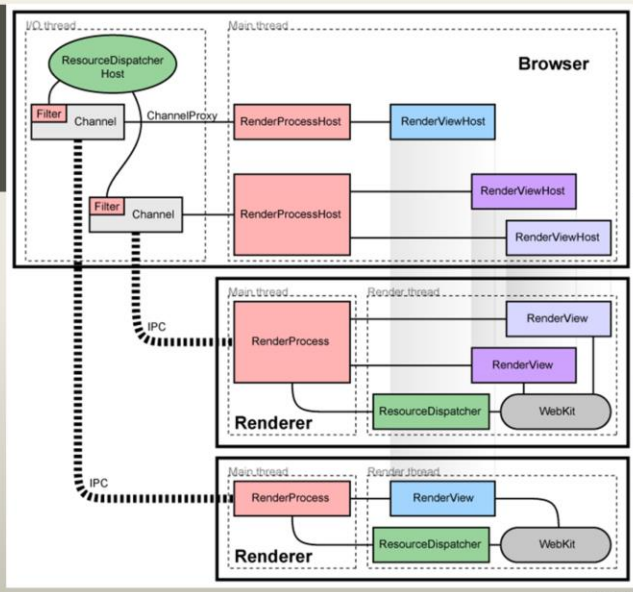
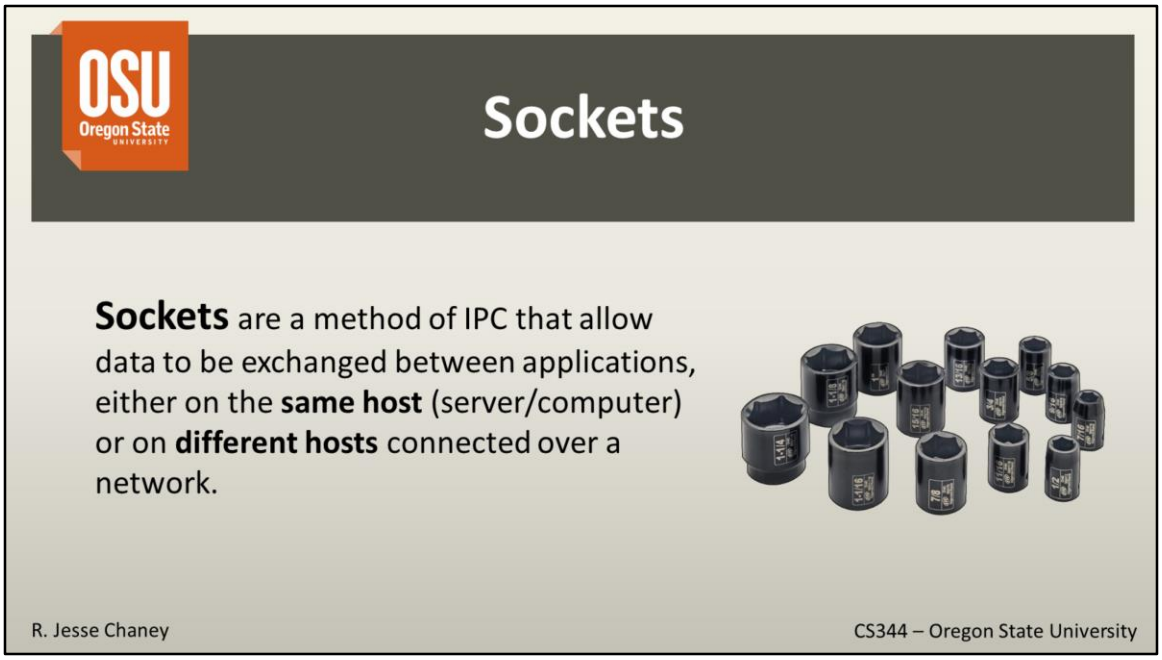

ONE DOES NOT SIMPLY
EAT TOO MANY TOOTSIE ROLLS
Tootsie Roll


IO thread
Main thread
Browser
ResourceDispatcher Host
Filter
Channel
ChannelProxy
RenderProcessHost
RenderViewHost
Filter
Channel
RenderProcessHost
RenderViewHost
RenderViewHost
IPC
Main thread
Render thread
RenderProcess
RenderView
RenderView
ResourceDispatcher
WebKit
Renderer
IPC
Main thread
Render thread
RenderProcess
RenderView
ResourceDispatcher
WebKit
Renderer

