CSE 344 System Programming

Socket Programming

- Sockets : An Introduction
- Unix Domains
- Fundamentals of TCP/IP Networks

Overview

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.

A socket is created using the *socket()* system call, which returns a file descriptor used to refer to the socket in subsequent system calls:

fd = socket(domain, type, protocol);

Communication Domain

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).

Sockets

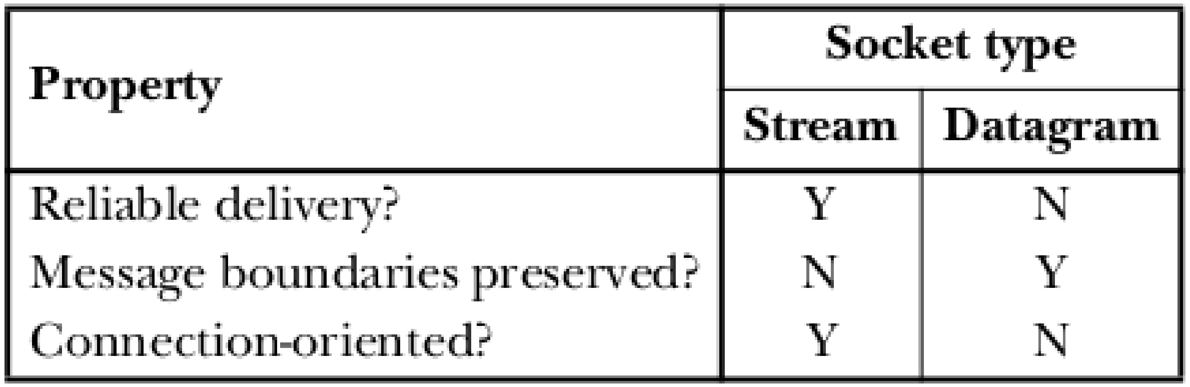
As of today modern operating systems support at least the following domains:

- The *UNIX* (AF_UNIX) domain allows communication between applications on the same host.
- The *IPv4* (AF_INET) domain allows communication between applications running on hosts connected via an Internet Protocol version 4 (IPv4) network.
- The *IPv6* (AF_INET6) domain allows communication between applications running on hosts connected via an Internet Protocol version 6 (IPv6) network.

*Although IPv6 is designed as the successor to IPv4, the latter protocol is still widely used.

Socket Types

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



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

Stream Sockets

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 , 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

- A stream socket is similar to using a pair of pipes to allow bidirectional communication between two applications, with the difference that (Internet domain) sockets permit communication over a network.
- Stream sockets operate in connected pairs. For this reason, stream sockets are described as connection-oriented. The term **peer socke**t 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. Sometimes, the term **remote** is used synonymously with **peer**.
- Analogously, sometimes the term **local** is used to refer to the application, socket, or address for this end of the connection. A stream socket can be connected to only one peer

Datagram Sockets

- 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.
- Unlike a stream socket, a datagram socket doesn't need to be connected to another socket in order to be used.
- In the Internet domain, datagram sockets employ the User Datagram Protocol (UDP), and stream sockets (usually) employ the Transmission Control Protocol (TCP).
- Instead of using the terms Internet domain datagram socket and Internet domain stream socket, we'll often just use the terms **UDP** socket and **TCP** socket, respectively.

Socket System Calls

The key socket system calls are the following:

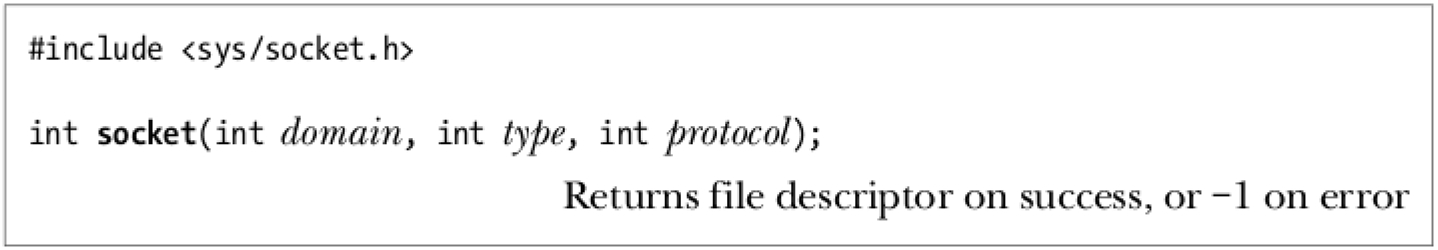
- The ***socket()*** system call creates a new socket.
- The ***bind()*** system call binds a socket to an address. Usually, a server employs this call to bind its socket to a well-known address so that clients can locate the socket.
- The ***listen()*** system call allows a stream socket to accept incoming 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.

