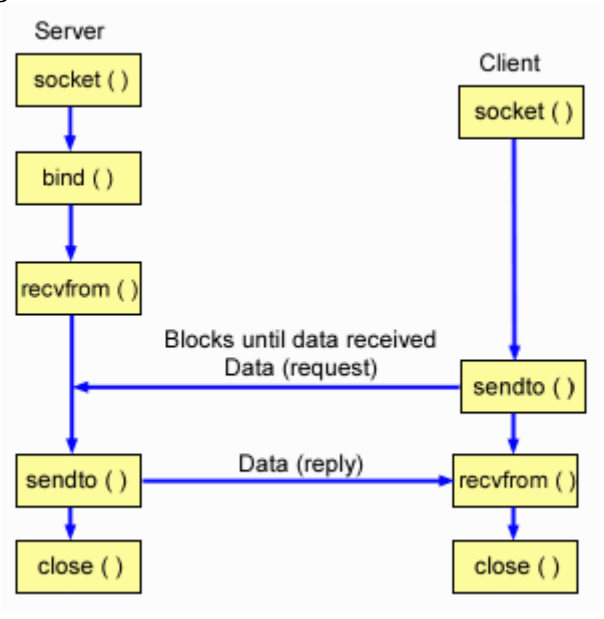
Computer Networks (CS425)

Instructor: Dr. Dheeraj Sanghi

Unix Socket Programming (Contd..)

Sequence of System Calls for Connectionless Communication

The typical set of system calls on both the machines in a connectionless setup is shown in Figure below.



- The **socket** and **bind** system calls are called in the same way as in the connection-oriented case. Again the bind call is optional at the client side.
- The **connect** function is not called in a connectionless communication with the sane intention as above. Instead, if we call a connect() in this case, then we are simply specifying a particular server address to which we have to send, and from which we have to receive the Datagrams
- Every time a packet has to be sent over a socket, the remote address has to be mentioned. This is because there is no concept of a connection that can remember which remote machine to send that packet to.
- The calls **sendto** and **recvfrom** are used to send datagram packets. The synopses of both are given below.

  int sendto(int skfd, void *buf, int buflen, int flags, struct sockaddr* to, int tolen); int recvfrom(int skfd, void *buf, int buflen, int flags, struct sockaddr* from, int fromlen);

sendto sends a datagram packet containing the data present in buf addressed to the address present in the **sockaddr** structure, to.

recvfrom fills in the buf structure with the data received from a datagram packet and the **sockaddr** structure, from with the address of the client from which the packet was received.

Both these calls block until a packet is sent in case of **sendto** and a packet is received in case of **recvfrom**. In the strict sense though **sendto** is not blocking as the packet is sent out in most cases and **sendto** returns immediately.

- Suppose if the program desires to communicate only to one particular machine and make the operating system discard packets from all other machines, it can use the connect call to specify the address of the machine with which it will exclusively communicate. All subsequent calls do not require the address field to be given. It will be understood that the remote address is the one specified in connect called earlier.

Socket Options and Settings

There are various options which can be set for a socket and there are multiple ways to set options that affect a socket.

Of these, **setsockopt()** system call is the one specifically designed for this purpose. Also, we can retrieve the option which are currently set for a socket by means of **getsockopt()** system call.

int setsockopt(int socket, int level, int option_name, const void *option_value, socklen_t option_len);

The socket argument must refer to an open socket descriptor. The level specifies who in the system is to interpret the option: the general socket code, the TCP/IP code, or the XNS code. This function sets the option specified by the option_name, at the protocol level specified by the level, to the value pointed to by the option_value for the socket associated with the file descriptor specified by the socket. The level argument specifies the protocol level at which the option resides. To set options at the socket level, we need to specify the level argument as **SOL_SOCKET**. To set options at other levels, we need to supply the appropriate protocol number for the protocol controlling the option. The option_name specifies a single option to set. The option_name and any specified options are passed uninterpreted to the appropriate protocol module for interpretations. The list of options available at the socket level **(SOL_SOCKET)** are:

- **SO_DEBUG**

Turns on recording of debugging information. This option enables or disables debugging in the underlying protocol modules. This option takes an int value. This is a boolean option.

- **SO_BROADCAST**

Permits sending of broadcast messages, if this is supported by the protocol. This option takes an int value. This is a boolean option.

- **SO_REUSEADDR**

Specifies that the rules used in validating addresses supplied to bind() should allow reuse of local addresses, if this is supported by the protocol. This option takes an int value. This is a boolean option.

- **SO_KEEPALIVE**

Keeps connections active by enabling the periodic transmission of messages, if this is supported by the protocol. This option takes an int value. If the connected socket fails to respond to these messages, the connection is broken and processes writing to that socket are notified with a SIGPIPE signal. This is a boolean option.