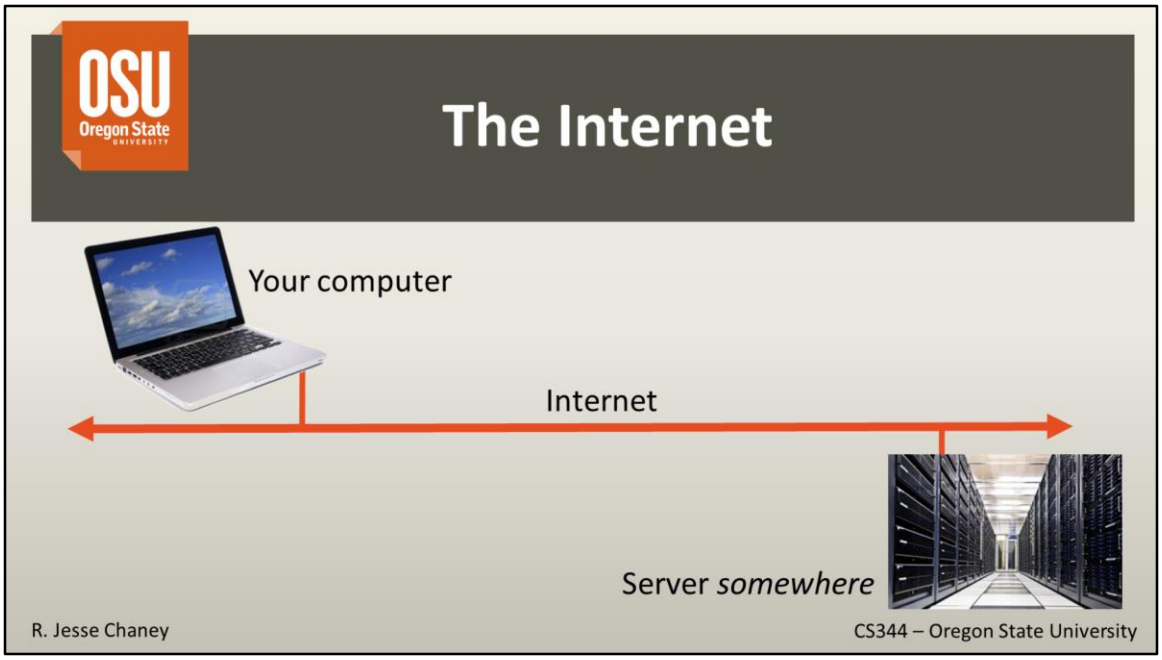


The first widespread implementation of the sockets API appeared with **4.2BSD in 1983**, and this API has been ported to virtually every UNIX

In a typical client-server scenario, applications communicate using sockets as follows:

- Each application creates a socket. The socket is the "apparatus" that allows communication, and both applications require one.
- The server binds its socket to a well-known address (name) so that clients can locate it. implementation, as well as most other operating systems.

Sockets may be the most flexible form of IPC that we'll study. The fact that sockets can be use for processes to communicate over a network to processes running on other systems makes them very popular. The traffic that you send out over the internet is all socket based.

