Socket Application Program Interface

Recommended reading: This note is based on Chapters 20,21 of the text book.

Client-server model

- Application architecture

An application program running on a host is called a process. Interaction between processes within a host is realized by the inter-process communication defined by the operating system (OS) running on the host. Interaction between processes in different hosts is realized by exchanging messages over a network connecting the hosts. TCP/IP provides the functions for the message exchange. A fundamental interaction (communication) pattern between processes over a network is known as client-server model.

A client is a process which sends a request to a server for a service (initiates a communication). A server is a process which reacts to the request from a server to provide the requested service. In a TCP/IP internet, a client may have dynamic port number and IP address, and may be intermittently connected, while a server is usually always on duty, has a permanent port number of IP address. There is no interaction between clients.

- Extension of client-server model

Based on the client-server model, peer-to-peer (P2P) model is developed. In a P2P communication, each communication end has both a client process and a server process. A server process in a P2P communication may not be always on duty, may have dynamic port number and IP address, and may be intermittently connected.

Socket API

- Application Interface to Protocols

An operating system in a computer provides an interface between application programs and TCP/IP protocols. The interface is known as Application Program Interface (API). An application program uses TCP/IP via the API for communication. TCP/IP standards do not specify the form of the API (it may depend on operating systems) but suggest its functionality which includes: allocate local resources for communication, specify local and remote communication endpoints, initiate a connection (client side), send a datagram (client side), wait for an incoming connection (server side), send or receive data, determine when data arrives, generate urgent data, handle incoming urgent data, terminate a connection gracefully, handle termination from remote site, abort communication, handle error conditions or a connection abort, and release local resources when communication finishes.

- The Socket API

The socket API is the result of transporting TCP/IP software into UNIX operating system by a group at the University of California at Berkeley. The system is known as BSD UNIX (Berkeley Software Distribution UNIX). The socket API is now a de facto standard and forms the basis of the API for many operating systems.

The socket API was designed based on an approach which defines functions that support network communication in general, and uses parameters to make TCP/IP communication a special case. All TCP/IP protocols are represented as a single family (PF_INET) in the socket API.
The socket API adds a new abstraction, socket, for network communication. Like a file descriptor, a socket is identified by a socket descriptor which is an integer. To access a file, an application program calls open function to create a file descriptor that it uses for the access. A system function socket is provided by the operating system. An application program calls function socket to create a socket descriptor that it uses to specify an endpoint of a network connection. The same value cannot be used for both a socket descriptor and a file descriptor within the same process because the descriptors are kept in the same table for each process. A socket is created by a single system call but additional system calls are needed to specify the details of its exact use. When a socket is created, a data structure is allocated to the socket that includes the following fields: protocol family (the value is specified in the call of socket, e.g., PF_INET), type of service (the value is specified in the call of socket, e.g., SOC_STREAM), local IP address, remote IP address, local port number, remote port number (these fields will be filled by additional system calls).

- **Specify endpoint address**

  When an application program communicates with another application program via a network connection, a socket descriptor is used to specify an endpoint of the connection. TCP/IP protocols define an endpoint in a network connection by an IP address and a protocol port number. When a socket is created, it does not have information on the endpoints. An application must specify the address for one or both endpoints to use the socket. The address specification is protocol dependent. For TCP/IP protocols, a single address family AF_INET is used. In practice, an application program uses the following C struct to represent a TCP/IP (IPv4) communication endpoint.

```c
struct sockaddr_in {
    u_char sin_len; /* total length */
    u_short sin_family; /* type of address */
    u_short sin_port; /* protocol port number */
    struct in_addr sin_addr; /* IP address */
    char sin_zero[8]; /* unused (set to zero) */
};
struct in_addr{ /*32-bits IPv4 address, in network byte order*/
    in_addr_t s_addr;
};
```

- **Primary system calls**

  Socket calls can be classified into two groups. One is the group of primary calls which provide access to the functionality of the socket. The other is the group of utility functions (such as getting an IP address by a host name) which make programming easier. The primary calls include:

  - socket, create a socket.
  - connect, (client) request a connection to server.
  - send, sendto, send (write) data to connection.
  - recv, recvfrom, receive (read) data from connection.
  - shutdown, end sending or receiving, partial close.
  - close, close connection.
– **bind**, assign an endpoint address to a socket.
– **listen**, make a socket passive and define the maximum number of connections can be queued for the socket.
– **accept**, (server) accept a connection request from a client.

- **Client action (Client-server communication)**
  
  socket, create a socket.
  connect, connect to a server.
  send/recv, transfer data (may be repeated).
  shutdown, end transfer data (partial close).
  close, close connection.

- **Server action**
  
  socket, create a socket.
  bind, assign the socket on the server a local endpoint address.
  listen, make the socket on the server passive and define the maximum number of connections can be queued for the socket.
  accept, establish the connection with a client.
  send/recv, transfer data (may be repeated).
  shutdown, end transfer data.
  close, close connection.

