmp/mysock", sizeof(addr.sun_path) - 1);
    connect(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un));
    while ((numRead = read(0, buf, 100)) > 0)
        write(sfd, buf, numRead);
    close(sfd);
}
```

# TCP/IP Networks

- internet
  - A network of computer networks
  - subnetwork: one of the networks composing an internet
- TCP/IP is a protocol suite that enables connectivity by providing tools/standards at different layers
- The Internet refers to the TCP/IP internet that connects millions of computers globally

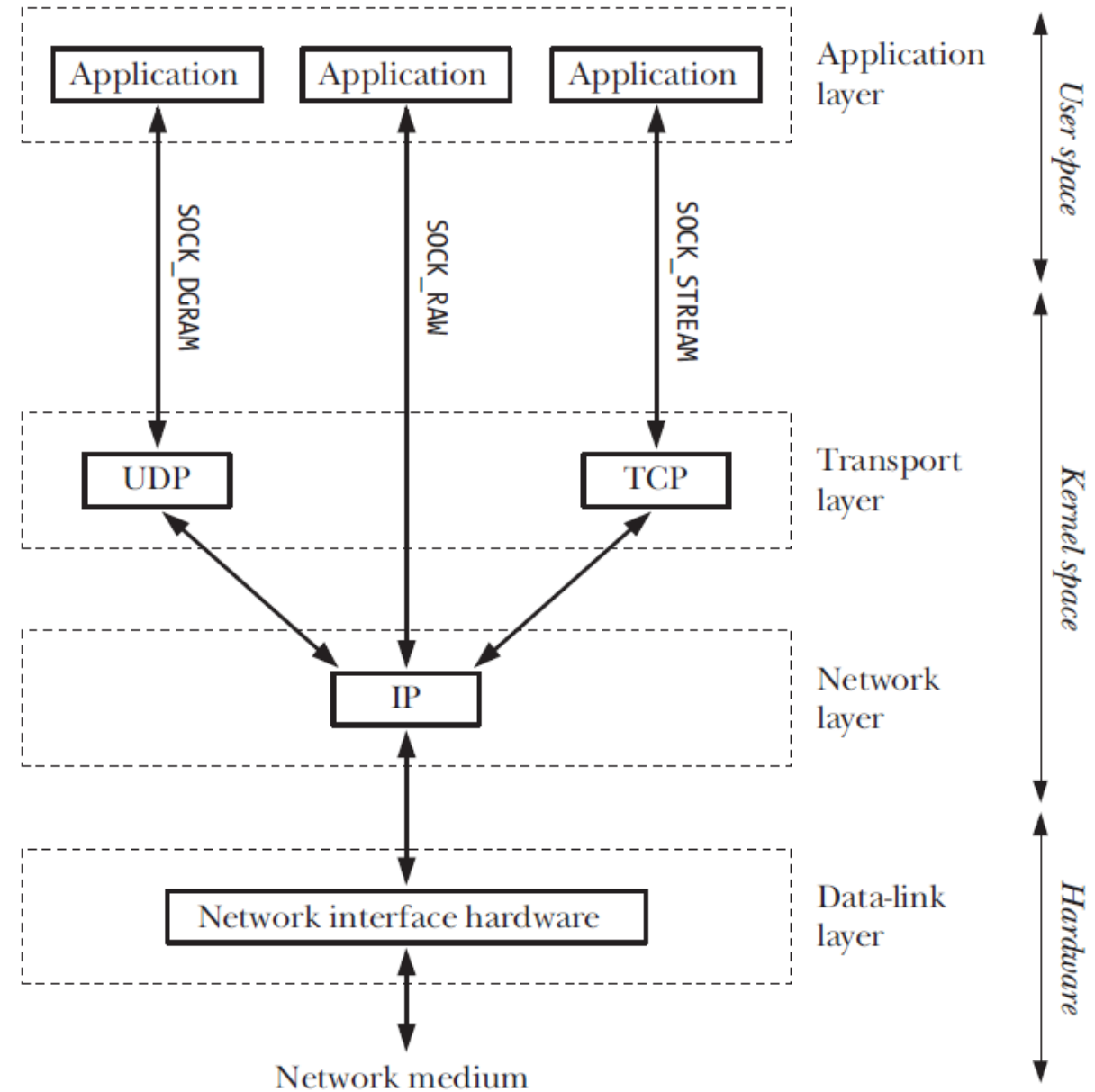


Figure 58-2: Protocols in the TCP/IP suite

# The Network Layer: IP

- Breaks data into fragments
- Routes data across the internet
- Provides services to the transport layer
- IP is connectionless and unreliable
  - Works with datagrams
- TCP/IP is built on top of IP and contains mechanisms to ensure reliability

# IP addresses (IPv4)

- An IP address specifies a network and a specific host in that network

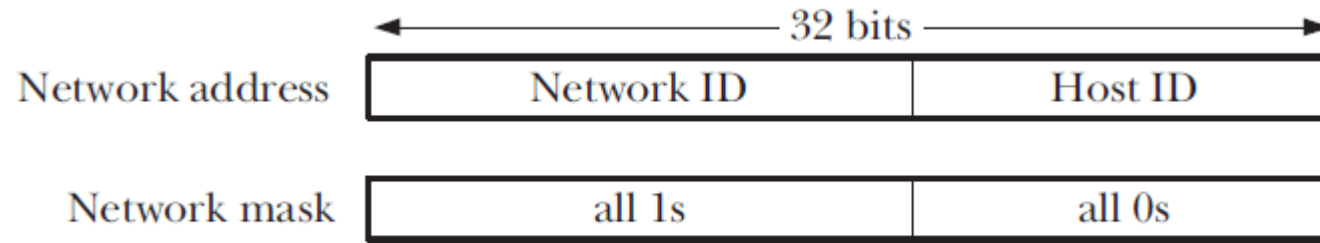


Figure 58-5: An IPv4 network address and corresponding network mask

- 204.152.189.0/24 means the first 24 bits are used for the network ID.
  - With 8 bits for hosts, we can use 254 unique Internet addresses in this network
  - Host 0 (network ID) and host 255 (broadcast) are not available

# Subnetting

- Subnetworks within a network can be defined to isolate or organize computers in a network in smaller groups.

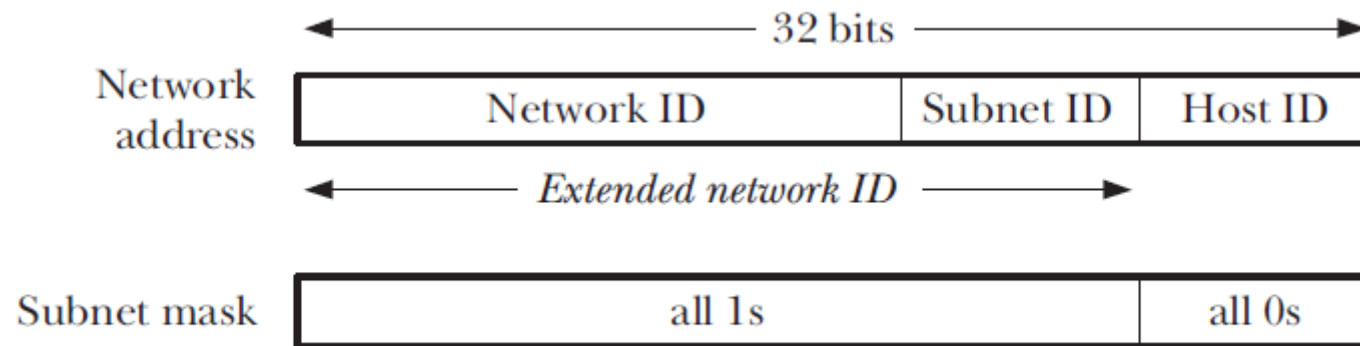


Figure 58-6: IPv4 subnetting

# Transport Layer

- There are two widely used transport-layer protocols in the TCP/IP suite:
