**Sockets**

# Sockets

Sockets are a method of IPC that allow communication between applications on the same host or on different hosts connected via a network.

In a typical 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”); and
* The range of communication (i.e., either between applications on the same host or between applications on different hosts connected via a network).

Modern operating systems support at least the following domains:

**UNIX (AF\_UNIX)** domain allows communication between applications on the same host

**IPv4 (AF\_INET)** domain allows communication between applications running on hosts connected via an IPv4

**IPv6 (AF\_INET6)** domain allows communication between applications running on hosts connected via an IPv6

**AF** - Address family

**PF** - Protocol family

**Socket domains:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Domain** | **Communication performed** | **Communication between applications** | **Address format** | **Address structure** |
| AF\_UNIX | within kernel | on same host | pathname | sockaddr\_un |
| AF\_INET | via IPv4 | on hosts connected via an IPv4 network | 32-bit IPv4 address + 16-bit port number | sockaddr\_in |
| AF\_INET6 | via IPv6 | on hosts connected via an IPv6 network | 128-bit IPv6 address + 16-bit port number | sockaddr\_in6 |

**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 (SOCK\_STREAM)** provide a reliable, bidirectional, byte-stream communication channel.

Stream sockets are described as connection-oriented.

**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.

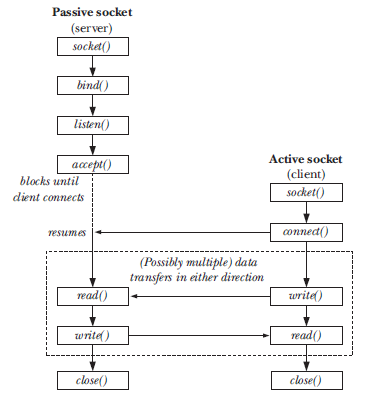
Datagram sockets are generic concept of a connectionless socket.

# Stream Sockets

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** (also called 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.

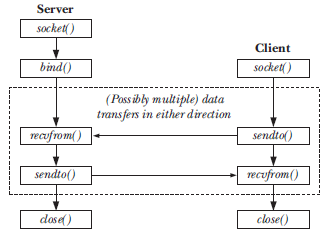
The server performs the passive open, and the client performs the active open.



Overview of system calls used with stream sockets

Since kernel 2.4.25, Linux allows this limit to be adjusted at run time via the Linux-pecific “/proc/sys/net/core/somaxconn” file.

# Datagram Sockets



Overview of system calls used with datagram sockets

# Using connect() with Datagram Sockets

Calling connect() on a datagram socket causes the kernel to record a particular address as this socket’s peer.

**Connected datagram socket:** a datagram socket on which connect() has been called

**Unconnected datagram socket**: a datagram socket on which connect() has not been called (i.e., the default for a new datagram socket).

After a datagram socket has been connected:

* Datagrams can be sent through the socket using write() (or send()) and are automatically sent to the same peer socket. As with sendto(), each write() call results in a separate datagram.
* Only datagrams sent by the peer socket may be read on the socket.

Note: the effect of connect() is asymmetric for datagram sockets. The above statements apply only to the socket on which connect() has been called, not to the remote socket to which it is connected (unless the peer application also calls connect() on its socket).

The obvious advantage of setting the peer for a datagram socket is that we can use simpler I/O system calls when transmitting data on the socket. Setting the peer is useful primarily in an application that needs to send multiple datagrams to a single peer (which is typical of some datagram clients).

# Creating a Connected Socket Pair: socketpair()

#include <sys/socket.h>

**int socketpair(int domain, int type, int protocol, int sockfd[2]);**

Returns 0 on success, or –1 on error

This socketpair() system call can be used only in the UNIX domain; that is, domain must be specified as AF\_UNIX. (since the socket pair is created on a single host system.) The socket type may be specified as either SOCK\_DGRAM or SOCK\_STREAM. The protocol argument must be specified as 0. The sockfd array returns the file descriptors referring to the two connected sockets.

Specifying type as SOCK\_STREAM creates the equivalent of a bidirectional pipe (also known as a stream pipe). Each socket can be used for both reading and writing, and separate data channels flow in each direction between the two sockets.

Typically, a socket pair is used in a similar fashion to a pipe. After the socketpair() call, the process then creates a child via fork(). The child inherits copies of the parent’s file descriptors, including the descriptors referring to the socket pair. Thus, the parent and child can use the socket pair for IPC.

One way in which the use of socketpair() differs from creating a pair of connected sockets manually is that the sockets are not bound to any address. This can help us avoid a whole class of security vulnerabilities, since the sockets are not visible to any other process.