- **Details on primary calls**

  The socket function

  ```
  # include <sys/types.h>
  # include <sys/socket.h>
  int socket(int family, int type, int protocol);
  ```

  is used to create a new socket. The call of the function returns a socket descriptor which is an integer. The parameter **family** specifies the family of the protocols that the application will use. Examples for family include PF_INET for IPv4, PF_INET6 for IPv6, etc. The parameter **type** specifies the type of data transfer. For TCP/IP protocol family, SOCK_STREAM for TCP, SOCK_DGRAM for UDP, etc. The parameter **protocol** specifies the particular protocol, for example, IPPROTO_TCP, IPPROTO_UDP, etc. The parameter protocol can be specified by value 0 if the parameter type has explicitly specified which protocol is used. The following is an example to create a new socket.

  ```
  int newsocket;
  newsocket = socket(PF_INET, SOCK_STREAM, 0);
  ```

  In the example, variable **newsocket** is used to keep the socket descriptor for the newly created IPv4 TCP socket. If there is an error in the socket creation process, a value −1 is returned. This socket has not been associated with any communication endpoint yet. The socket is an active one.

  The connect function
# include <sys/types.h>
# include <sys/socket.h>
int connect(int sd, struct sockaddr *serveraddress, int len);

is used by a client to establish an active connection to a remote server after a socket is created. The parameter \texttt{sd} is a socket descriptor which specifies the client’s socket. The parameter \texttt{serveraddress} specifies the IP address and port number of the server and is a pointer to a generic address structure (cast from the actual address structure \texttt{sockaddr} in used). The parameter \texttt{len} specifies the number of bytes in the \texttt{serveraddress}.

The \texttt{send} function

# include <sys/types.h>
# include <sys/socket.h>
int send(int sd, char *buffer, int len, int flag);

is used to send data through a TCP connection. The parameter \texttt{sd} is a socket descriptor which specifies the socket to which the data is sent. The parameter \texttt{buffer} is the address of the data to be sent. The parameter \texttt{len} specifies the length of data. The parameter \texttt{flag} is optional and takes value 0 if not used.

The \texttt{recv} function

# include <sys/types.h>
# include <sys/socket.h>
int recv(int sd, char *buffer, int len, int flag);

is used to receive data from a TCP connection. The parameter \texttt{sd} is a socket descriptor specifies the socket from which the data is received. The parameter \texttt{buffer} is the address to hold the received data. The parameter \texttt{len} specifies the length of data. The parameter \texttt{flag} is optional and takes value 0 if not used.

Similar functions used for UDP are \texttt{sendto} and \texttt{recvfrom}. There are a number of other functions, such as \texttt{read} and \texttt{write}, performing similar works.

The \texttt{shutdown} function

# include <sys/types.h>
# include <sys/socket.h>
int shutdown(int sd, int dir);

is used to close a TCP connection in a direction specified by parameter \texttt{dir}. For \texttt{dir}=0, reading is ended, \texttt{dir}=1, writing is ended, and \texttt{dir}=2, both reading and writing are ended.

The \texttt{close} function

# include <sys/types.h>
# include <sys/socket.h>
int close(int sd);
is used to finish using the socket specified by \texttt{sd}. The default results of the function are marking the socket closed, sending a FIN to start a three way terminating handshake for the TCP connection, preventing further use of the socket, and allowing any previously queued data sent before closing the connection.

In the client-server communication model, functions \texttt{socket}, \texttt{connect}, \texttt{send}, \texttt{recv}, \texttt{close} are fundamental for application programs at client side. For application programs at the server side, in addition to the above functions, more system calls are needed.

The \texttt{bind} function

\begin{verbatim}
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sd, struct sockaddr *localaddress, int len);
\end{verbatim}

is used to associate the socket specified by the \texttt{sd} with a local (the same host that the process performs the binding) endpoint address. Usually, a server uses \texttt{bind} to specify the well-known port at which it will await connections. For TCP/IP protocols, the endpoint address includes an IP address and a protocol port number.

The \texttt{listen} function

\begin{verbatim}
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sd, int size);
\end{verbatim}

is used to turn socket specified by the \texttt{sd passive} and make it ready to accept incoming connections. The parameter \texttt{size} specifies the maximum number of pending connections for this socket.

The \texttt{accept} function

\begin{verbatim}
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sd, struct sockaddr *clientaddress, int len);
\end{verbatim}

is used to extract the next incoming connection request. The parameter \texttt{sd} is a socket descriptor which specifies a passive socket (after call of \texttt{socket}, \texttt{bind}, and \texttt{listen} by a server). If the connection request has been successfully accepted, function \texttt{accept} creates a new socket descriptor for each new connection request and returns the descriptor for the socket, otherwise it returns \texttt{-1}. The server uses the new socket for the new connection and the original socket (specified by \texttt{sd}) to accept additional connection requests. After a server finishes using the new socket, it closes the socket. The parameter \texttt{clientaddress} returns the connection endpoint address (IP address and port number of the host running the client) whose request has been accepted. The parameter \texttt{len} returns the number of bytes in \texttt{clientaddress}. If there is no connection request, the function waits for an incoming request, and when a request is received, it creates a socket for it.