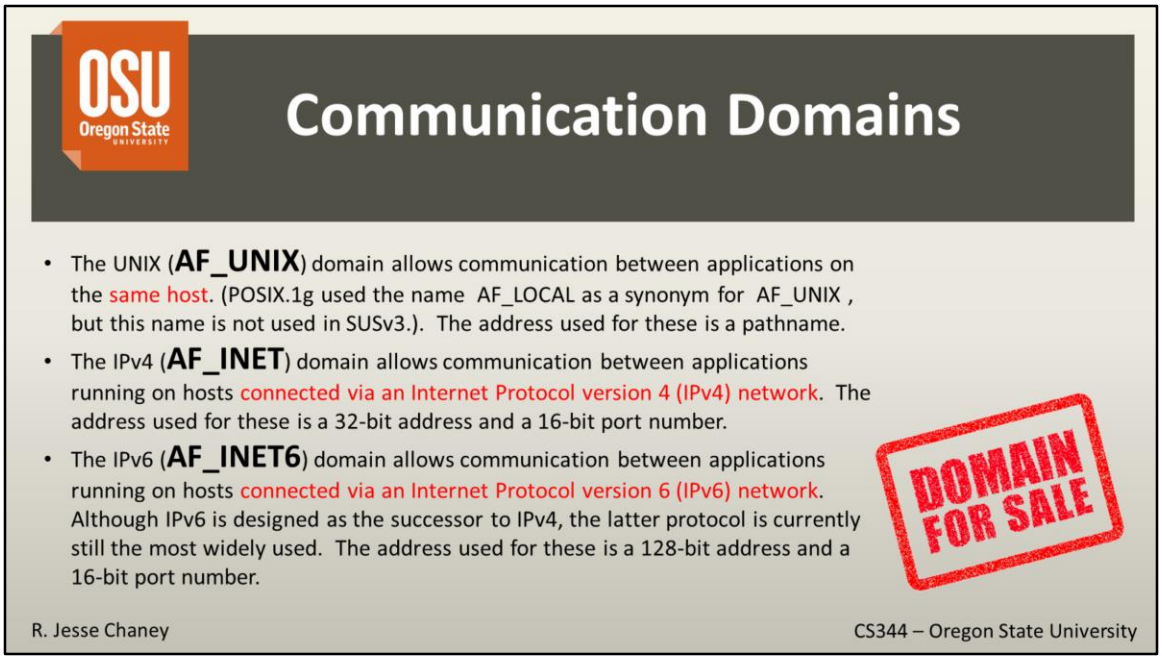


When you are making connections over a network to another computer, it is very (VERY) likely that you are doing it over a socket connection.

That other computer could be in the next room, or it could be in orbit someplace.

In a typical socket based client-server scenario, applications communicate using sockets as follows:

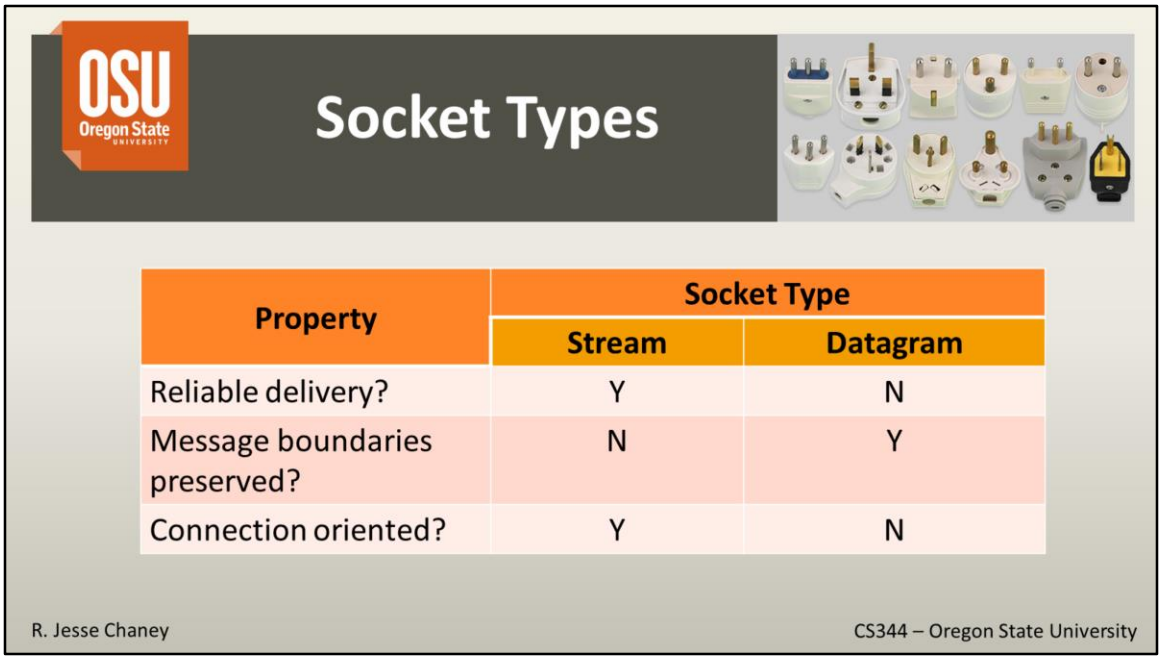
- Each application creates a socket. A socket is the “apparatus” that allows communication, and both applications require one.
- The server binds its socket to a well-known address (name) so that clients can locate it.



Sockets exist in a communication domain, which determines:

- the method of identifying a socket (i.e., the format of a socket “address”);
- the range of communication (i.e., either between applications on the same host or between applications on different hosts connected via a network).

The IPv$4 addresses are the normal dotted addresses. The IPv6 addresses are something we’ve not quite gotten accustomed to saying yet.