  - User Datagram Protocol (UDP) is the protocol used for datagram sockets.
  - Transmission Control Protocol (TCP) is the protocol used for stream sockets.
- Use of “port” numbers
  - The task of the transport protocol is to provide an end-to-end communication service to applications residing on different hosts (or sometimes on the same host).
  - In order to do this, the transport layer requires a method of differentiating the applications on a host.
  - In TCP and UDP, this differentiation is provided by a 16-bit port number.
  - Well-known port numbers have service names, see `/etc/services`.

# TCP

- Provides a seamless connection in terms of sockets to applications

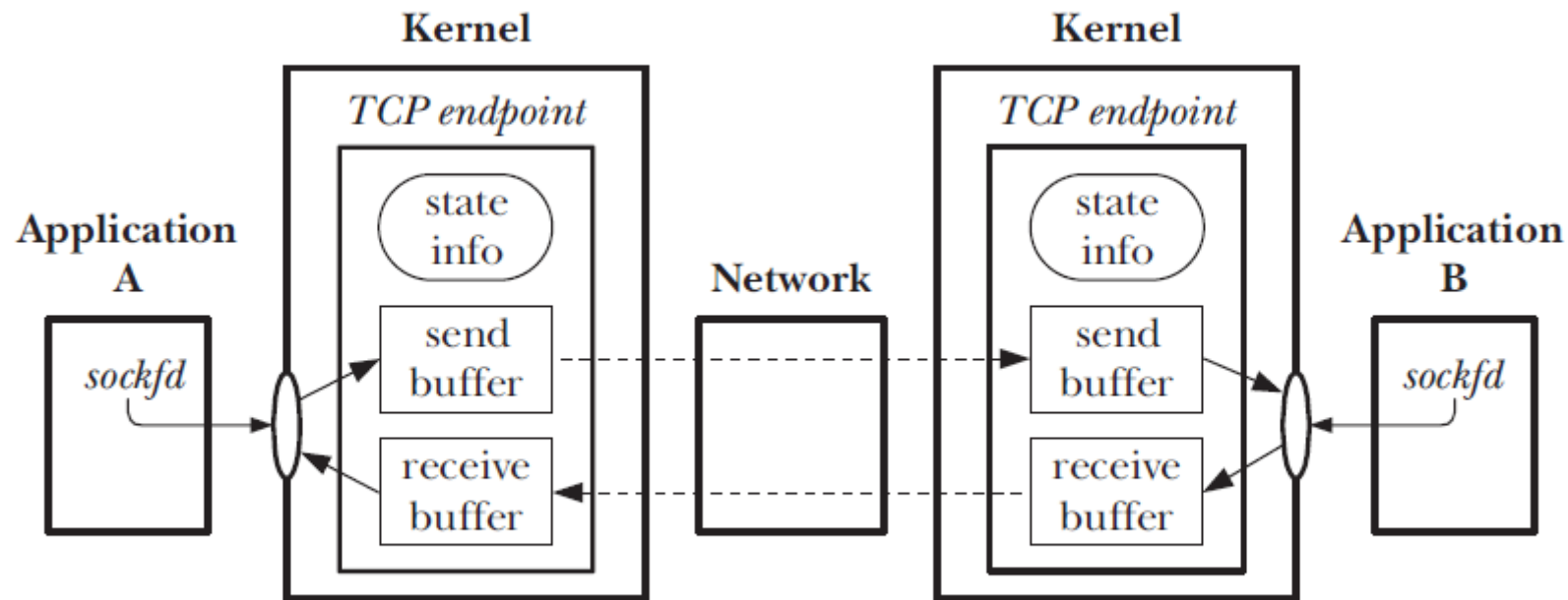


Figure 58-8: Connected TCP sockets

# Internet Domain Sockets

- Internet domain stream sockets are implemented on top of TCP. They provide a reliable, bidirectional, byte-stream communication channel.
- Internet domain datagram sockets are implemented on top of UDP.

# Network Byte Order

- The communication over a network is potentially between very different systems
  - Not the same as pipes, UNIX domain sockets on the same machine
- The byte order on a host computer could be either one of the two ordering approaches
  - Big Endian: most significant byte on the left
  - Little Endian: most significant byte on the right
- We need to make sure there is a standard accepted by all the computers over a network → network byte order
- Utility functions to convert long and short integers from one order to the other: `htons()`, `htonl()`, `ntohs()`, `ntohl()`

# Internet socket addresses

- Can be cast to the generic struct sockaddr

