EE 595- Tutorial on the Client/Server Paradigm and Socket Programming

Overview

Typical network applications are run in a *client*/*server* mode A client usually *initiates* a connection and requests service(s) from the server while a server usually *listens* and waits for incoming connection requests and provides services. An example will be a web browser (client) connecting to a web server and asking for an HTML file. The client/server application sits on top of an *application layer protocol*, which defines the messages that are exchanged by the peer processes and the actions taken. HTTP (HyperText Transfer Protocol) is an example of an application layer protocol. It defines the (HTTP) transactions between a web browser and a web server. An application layer process in a remote host (either client or server) uses the services provided by the transport layer, to communicate with its peer process running in the other remote host (either server or client).

Background Material

A *process* is a program running in a host. The communication between two processes within the same host is done by *Inter-Process Communication (IPC)* defined by the operating system (OS), while the communication between two (peer) processes running in different hosts is defined by an *application layer protocol*. Application layer protocols define the messages exchanged and actions taken by the two peer processes, regardless of what OS each host is running.

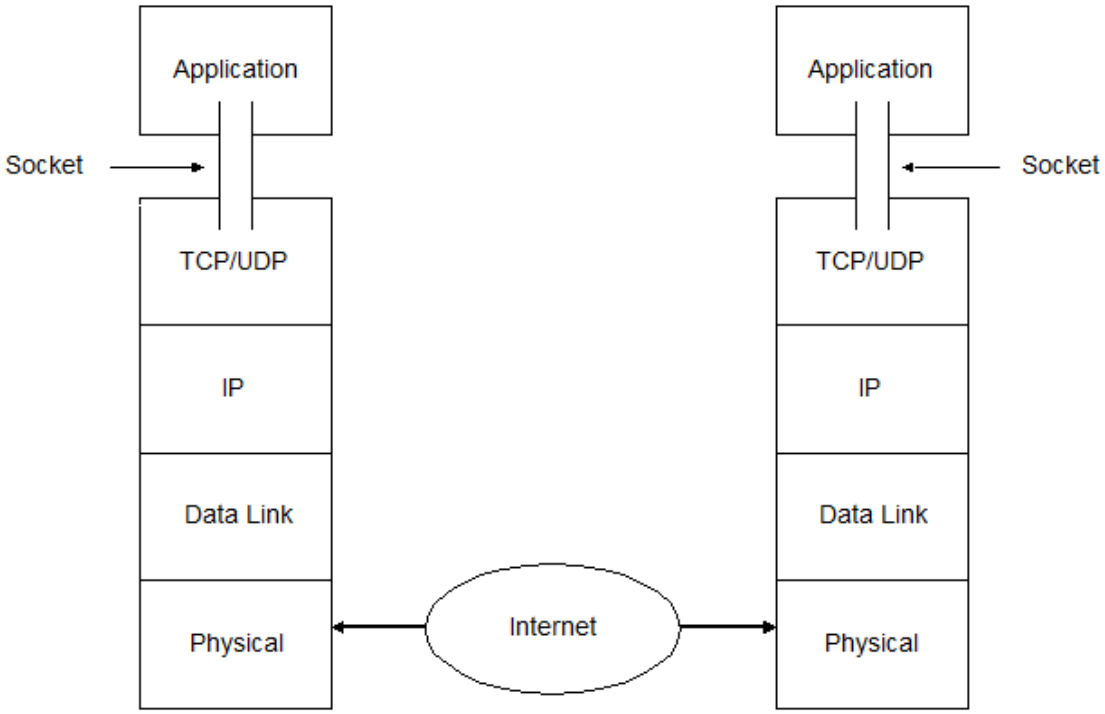

Figure 1 Sockets in the TCP/IP PRM

Figure 1 Sockets in the TCP/IP PRM

In principle, a connection is uniquely defined by a quadruple: source IP address, source port number, destination IP address, destination port number. (Remember that a port number is necessary for multiplexing and de-multiplexing at the transport layer.) The two values that identify each endpoint of a connection — an IP address and a port number — are often called a *socket*. *A socket pair* fully identifies a given instantiation of an application and connects the application protocol to TCP or UDP, which provide a streaming (connection-oriented) or a datagram (connectionless) service respectively. Programmers must declare which type of service they want when the sockets are created. When a socket is being created, a *socket identifier* is returned. It is a (local) value assigned by the OS for each socket which is mapped to a particular connection when the connection is set up. Programmers will use this value to identify a particular connection for data transfer, so the use of it is like a file handler.

