ELEN 602: Computer Communications and Networking

Socket Programming Basics

A. Introduction

In the classic client-server model, the client sends out requests to the server, and the server does some processing with the request(s) received, and returns a reply (or replies) to the client. The terms request and reply here may take on different meanings depending upon the context, and method of operation. An example of a simple and ubiquitous client-server application would be that of a web-server. A client (Internet Explorer, or Netscape) sends out a request for a particular web page, and the web-server (which may be geographically distant, often in a different continent!) receives and processes this request, and sends out a reply, which in this case, is the web page that was requested. The web page is then displayed on the browser (client).

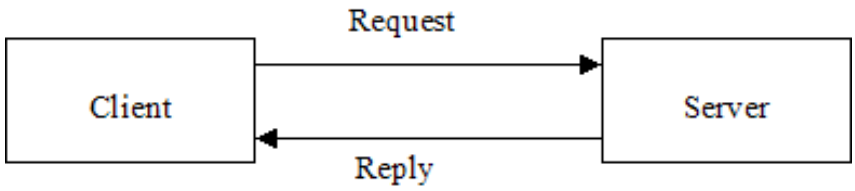

Fig. 1. Client-Server Paradigm

Further, servers may be broadly classified into two types based on the way they serve requests from clients. *Iterative Servers* can serve only one client at a time. If two or more clients send in their requests at the same time, one of them has to wait until the other client has received service. On the other hand, *Concurrent Servers*

can serve multiple clients at the same time. Typically, this is done by spawning off a new server process on the receipt of a request - the original process goes back to listening to new connections, and the newly spawned off process serves the request received.

We can realize the client-server communication described above with a set of network protocols, like the TCP/IP protocol suite, for instance. In this tutorial, we will look at the issue of developing applications for realizing such communication over a network. In order to write such applications, we need to understand sockets.

B. What are sockets?

Sockets (also called Berkeley Sockets, owing to their origin) can simply be defined as end-points for communication. To provide a rather crude visualization, we could imagine the client and server hosts in Figure 1 being connected by a pipe through which data-flow takes place, and each end of the pipe can now be construed as an "end-point". Thus, a socket provides us with an abstraction, or a logical end point for communication. There are different types of sockets. *Stream Sockets*, of type SOCK_STREAM are used for connection oriented, TCP connections, whereas *Datagram Sockets* of type SOCK_DGRAM are used for UDP based applications. Apart from these two, other socket types like SOCK_RAW and SOCK_SEQPACKET are also defined.

C. Socket layer, and the Berkeley Socket API

Figure 2 shows the TCP/IP protocol stack, and shows where the "Socket layer" may be placed in the stack. Again, please be advised that this is just a representation to indicate the level at which we operate when we write network programs using

sockets. As shown in the figure, sockets make use of the lower level network protocols, and provide the application developer with an interface to the lower level network protocols. A library of system calls are provided by the socket layer, and are termed as the "Socket API". These system calls can be used in writing socket programs. In the sections that follow, we will study these system calls in detail.

We shall study socket programming in the UNIX environment.

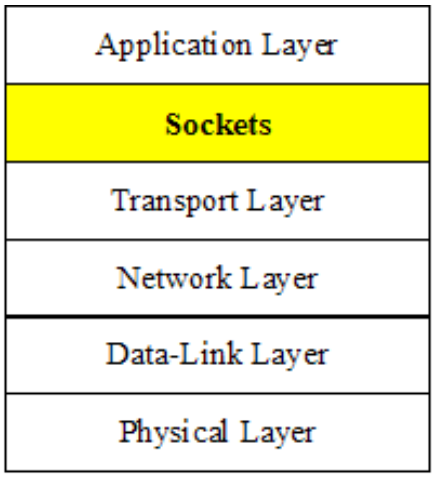

Fig. 2. TCP/IP Protocol Stack.

D. Basic Socket system calls

Figure 3 shows the sequence of system calls between a client and server for a connection-oriented protocol. Let us take a detailed look at some of the socket system calls:

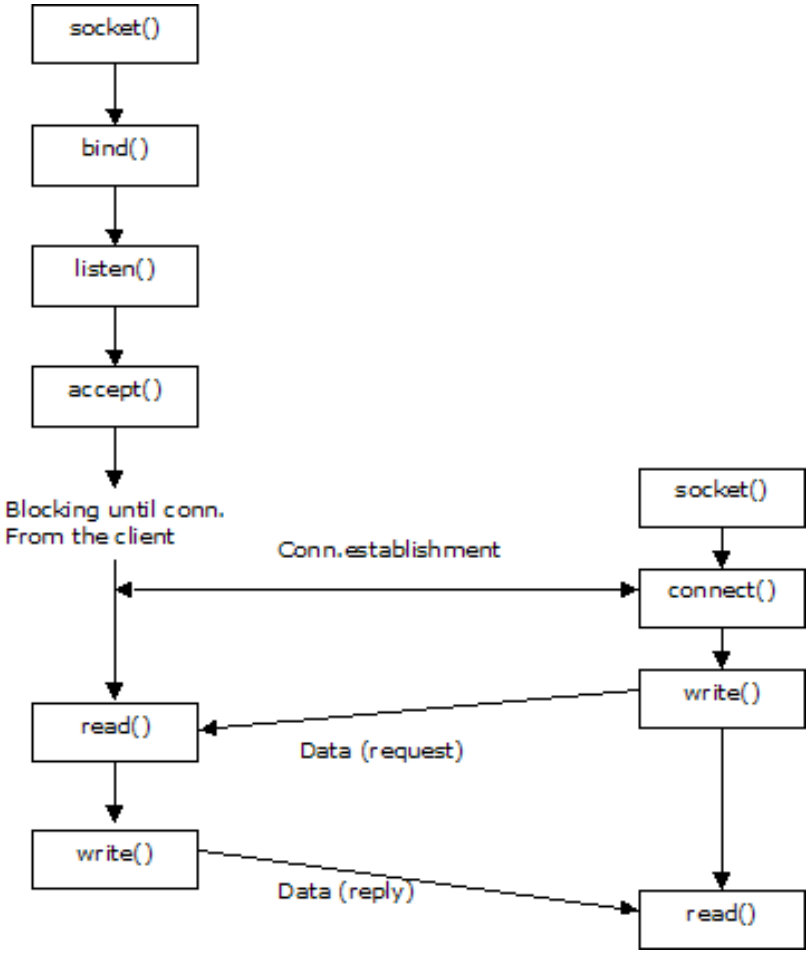

Fig. 3. Socket system calls for connection-oriented case.

1. The `socket()` system call

`socket()` system call syntax:

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

The `socket()` system call creates a socket and returns a socket file descriptor to the socket created. The descriptor is of data type `int`.

Here, `domain` is the address family specification, `type` is the socket type and the protocol field is used to specify the protocol to be used with the address family specified. The address family can be one of AF_INET (for Internet protocols like TCP, UDP, which is what we are going to use) or AF_UNIX (for Unix internal protocols), AF_NS, for Xerox network protocols or AF_IMPLINK for the IMP link layer. (The `type` field is the socket type - which may be SOCK_STREAM for stream sockets (TCP connections), or SOCK_DGRAM (for datagram connections). Other socket types are defined too. SOCK_RAW is used for raw sockets, and SOCK_SEQPACKET is used for a sequenced packet socket. The `protocol` argument is typically set to 0. You may also specify a protocol argument to use a specific protocol for your application.

2. The `bind()` system call

The `bind()` system call is used to specify the association <Local-Address, Local-Port>. It is used to bind either connection oriented or connectionless sockets. The `bind()` function basically associates a name to an unnamed socket. "Name", here refers to three components - The address family, the host address, and the port number at which the application will provide its service. The syntax and arguments taken by the bind system call is given below:

```
int result = bind(int sd, struct sockaddr *address, int addrlen);
```

Here, `sd` is the socket file descriptor returned by the `socket()` system call before, and `name` points to the `sockaddr` structure, and `addrlen` is the size of the `sockaddr` structure.

Like all other socket system calls, upon success, `bind()` returns 0. In case of error, `bind()` returns -1.

3. The `listen()` system call

After creation of the socket, and binding to a local port, the server has to wait on incoming connection requests. The `listen()` system call is used to specify the queue or backlog of waiting (or incomplete) connections. The syntax and arguments taken by the `listen()` system call is given below:

```
int result = listen(int sd, int backlog);
```

Here, `sd` is the socket file descriptor returned by the `socket()` system call before, and `backlog` is the number of incoming connections that can be queued.

Upon success, `listen()` returns 0. In case of error, `listen()` returns -1.

4. The `accept()` system call

After executing the `listen()` system call, a server waits for incoming connections. An actual connection setup is completed by a call to `accept()`. `accept()` takes the first connection request on the queue, and creates another socket descriptor with the same properties as `sd` (the socket descriptor returned earlier by the `socket()` system call). The new socket descriptor handles communications with the new client while the earlier socket descriptor goes back to listening for new connections. In a sense, the `accept()` system call completes the connection, and at the end of a successful `accept()`, all elements of the four tuple (or the five tuple - if you consider “protocol”

as one of the elements) of a connection are filled. The "four-tuple" that we talk about here is <Local Addr, Local Port, Remote Addr, Remote Port>. This combination of fields is used to uniquely identify a flow or a connection. The fifth tuple element can be the protocol field. No two connections can have the same values for all the four (or five) fields of the tuple. `accept()` syntax:

```
int newsd = accept(int sd, void *addr, int *addrlen);
```

Here, `sd` is the socket file descriptor returned by the `socket()` system call before, and `addr` is a pointer to a structure that receives the address of the connecting entity, and `addrlen` is the length of that structure.

Upon success, `accept()` returns a socket file descriptor to the new socket created. In case of error, `accept()` returns -1.