Socket I/O

- Socket I/O can be performed using the conventional *read()* and *write()* system calls, or using a range of socket-specific system calls (e.g., *send()*, *recv()*, *sendto()*, and *recvfrom()*).
- By default, these system calls block if the I/O operation can’t be completed immediately.
- Nonblocking I/O is also possible, by using the *fcntl()* F_SETFL operation to enable the O_NONBLOCK open file status flag.

Creating A Socket

- The *socket()* system call creates a new socket



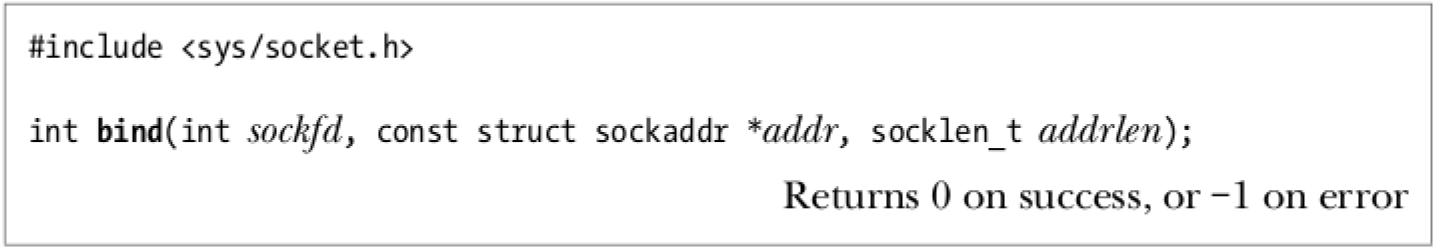
```
#include <sys/socket.h>

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

Returns file descriptor on success, or −1 on error

- 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 specified in this course. Nonzero protocol values are used with some socket types that we don’t describe. For example, protocol is specified as IPPROTO_RAW for raw sockets (SOCK_RAW).
- On success, *socket()* returns a file descriptor used to refer to the newly created socket in later system calls.

Binding a Socket to an Address

The *bind()* system call binds a socket to an address.



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

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

Returns 0 on success, or –1 on error

The *sockfd* argument is a file descriptor obtained from a previous call to socket(). 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 *addrlen* argument specifies the size of the address structure.

Generic Socket Address Structures

- The addr and addrlen arguments to *bind()* require some further explanation.
- For each socket domain, a different structure type is defined to store a socket address. However, because system calls such as *bind()* are generic to all socket domains, they must be able to accept address structures of any type. In order to permit this, the sockets API defines a generic address structure, *struct sockaddr*.
- The only purpose for this type is to cast the various domain-specific address structures to a single type for use as arguments in the socket system calls.

struct sockaddr

The sockaddr structure is typically defined as follows:

```
struct sockaddr {
    sa_family_t sa_family;      /* Address family (AF_* constant) */
    char        sa_data[14];    /* Socket address (size varies
                                   according to socket domain) */
};
```

- This structure serves as a template for all of the domain-specific address structures.
- Each of these address structures begins with a family field corresponding to the *sa_family* field of the *sockaddr* structure.
- The value in the family field is sufficient to determine the size and format of the address stored in the remainder of the structure.

struct sockaddr

struct sockaddr is a super-type.

In practice we use sub-types (specific to the context), that we then cast into struct sockaddr

e.g. struct sockaddr_un, struct sockaddr_in, etc

check the bind man page for details.

Stream Sockets

The operation of stream sockets can be explained by analogy with the telephone system:

- The *socket()* system call, which creates a socket, is the equivalent of installing a telephone. In order for two applications to communicate, each of them must create a socket.
- Communication via a stream socket is analogous to a telephone call. One application must connect its socket to another application's socket before communication can take place.
- Once a connection has been established, data can be transmitted in both directions between the applications until one of them closes the connection using *close()*. Communication is performed using the conventional *read()* and *write()* system calls or via a number of socket specific system calls (such as *send()* and *recv()*) that provide additional functionality.

Stream Sockets

Two sockets are connected as follows:

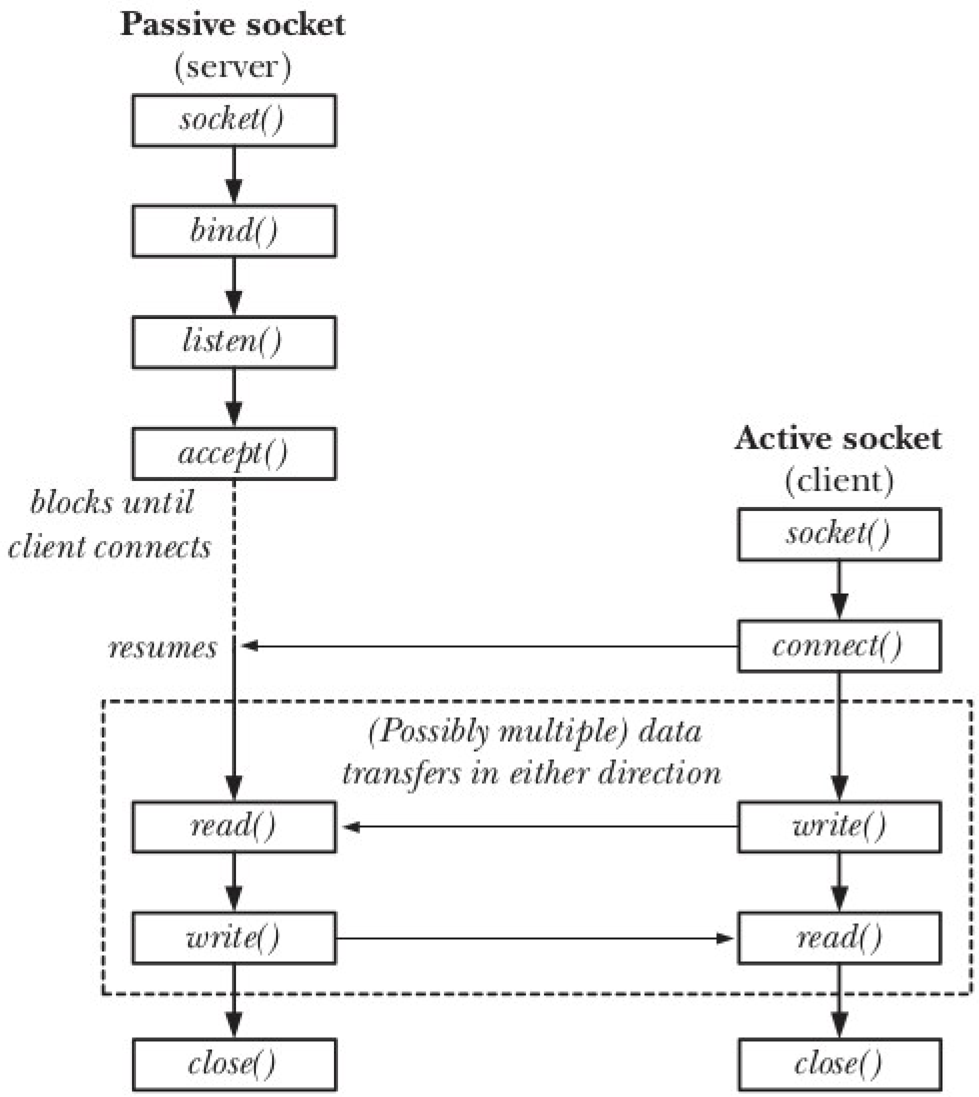
- One application calls *bind()* in order to bind the socket to a well-known address, and then calls *listen()* to notify the kernel of its willingness to accept incoming connections.
- The other application establishes the connection by calling *connect()*, specifying the address of the socket to which the connection is to be made.
- The application that called *listen()* then accepts the connection using *accept()*. **If the *accept()* is performed before the peer application calls *connect()*, then the *accept()* blocks**

Stream Sockets

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

- By default, 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 (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

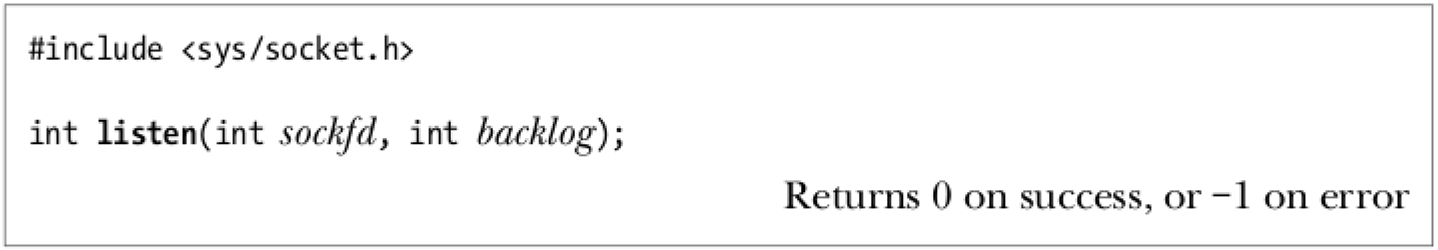
Overview of systems calls : Stream Sockets



Listenning for incomming Connections

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.



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

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

Returns 0 on success, or –1 on error

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()*.

The kernel must record some information about each pending connection request so that a subsequent accept() can be processed. **The backlog argument allows us to limit the number of such pending connections**

Accepting a Connection

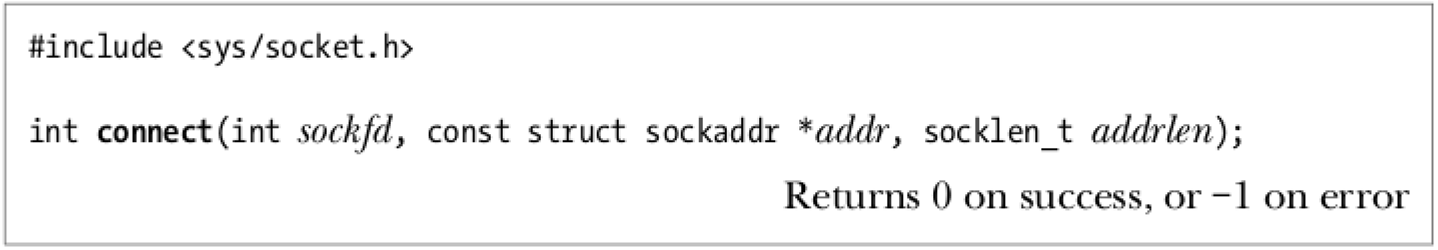
The *accept()* system call accepts an incoming connection on the listening stream socket referred to by the file descriptor sockfd. If there are no pending connections when *accept()* is called, the call blocks until a connection request arrives.

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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
                                  Returns file descriptor on success, or –1 on error
```