From the programmer's point-of-view, all network protocols below the transport layer are *transparent* – the application pushes data through the socket, and the remote

host retrieves the data received through its socket. The socket can be seen as the input and output of a logical pipe between the two end-hosts. The characteristics of the pipe depends upon which transport layer is used. If the application uses TCP, for example, the programmer does not have to worry and take care of retransmission, reordering, etc. All the data available to the socket on the remote host is exactly what the sender has sent. This enables the programmer to focus on the application development, and leave the data communication details to the operating system.

Typically, a well known port number is dedicated to a particular application at the server side, since the server has to listen for incoming requests. For example, the port number 80 refers to an HTTP server. So an HTTP client that wants to request a service from a given HTTP server would create a local socket over TCP with its IP address and a port number obtained through its OS and use the IP address from the server and port number 80 to send its connection request. The server would receive this request on its permanently opened socket = (server IP address, port # = 80) and create a specific socket for this connection.

Sockets Programming:

Data representation and conversion:

Some computers use the *big-endian* convention to store their data. This refers to the representation of objects such as integers within a word. A big-endian machine stores them in the expected way: the high byte of an integer is stored in the leftmost byte, while the low byte of an integer is stored in the rightmost byte. For example, the representation of a 32-bit (long int) data in a Sun Sparc is using the big-endian convention, so the number 5 + 6 * 256 would be stored as 0065. A small-endian machine stores them the other way, for example, the i386 is using small-endian, so 5600 will be stored instead. If a Sparc sends an integer to an i386, the i386 will see 5 + 6 * 256 as 5 * (256^3) + 6 * (256^2). To avoid this, two communicating machines must agree on data representation.

Since IP uses big-endian as the method to represent data, it will be a good idea to convert all the data to big-endian at the sender side before passing them to the network. In that way we can ensure that all the data in the network is going to be stored in big-endian format. At the receiving end, if the host uses small-endian, then we know we have

to reorder the data to small-endian first before we can use it. Obviously, if the receiving host is using big-endian, data received from the network can be used right away.

To handle byte reordering for data, we need certain conversion functions. These functions should be able to convert the current format used in a host to big-endian, and convert back from big-endian to the format that the host is using. Which "endian" is the host using is sometime not known to the programmer, but the OS must know it. Therefore it is a good practice to always use these conversion functions to convert the data to big-endian format when you send and receive integers over the network, even if you are sure about what kind of representation of data the host is using. Besides, this will also make your program more portable to other platforms or machines. The functions will just be defined to null macros if the host uses big-endian, since no conversion is needed, and will "reverse" the byte-ordering if a host uses small-endian. The following functions are defined in the library `<netinet/in.h>` for converting 32-bit data (long integer):

- `htonl(long int)` – conversion from **h**ost format (machine dependent format) to **n**etwork format (big-endian)
- `ntohl(long int)` – conversion from **n**etwork format (big-endian) to **h**ost format (machine dependent format)

Although programmers cannot select which byte-ordering format to use (it is machine dependent), they have a choice of the precision of the data that they want to send. They can choose from sending `int` (16-bit integer) or `long int` (32-bit integer), depending on the need of the application. However, we cannot re-order 2-byte data the same way as we do for 4-byte data, so we have the following functions for 16-bit data:

- `htons(int)` - conversion from host format to network format, for **short** integer (16-bit)
- `ntohs(int)` - conversion from network format to host format, for **short** integer (16-bit)

It is the programmer's responsibility to ensure that the precision of data is consistent at both ends, i.e., when sending/receiving 16-bit data, make sure to call

`htons`/`ntohs` and not `htonl`/`ntohl` and when sending/receiving 32-bit data make sure to call `htonl`/`ntohl`.