Every sockets implementation provides at least two types of sockets: **stream** and **datagram**. These socket types are supported in both the UNIX and the Internet domains.

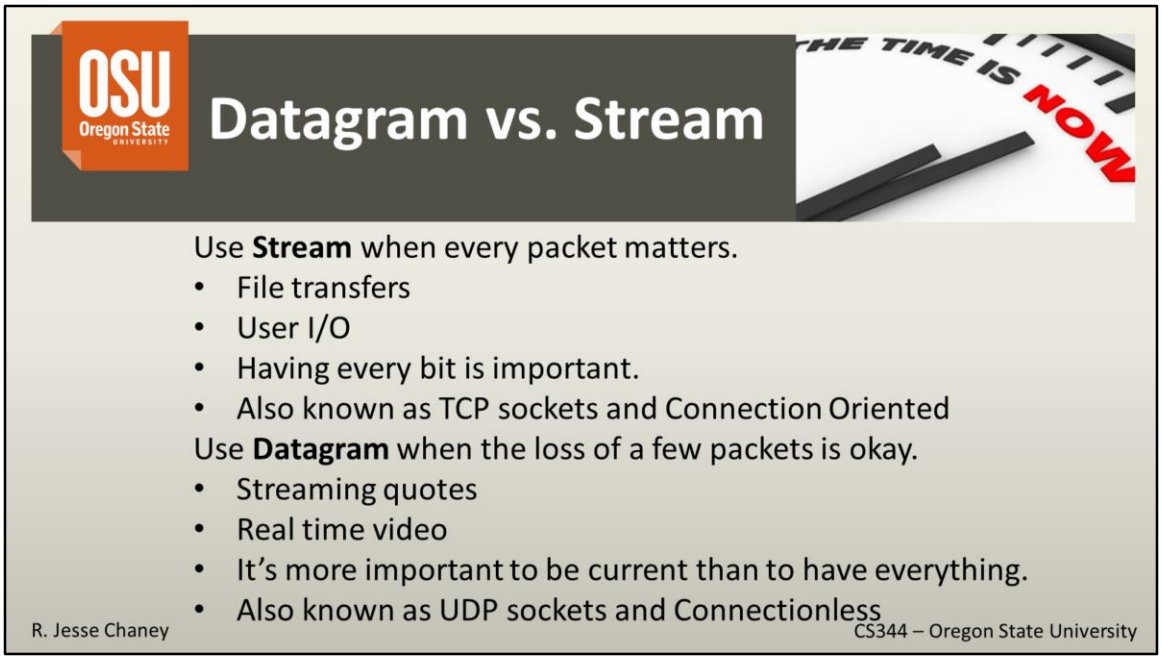
Stream sockets (SOCK_STREAM) provide a **reliable**, **bidirectional**, **byte-stream** communication channel. By the terms in this description, we mean the following:

- Reliable means that we are guaranteed that either the transmitted data will arrive intact at the receiving application, exactly as it was transmitted by the sender (assuming that neither the network link nor the receiver crashes), or that we'll receive notification of a probable failure in transmission.
- Bidirectional means that data may be transmitted in either direction between two sockets.
- Byte-stream means that, as with pipes, there is no concept of message boundaries.

Stream sockets operate in connected pairs. For this reason, stream sockets are described as connection-oriented. The term peer socket refers to the socket at the other end of a connection; peer address denotes the address of that socket; and peer application denotes the application utilizing the peer socket.

Datagram sockets (SOCK_DGRAM) allow data to be exchanged in the form of messages called datagrams. With datagram sockets, message boundaries are preserved, but data transmission is not reliable. Messages may arrive out of order, be duplicated, or not arrive at all.

In the Internet domain, datagram sockets employ the User Datagram Protocol (UDP), and stream sockets (usually) employ the Transmission Control Protocol (TCP).



Transmission Control Protocol (aka TCP) are Stream sockets
User Datagram Protocol (UDP) are Datagram sockets

Datagram works a bit differently from Stream. When you send data via Stream you first create a connection. Once the Stream connection is established Stream **guarantees** that your data arrives at the other end, or it will tell you that an error occurred.

With Datagram you just send packets of data to some IP address on the network. **You have no guarantee that the data will arrive**. You also have no guarantee about the order which Datagram packets arrive in at the receiver. This means that **Datagram has less protocol overhead** (no stream integrity checking) than Stream.