- **SO_LINGER**

Lingers on a close() if data is present. This option controls the action taken when unsent messages queue on a socket and close() is performed. If SO_LINGER is set, the system blocks the process during close() until it can transmit the data or until the time expires. If SO_LINGER is not specified, and close() is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option takes a linger structure, as defined in the <sys/socket.h> header, to specify the state of the option and linger interval.

- **SO_OOBINLINE**

Leaves received out-of-band data (data marked urgent) in line. This option takes an int value. This is a boolean option.

- **SO_SNDBUF**

Sets send buffer size. This option takes an int value.

- **SO_RCVBUF**

Sets receive buffer size. This option takes an int value.

- **SO_DONTROUTE**

Requests that outgoing messages bypass the standard routing facilities. The destination must be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option takes an int value. This is a boolean option.

- **SO_RCVLOWAT**

Sets the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different than that returned, e.g. out of band data). This option takes an int value. Note that not all implementations allow this option to be set.

- **SO_RCVTIMEO**

Sets the timeout value that specifies the maximum amount of time an input function waits until it completes. It accepts a timeval structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it returns with a partial count or errno set to [EAGAIN] or [EWOULDBLOCK] if no data were received. The default for this option is zero, which indicates that a receive operation will not time out. This option takes a timeval structure. Note that not all implementations allow this option to be set.

- **SO_SNDLOWAT**

Sets the minimum number of bytes to process for socket output operations. Non-blocking output operations will process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option takes an int value. Note that not all implementations allow this option to be set.

- **SO_SNDTIMEO**

Sets the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it returns with a partial count or with errno set to [EAGAIN] ore [EWOULDBLOCK] if no data were sent. The default for this option is zero, which indicates that a send operation will not time out. This option stores a timeval structure. Note that not all implementations allow this option to be set.

For boolean options, 0 indicates that the option is disabled and 1 indicates that the option is enabled.Options at other protocol levels vary in format and name.

Some of the options available for the **IP_PROTO_TCP** socket are:

- **TCP_MAXSEG**

Returns the maximum segment size in use for the socket.The typical value for a 43.BSD socket using an Ethernet is 1024 bytes.

- **TCP_NODELAY**

When TCP is being used for a remote login,there will be many small data packets sent from the client's system to the server.Each packet can contain a single character that the user enters which is sent to the server for echoing and processing.It might be desirable to reduce the number of such small packets by combining a number of them into one big packet.But this causes a delay between the typing of a character by the user and its appearance on its monitor.This is certainly not something the user will appreciate. For such services it is desirable that the client's packets be sent as soon as they are ready.The **TCP_NODELAY** option is used for these clients to defeat the buffering algorithm, and allow the client's TCP to send small packets as soon as possible.

int getsockopt(int socket, int level, int option_name, void *option_value, socklen_t *option_len);

This function retrieves the value for the option specified by the option_name argument for the socket. If the size of the option value is greater than option_len, the value stored in the object pointed to by the option_value will be silently truncated. Otherwise, the object pointed to by the option_len will be modified to indicate the actual length of the value. The level specifies the protocol level at which the option resides. To retrieve options at the socket level, we need to specify the level argument as **SOL_SOCKET.** To retrieve options at other levels, we need to supply the appropriate protocol number for the protocol controlling the option. The socket in use may require the process to have appropriate privileges to use the **getsockopt()** function. The list of options for option_name is the same as those available for **setsockopt()** system call.

Servers

Normally a client process is started on the same system or on another system that is connected to the server's system with a network. Client processes are often initiated by the interactive user entering a command to a time-sharing system. The client process sends a request across the network to the server requesting service of some form. In this way, normally a server handles only one client at a time. If multiple client connections arrive at about the same time, the kernel queues them upto some limit, and returns them to accept function one at a time. But if the server takes more time to service each client (say a few seconds or minutes), we would need some way to overlap the service of one client with another client. Multiple requests can be handled in two ways. In fact servers are classified on the basis of the way they handle multiple requests.

1. **Iterative Servers:** When a client's request can be handled by the server in a known, finite amount of time, the server process handles the request itself. These servers handles one client at a time by iterating between them.
2. **Concurrent Servers:** These servers handle multiple clients at the same time. Among the various approaches that are available to handle these multiple clients, simplest approach is to call the UNIX fork system call ,

creating a child process for each client. When a connection is established, accept returns, the server calls fork, and then the child process services the client and the parent process waits for another connection. The parent closes the connected socket since the child handles this new client.

Internet Superserver (inetd)

Servers must keep running at all times. However, all the servers are not working all this time but merely waiting for a request from a client. To avoid this waste of resources, a single server is run which waits on all the port numbers. This Internet super server is called **inetd** in UNIX. It is referred to as the ``Internet Super-Server'' because it manages connections for several daemons. Programs that provide network service are commonly known as daemons. **inetd** serves as a managing server for other daemons. When a connection is received by **inetd,** it determines which daemon the connection is destined for, spawns the particular daemon and delegates the socket to it. Running one instance of **inetd** reduces the overall system load as compared to running each daemon individually in stand-alone mode. This daemon provides two features -

1. It allows a single process **(inetd)** to be waiting to service multiple connection requests, instead of one process for each potential service. This reduces the total number of processes in the system.