Procedure to create TCP sockets

Before we describe the use of different socket functions, we will first describe some structures defined in `<netinet/in.h>` which are needed when using the functions. The following is the structure for a socket in an Internet setting, i.e., an IP address, a port number, a transport protocol, and an IP version (here IPv4):

```
struct sockaddr_in {
  unsigned char      sin_len;  /*length of structure (16 for IPv4)*/
  short int          sin_family;  /* AF_INET (IPv4) */
  unsigned short int sin_port;    /* 16-bit TCP/UDP Port number */
  struct in_addr     sin_addr;    /* 32-bit long IPv4 address */
  unsigned char      sin_zero[8]; /* Unused */
};

struct in_addr {
  unsigned long s_addr; /* 32-bit long IPv4 address */
};
```

Since IPv4 addresses are usually given in *dot format*, we need to convert them into network data, which are unsigned 32-bit integers defined in `struct in_addr` above, by the following function before we put it into the `s_addr` field:

```
int inet_aton(const char *ip, struct in_addr *inp);
```

where `ip` is the IPv4 address in dot format and `inp` is the return value.

The following is a list of header files you need to include in order to use socket functions, and a list of the functions we will use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int socket(int family, int type, int protocol);
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
int connect(int sockfd, struct sockaddr *serv_addr, int
addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, void *addr, int *addrlen);
int write(int sockfd, const void *msg, int len);
int read(int sockfd, void *buf, int len);
int close(int sockfd);
```

Below is a brief introduction of the above functions. For details, please refer to the corresponding man pages (for example, under linux operating system, you would type in `-bash-3.00$ man socket`). Figure 2 shows the flow and relation between the socket functions in the case *where the server deals with one connection at a time* (i.e., only one connection is being served at a time, a new connection can only be served after the current connection has been released):

`socket()`

- The function creates a socket (i.e., allocates resources such as buffer) and return a socket identifier (a nonnegative integer) if successful, -1 if unsuccessful.
- This function takes in three arguments. The first one is the protocol family, and the constant `AF_INET` is defined for IPv4 family, which will be used in this experiment; the second one is the type of connection, and the constant `SOCK_STREAM` is defined for TCP connections; the last one is set to `0` throughout this experiment.
- This function is required for both the server and the client.

`bind()`

- The function assigns a local protocol address to a socket (i.e., in IPv4 it would be the IP address, the port number, and the transport protocol).
- It takes three arguments. The first one is the socket identifier of the socket that will be bind-ed; the second one is the structure of the internet socket address defined earlier and the last one is the length of the structure.