<u>Note that *accept()* creates a new socket, and it is this new socket that is connected to the peer socket that performed the *connect()*.</u> 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.

Connecting to a Peer Socket

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*.



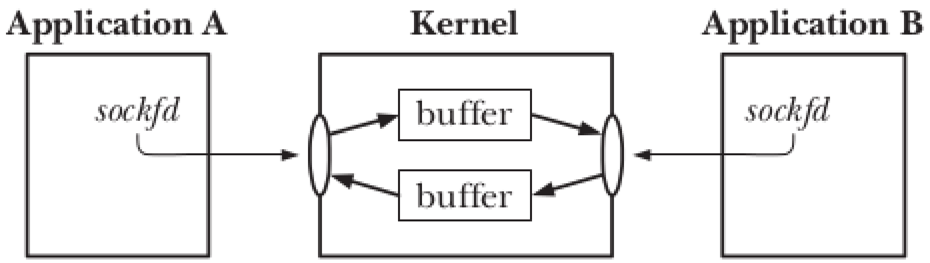
```
#include <sys/socket.h>

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

Returns 0 on success, or −1 on error

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

I/O on Stream Sockets

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 use the read() and write() system calls. Since sockets are bidirectional both calls may be used on each end of the connection.

Terminating a Connection

- The usual way of terminating a stream socket connection is to call **close()**. If multiple file descriptors refer to the same socket, then the connection is terminated when all of the descriptors are closed.
- Suppose that, after we close a connection, the peer application crashes or otherwise fails to read or correctly process the data that we previously sent to it. In this case, we have no way of knowing that an error occurred. If we need to ensure that the data was successfully read and processed, then we must build some type of acknowledgement protocol into our application. **This normally consists of an explicit acknowledgement message passed back to us from the peer.**

Datagram Sockets

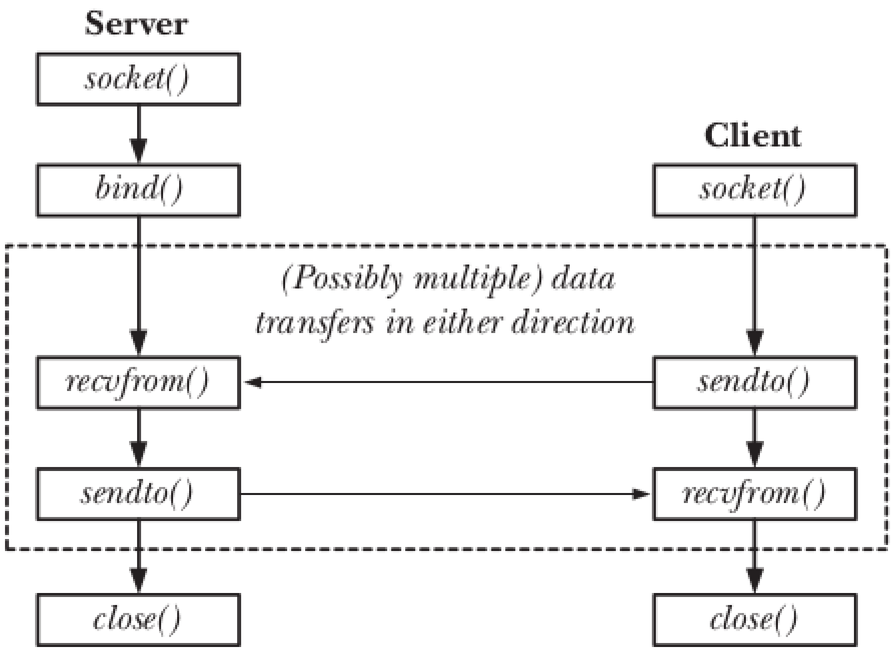
The operation of datagram sockets can be explained by analogy with the postal system:

- 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

Datagram Sockets

- 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.
- 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.
- When the socket is no longer needed, the application closes it using *close()*.
- There is no guarantee that they will arrive in the order they were sent, or even arrive at all. Since the underlying networking protocols may sometimes retransmit a data packet, the same datagram could arrive more than once.

Datagram Sockets



Overview of system calls used with datagram sockets

Exchanging Datagrams

The *recvfrom()* and *sendto()* system calls receive and send datagrams on a datagram socket.