Datagram is appropriate for data transfers where it doesn't matter if a packet is lost in transition. For instance, imagine a **transfer of a live TV-signal over the internet**. You want the signal to arrive at the clients as close to live as possible. Therefore, if a frame or two are lost, you don't really care. You don't want the live broadcast to be delayed just to make sure all frames are shown at the client. You'd rather skip the missed frames, and move directly to the newest frames at all times.

This could also be the case with a **surveillance camera broadcasting** over the internet. Who cares what happened in the past, when you are trying to monitor the present. You don't want to end up being 30 seconds behind reality, just because you want to show all frames to the person monitoring the camera. It is a bit different with the storage of the camera recordings. You may not want to lose a single frame when recording the images from the camera to disk. You may rather want a little delay, than not have those frames to go back and examine, if something important occurs.

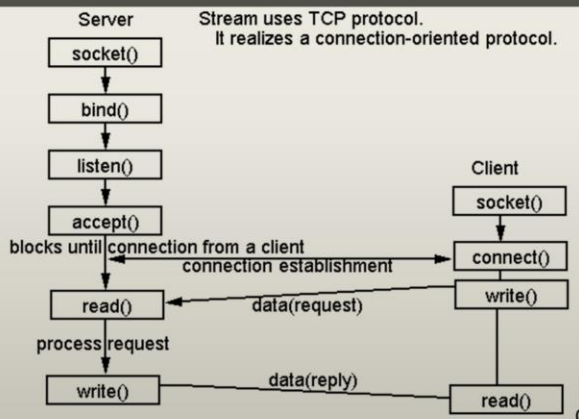
Stock quotes

Socket System Calls

The key socket system calls are the following:

- The **socket()** system call creates a new socket.
- The **bind()** system call binds a socket to an address. Usually, a server employs this call to bind its socket to a well-known address so that clients can locate the socket.
- The **listen()** system call allows a stream socket to accept incoming connections from other sockets.
- The **accept()** system call accepts a connection from a peer application on a listening stream socket, and optionally returns the address of the peer socket.
- The **connect()** system call establishes a connection with another socket.

Stream Sockets in an Application

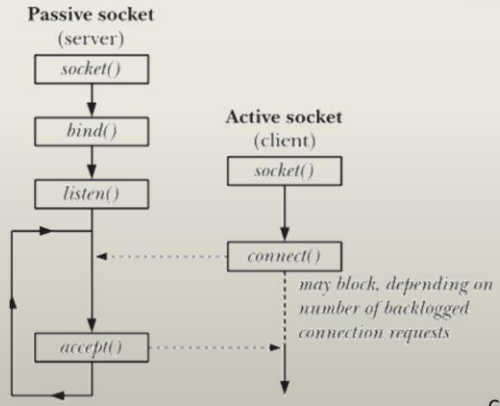


Active vs Passive Sockets

Stream sockets are often distinguished as being either active or passive:

- A socket that has been **created using socket() is active**. An active socket can be used in a **connect()** call to establish a connection to a passive socket. This is referred to as performing an **active open**.
- A **passive socket** (or a **listening socket**) is one that has been marked to **allow incoming connections** by calling **listen().** Accepting an incoming connection is referred to as performing a **passive open**.

Active Client and Passive Server



R. Jesse Chaney

CS344 – Oregon State University

To understand the purpose of the backlog argument, we first observe that the client may call connect() before the server calls accept().

Listening Sockets

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

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

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

int listen(int sockfd, int backlog);

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

R. Jesse Chaney CS344 – Oregon State University

- The domain argument specifies the communication domain for the socket.
- The type argument specifies the socket type.
    - This argument is usually specified as either SOCK_STREAM , to create a stream socket, or SOCK_DGRAM , to create a datagram socket.
- **The protocol argument is always specified as 0 for the socket types we describe in the book**.
    - Nonzero protocol values are used with some socket types that we don't describe.

On success, **socket()** returns a **file descriptor** used to refer to the newly created socket in later system calls.

The **bind()** system call **binds a socket to an address**. The addr argument is a pointer to a structure specifying the address to which this socket is to be bound. The type of structure passed in this argument depends on the socket domain.