- Only the server needs to call `bind()`, because in the client side, this is done by the function `connect`.

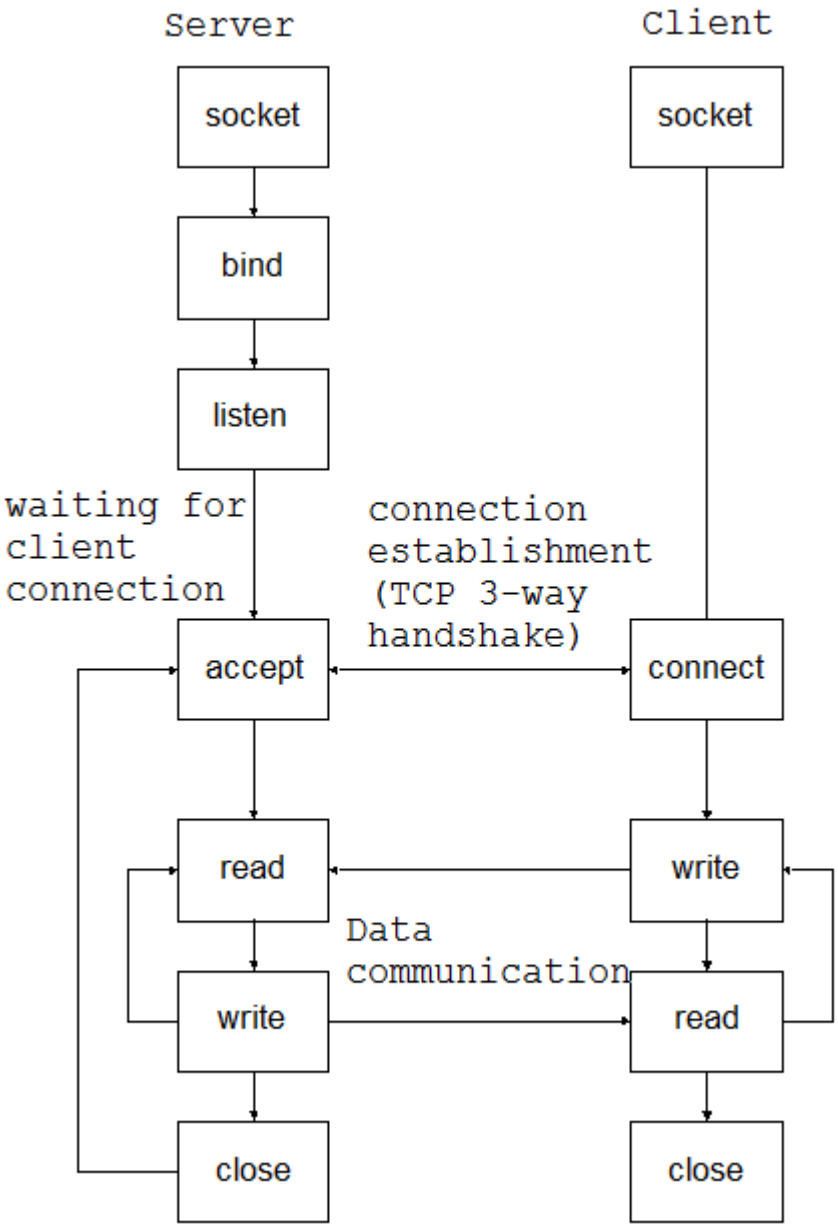

Figure 2 The flow of making a client/server communication in the case of a server dealing with one connection at a time.

`listen()`

- The `listen()` function is only used by a TCP server. It will convert an unconnected socket into a *passive socket*, meaning that the OS should accept incoming connection requests directed to this socket. At this moment, the clients can start connecting to the server.

- It takes in two arguments, the socket identifier and the *backlog*. It is the size of the buffer containing the requests for new connections waiting to be processed (set it to **10** for the experiments in this course).

`accept()`

- `accept()` is called by the TCP server to return the socket identifier of a completed connection, which is extracted from the queue mentioned in `listen()`. The identifier is generated by the operating system and is different from the listening socket identifier. This identifier will be used for the transaction of data. The structure of the internet socket address which contains the client's IP address and the port number will be returned by reference.
- It takes in three arguments: the socket identifier of the listening socket, the pointer to a structure of internet socket address, and a pointer to the length of the socket. The client's IP address and port number will be stored in the structure that the pointer points to, which will be accessible to the server.

`connect()`

- `connect()` is used by a TCP client to establish a connection with a TCP server. It will return –1 if the connection cannot be established, whether the server is down or the server buffer is full.
- It takes in three arguments: The socket identifier, the structure of internet socket address, which contains the server's IP address and port number, and the length of the structure.

`read()/write()`

- send/receive data between client and server.
- Three arguments are needed: The socket identifier, the pointer to the buffer where the data received/to be sent is stored, and the size of the buffer.

`close()`

- Close the connection and free the socket identifier.
- It takes only the socket identifier of the socket you want to close.

Procedure to create UDP sockets

Some of the key differences between UDP and TCP based socket programming will be discussed in class. You are also encouraged to refer to some of the online socket tutorials.

Concurrency using `select()`

You can learn more information by **`man select`** or by referring to the first book in references. We will also discuss this further in class.

References

[Ste90] R. Stevens, *UNIX Network Programming*, Prentice Hall, 1990.

[BFF96] T. Berners-Lee, R. Fielding and H. Frystyk, *Hypertext Transfer Protocol -- HTTP/1.0*, RFC 1945, May 1996.

For other references, see lab webpage.