```
#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 return value and the first three arguments to these system calls are the same as for *read()* and *write()*.

Using *connect()* with Datagram Sockets

- Even though datagram sockets are connectionless, the *connect()* system call serves a purpose when applied to datagram sockets.
- Calling *connect()* on a datagram socket causes the kernel to record a particular address as this socket’s peer. The term connected datagram socket is applied to such a socket. The term unconnected datagram socket is applied to a datagram socket on which *connect()* has not been called

Using *connect()* with Datagram Sockets

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.
- 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.

Dealing with endian-ness across systems

Sending/receiving data across systems ignoring endian-ness is a recipe for disaster. Example:

e.g. an Intel (Little-endian) system sending a word to a Motorola system (Big-endian).

You could of course solve this by first sending some scouting messages to figure out the endian-ness of the target system, but there is a more standard approach.

Dealing with endian-ness across systems

Before sending anything you always convert it to “network order” (big-endian).

And after receiving something, you first convert it into “host order” (whatever the endian-ness of the local host is..) and then use it.

16bit: ntohs and htons

32bit: ntohl and htonl

e.g. the “uint16_t htons(uint16_t n)” function converts the unsigned short integer n from host byte order to network byte order.

Summary

- A typical **stream socket server** creates its socket using **socket()**, and then binds the socket to a well-known address using **bind()**. The server then calls **listen()** to allow connections to be received on the socket. Each client connection is then accepted on the listening socket using **accept()**, which returns a file descriptor for **a new socket** that is connected to the client's socket.
- A typical **stream socket client** creates a socket using **socket()**, and then establishes a connection by calling **connect()**, specifying the server's well-known address. After two stream sockets are connected, data can be transferred in either direction using **read()** and **write()**. Once all processes with a file descriptor referring to a stream socket endpoint have performed an implicit or explicit **close()**, the connection is terminated.

Summary

- A typical **datagram socket server** creates a socket using ***socket()***, and then binds it to a well-known address using ***bind()***. Because datagram sockets are connectionless, the server's socket can be used to receive datagrams from any client.
- Datagrams can be received using ***read()*** or using the socket specific ***recvfrom()*** system call, which returns the address of the sending socket. A datagram socket client creates a socket using ***socket()***, and then uses ***sendto()*** to send a datagram to a specified address. The ***connect()*** system call can be used with a datagram socket to set a peer address for the socket. After doing this, it is no longer necessary to specify the destination address for outgoing datagrams; a ***write()*** call can be used to send a datagram