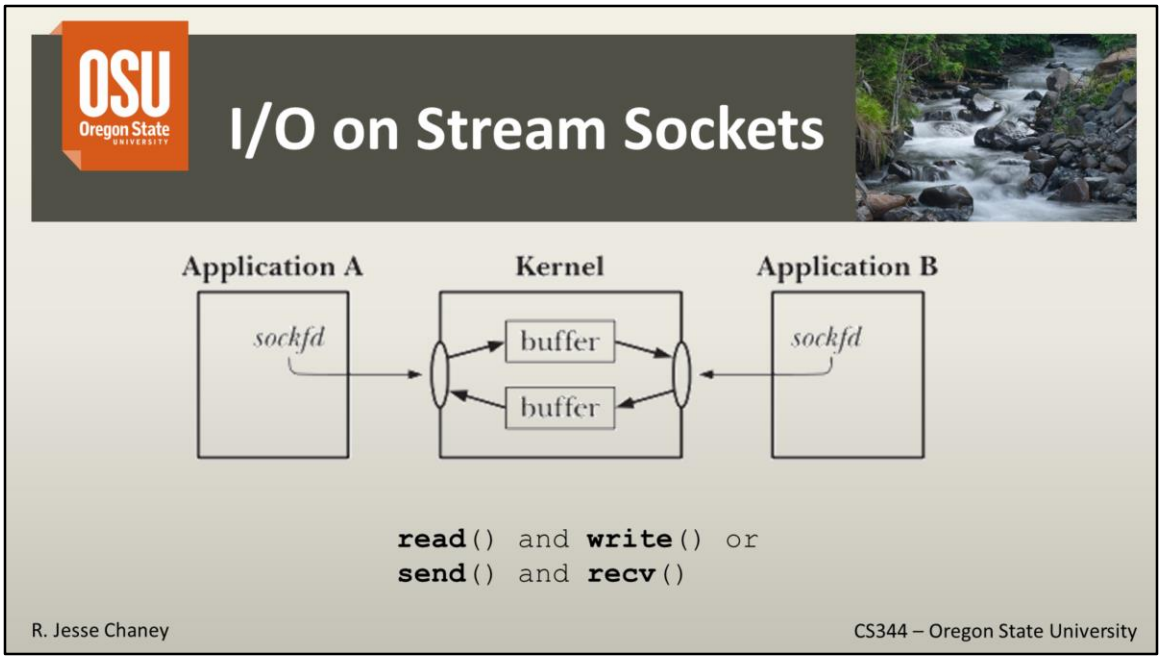
The listen() system call marks the stream socket referred to by the file descriptor sockfd as **passive**. The socket will subsequently be **used to accept connections** from other (active) sockets.

The **accept()** system call **accepts an incoming connection on the listening stream socket** referred to by the file descriptor sockfd. If there are no pending connections when accept() is called, the call **blocks until a connection request arrives**.

A **key point to understand about accept()** is that it **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 **connect()** system call **connects the active socket** referred to by the file descriptor sockfd **to the listening socket** whose address is specified by addr and addrlen.

If connect() fails and we wish to reattempt the connection, then SUSv3 specifies that the portable method of doing so is to close the socket, create a new socket, and reattempt the connection with the new socket.



A pair of connected stream sockets provides a **bidirectional communication channel** between the two endpoints.

The semantics of I/O on connected stream sockets are similar to those for pipes:

- To perform I/O, we use the read() and write() system calls (or the socket-specific send() and recv(), which we describe in Section 61.3). 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. Afterward, when the peer application attempts to read from the other end of the connection, it receives end-of-file (once all buffered data has been read). If the peer application attempts to write to its socket, it receives a SIGPIPE signal, and the system call fails with the error EPIPE .

The operation of datagram sockets can be explained by analogy with the postal system:

- The **socket()** system call is the equivalent of **setting up a mailbox**. Each application that wants to send or receive datagrams creates a datagram socket using socket().
- In order to allow another application to send it datagrams (letters), an application uses **bind()** to bind its socket to a well-known address. Typically, a server binds its socket to a well-known address, and a client initiates communication by sending a datagram to that address. (In some domains—notably the UNIX domain—the client may also need to use bind() to assign an address to its socket if it wants to receive datagrams sent by the server.)
- To send a datagram, an application calls sendto(), which takes as one of its arguments the address of the socket to which the datagram is to be sent. This is analogous to putting the recipient's address on a letter and posting it.
- In order to receive a datagram, an application calls recvfrom(), which may block if no datagram has yet arrived. Because recvfrom() allows us to obtain the address of the sender, we can send a reply if desired. (This is useful if the sender's socket is bound to an address that is not well known, which is typical of a client.) Here, we stretch the analogy a little, since there is no requirement that a posted letter is

marked with the sender's address.

- When the socket is no longer needed, the application closes it using close().

Datagram Sockets in an Application