• Utility functions

In an application program using socket for communication, byte order conversion and IP address translation are often needed. Such works can be realized by utility functions.
• **Byte order conversion**

TCP/IP protocols specify a standard representation, known as *network byte order*, for binary integers used in protocol headers. Some socket functions require parameters stored in the network byte order, for example, the protocol port in `sockaddr` in `struct`. On the other hand, the order of the bytes in a multiple byte binary integer stored in a computer (host) is system dependent. There are two different orders: *little endian* (low order byte first) and *big endian* (high order byte first). For example, the host ordering in Sparc systems is big endian and that in Intel systems is little endian. The network byte order is big endian. The following functions provide the conversions between the host ordering and network ordering.

```c
unit_16t htons(unit_16t v);
```

Convert host byte order to network byte order for short integer (16 bits).

```c
unit_32t htonl(unit_32t v);
```

Convert host byte order to network byte order for long integer (32 bits).

```c
unit_16t ntohs(unit_16t v);
```

Convert network byte order to host byte order for short integer (16 bits).

```c
unit_32t ntohl(unit_32t v);
```

Convert network byte order to host byte order for long integer (32 bits).

• **IP address translation**

An application program needs to specify the endpoint addresses for using a socket (e.g., a client specifies a server) by `sockaddr` in `struct`. This may require converting a dotted decimal IP address (for IPv4) or a domain name in text form into a 32-bit binary IP address.

The function `inet_pton` (presentation to network byte order)

```c
#include <netinet/in.h>
int inet_pton(int family, const char *strptr, void *addrptr);
```

is used to convert a dotted decimal (or hexadecimal, for IPv6) IP address into a binary IP address in the network byte order. The parameter `family` is either AF_INET or AF_INET6. The parameter `strptr` is the dotted decimal (or hexadecimal) IP address. The parameter `addrptr` is the returned binary IP address. The function returns 1 if the conversion is successful, 0 if presentation error, -1 error.

The function `inet_ntop` (network order to presentation)

```c
#include <netinet/in.h>
int inet_ntop(int family, const char *addrptr, char *strptr, size_t len);
```
is used to convert a binary IP address in network byte order into a dotted decimal (or hexadecimal, for IPv6) IP address. The parameter family is either AF_INET or AF_INET6. The parameter addrptr is the binary IP address. The parameter strptr is the returned dotted decimal (or hexadecimal) IP address. The function returns 1 if the conversion is successful, 0 if presentation error, -1 error.

The function `gethostbyname`

```
#include <netdb.h>
struct hostent *gethostbyname(const char *hostname);
```

is used to convert a domain name into a binary IP address. The function returns the address of a `hostent` structure which is defined in `netdb.h` and contains the binary IP address (and other information) of the host specified by parameter `hostname`.

```
struct hostent{
    char    *h_name;          /* official host name */
    char    **h_aliases;      /* pointer to array of pointers to alias names */
    int     h_addrtype;      /* host address type (AF_INET or AF_INET6) */
    int     h_length;        /* address length */
    char    **h_addr_list;   /* pointer to array of pointers to IP addresses */
}
#define h_addr h_addr_list[0]
```

An example

```
struct hostent *hptr;
char *hostname="dogwood.css.sfu.ca";
if (hptr=gethostbyname(hostname)){
    /* binary IP address is in hptr->h_addr now */}
else {/* error */}
```

• Interactive/concurrent server

An interactive server processes one connection request at a time (only one socket is opened). When the processing on the connection is completed, the socket is closed and the next connection can be accepted. A concurrent server can handle multiple connections simultaneously. In general, it is more difficult to write a program for concurrent servers but concurrent servers have better performance. There are different approaches to realize concurrent servers. A simple one is using the function `fork`.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

To handle concurrent connections, after a server (called parent server) accepts a new connection, the parent server calls the function `fork` to make a copy of itself. The copy is called a child server. After the successful creation of the child server, the parent server closes the socket for the newly
accepted connection and keeps waiting for other connections. The child server which has access
to both the socket for accepting connections and the socket for the new connection first closes
the socket for accepting connections and handles the communication via the socket for the new
connection. When the communication is completed, the child server closes the socket for the new
connection and exits. An example:

```c
listen(acceptsd, size);
while (1){
    newsd=accept(acceptsd, ...);
    if (pid=fork())==0){/* child server’s action */
        close(acceptsd);      /* close the socket for accepting */
        process-sth(newsd);   /* do sth using the connection specified by newsd*/
        close(newsd);         /* close the socket for the new connection */
        exit(0);
    }
    close(newsd); /* parent server close the socket for new connection */
}
```

- **Peer-to-peer communication**

  The communication is symmetric (peer-to-peer). Main steps in peer-to-peer communication pro-
  gram includes:
  - `socket`, create a socket
  - `bind`, associate the socket an endpoint address (optional for initiator)
  - `sendto/recvfrom`, transfer data (may be repeated)
  - `shutdown`, end sending/receiving data
  - `close`, close connection

  The socket in the above model is not connected and is used for connectionless protocol like UDP.
  An application program can use a UDP socket in either unconnected mode or connected mode.
  For connected mode, the program calls function `connect` to specify a remote endpoint address.
  In the connected mode, instead of `sendto/recvfrom`, `send/recv` are used for transfer data.