```

struct in_addr {
    in_addr_t s_addr;
};

struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr sin_addr;
    unsigned char  __pad[X];
};

```

# Host and service (port) conversion functions

- There are several utility functions to convert human readable IP addresses to their byte representation in these socket address structures
- `inet_addr()`
- `inet_pton()`
- `inet_ntop()`
- `get_addrinfo()` is a more generic function that can help you get a `struct addrinfo` type address given a domain and service name.
- Example: `getaddress.c`

# getaddress.c

```
#include <arpa/inet.h>
int main() {
    char domain[256];
    char res[256];
    struct addrinfo hints;
    struct addrinfo *result,*rp;
    printf("Enter an http domain name:");
    scanf("%s",domain);

    memset(&hints, 0, sizeof(struct addrinfo));

    hints.ai_family = AF_INET;

    getaddrinfo(domain,"http",&hints,&result);

    for (rp = result; rp!=NULL; rp = rp->ai_next) {
        printf("Address: %s\n",inet_ntop(AF_INET,&(((struct
            sockaddr_in *) (rp->ai_addr))->sin_addr),res,256));
    }
}
```

# struct addrinfo

```
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 */  
};
```

# Client Server Application

## Example using TCP/IP

# TCP server

```
#include <arpa/inet.h>
#include <netinet/in.h>
int main(int argc, char *argv[]) {
    struct sockaddr_in addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[100];
    sfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&addr, 0, sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(60000);
    addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_in));
    listen(sfd, 5);
    for (;;) { /* Handle client connections iteratively */
        cfd = accept(sfd, NULL, NULL);
        while ((numRead = read(cfd, buf, 100)) > 0)
            write(1, buf, numRead);
        close(cfd);
    }
}
```

# TCP client

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

int main(int argc, char *argv[]) {
    struct sockaddr_in addr;
    int sfd;
    ssize_t numRead;
    char buf[100];
    sfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&addr, 0, sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    addr.sin_port = htons(60000);
    connect(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_in));
    while ((numRead = read(0, buf, 100)) > 0)
        write(sfd, buf, numRead);
    close(sfd);
}
```