5. The `connect()` system call

A client process also starts out by creating a socket by calling the `socket()` system call. It uses `connect()` to connect that socket descriptor to establish a connection with a server. In the case of a connection oriented protocol (like TCP/IP), the `connect()` system call results in the actual connection establishment of a connection between the two hosts. In case of TCP, following this call, the three-way handshake to establish a connection is completed. Note that the client does not necessarily have to bind to a local port in order to call `connect()`. Clients typically choose ephemeral port numbers for their end of the connection. Servers, on the other hand, have to provide service on well-known (premeditated) port numbers. `connect()` syntax:

```
int result = connect(int sd, struct sockaddr *servaddr, int addrlen);
```

Here, `sd` is the socket file descriptor returned by the `socket()` system call be-

fore, `servaddr` is a pointer to the server's address structure (port number and IP address). `addrlen` holds the length of this parameter and can be set to `sizeof(struct sockaddr)`.

Upon success, `connect()` returns 0. In case of error, `connect()` returns -1.

6. The `send()`, `recv()`, `sendto()` and `recvfrom()` system calls

After connection establishment, data is exchanged between the server and client using the system calls `send()`, `recv()`, `sendto()` and `recvfrom()`. The syntax of the system calls are as below:

```
int nbytes  =  send(int sd, const void *buf, int len, int flags);
int nbytes  =  recv(int sd, void *buf, int len, unsigned int flags);
int nbytes  =  sendto(int sd, const void *buf, int len, unsigned int
                      flags, const struct sockaddr *to, int tolen);
int nbytes  =  recvfrom(int sd, void *buf, int len, unsigned int
                        flags, struct sockaddr *from, int *fromlen);
```

Here, `sd` is the socket file descriptor returned by the `socket()` system call before, `buf` is the buffer to be sent or received, `flags` is the indicator specifying the way the call is to be made (usually set to 0). `sendto()` and `recvfrom()` are used in case of connectionless sockets, and they do the same function as `send()` and `recv()` except that they take more arguments (the "to" and "from" addresses - as the socket is connectionless)

Upon success, all these calls return the number of bytes written or read. Upon failure, all the above calls return -1. `recv()` returns 0 if the connection was closed by the remote side.

7. The `close()` system call

The `close()` system call is used to close the connection. In some cases, any remaining data that is queued is sent out before the `close()` is executed. The `close()` system call prevents further reads or writes to the socket. `close()` syntax:

```
int result = close (int sd);
```

Here, `sd` is the socket file descriptor returned by the `socket()` system call before.

8. The `shutdown()` system call

The `shutdown()` system call is used to disable sends or receives on a socket. `shutdown()` syntax:

```
int result = shutdown (int sd, int how);
```

Here, `sd` is the socket file descriptor returned by the `socket()` system call before. The parameter `how` determines how the shutdown is achieved. If the value of the parameter `how` is 0, further receives on the socket will be disallowed. If `how` is 1, further sends are disallowed, and finally, if `how` is 2, both sends and receives on the socket are disallowed. Remember that the `shutdown()` function does not close the socket. The socket is closed and all the associated resources are freed only after a `close()` system call. Upon success, `shutdown()` returns 0. In case of error, `shutdown()` returns -1.

E. Byte ordering, and byte ordering routines

When data is transmitted on networks, the byte ordering of data becomes an issue. There are predominantly two kinds of byte orderings - *Network Byte Order* (Big-Endian byte order) and *Host Byte Order* (Little-Endian byte order). The network byte order has the most significant byte first, while the host byte order has the least

significant byte first. Different processor architectures use different kinds of byte orderings. Data transmitted on a network is sent in the network byte order. Hence, because of disparities among different machines, when data needs to be transmitted over a network, we need to change the byte ordering to the network byte order. The following routines help in changing the byte order of the data:

```
u_short result  =  ntohs (u_short netshort);
u_short result  =  htons (u_short hostshort);
 u_long result  =  ntohl (u_long netlong);
 u_long result  =  htonl (u_long hostlong);
```

The `hton*` routines convert host-byte-order to the network-byte-order. The `ntoh*` routines do the opposite.

F. Important `structs`

This section has definitions for the structs used in the socket system calls.

```
struct sockaddr {
   unsigned short    sa_family;      // address family, AF_xxx
   char              sa_data [14];   // 14 bytes of protocol address
};
```

The `sockaddr` structure holds the socket address information for all types of sockets. The `sa_family` field can point to many address families. The Internet address family is denoted by `AF_INET`, and that encompasses most of the popular protocols we use (TCP/UDP), and so the Internet address specific parallel `sockaddr` structure is called `sockaddr_in`. The fields are self-explanatory. The `sin_zero` field serves as padding, and is typically set to all zeroes using `memset()`.

```
struct sockaddr_in {
   short int            sin_family;
   unsigned short int   sin_port;
   struct in_addr       sin_addr;
   unsigned char        sin_zero[8];
};
```