2. It simplifies the writing of the server processes to handle the requests, since many of the start-up details are handled by **inetd.**

The **inetd** process uses fork and exec system calls to invoke the actual server process.The only way the server can obtain the identity of the client is by calling the **getpeername** system call.

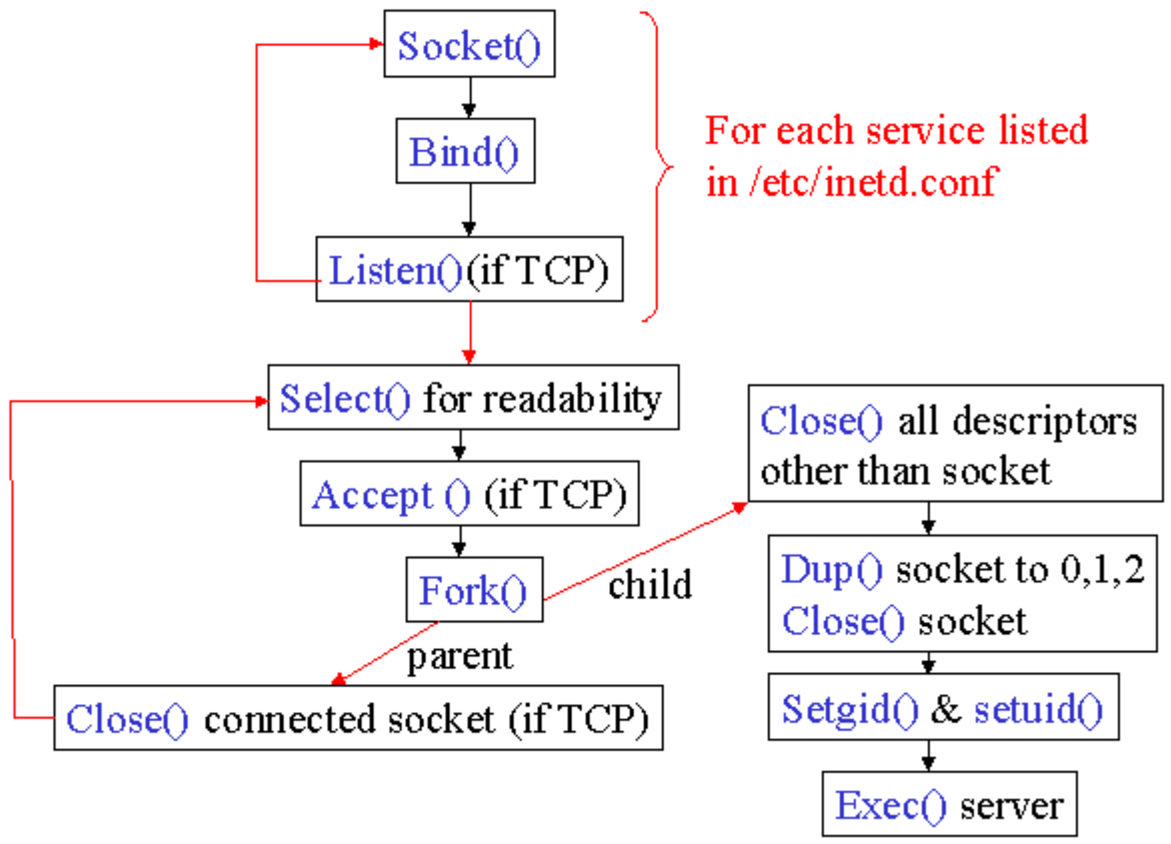
int getpeername(int sockfd, struct sockaddr *peer, int *addrlen);

Getpeername returns the name of the peer connected to socket *sockfd*. The *addrlen* parameter should be initialised to indicate the amount of space pointed to by peer . On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

A similar system call is **getsockname. Getsockname** returns the current name for the specified socket. The *addrlen* parameter should be initialized to indicate the amount of space pointed to by name. On return it contains the actual size of the name returned (in bytes).

int getsockname(int sockfd, struct sockaddr *name, int *addrlen);

Following illustrates the steps performed by **inetd**



However, the **inetd** process has its own disadvantages. The code of a server is to be copied from the disk to memory each time it is execed, and this is expensive. So, if there is an application which is called frequently, like e-mail server, the above mentioned approach (using **inetd**) is not recommended.

Image References

- http://publib.boulder.ibm.com/iseries/v5r1/ic2924/info/rzab6/rxab6503.gif
- http://www.cs.odu.edu/~cs779/spring03/lectures/inetd.gif

---

back to top
Prev| Next | Index