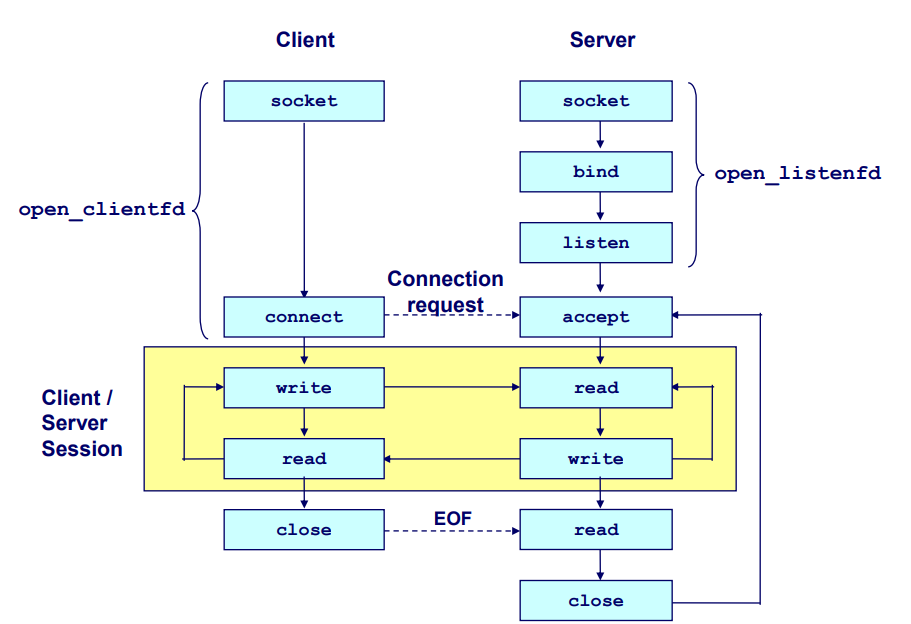
## Socket Programing

**Objective:** Develop a Chat application using Socket Inter process commination.

**Block diagram:**

**Overview:**



**Required skills:**

Basic knowledge on posix sockets/C++

**Compiler/Tools:**

G++ Compiler, vim editor or any C++ IDE.

**Protocols:**

TCP/IP.

**Explain Code flow:**

All POSIX socket programming starts with the creation of a socket file descriptor using the socket API, which takes the following form:

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

**Domain filed:** Defines the address type used when creating the socket. In most cases, this would be AF\_INET for IPv4 or AF\_INET6 for IPv6. In the case of our examples in this chapter, we will use AF\_INET.

**Type field:** Usually takes on SOCK\_STREAM for a TCP connection or SOCK\_DGRAM for a UDP connection.

**Protocol field:** in this API will be set to 0 in this application, telling the API to use the default protocol for whichever socket type is specified.

Upon completion of this API, a socket file descriptor is returned, which will be needed by the remaining POSIX APIs. If this API fails, -1 is returned, and **errno** is set to an appropriate error code. It should be noted that errno is not thread-safe and its use should be handled with care. A great way to handle these types of errors is to immediately convert the errno into a C++ exception, which can be done using the following:

if ((m\_fd = ::socket(AF\_INET, SOCK\_STREAM, 0) == -1) {  
 throw std::runtime\_error(strerror(errno));  
}

# Close() API:

Finally, when the socket is no longer needed, it should be closed like any other file descriptor using the POSIX close() function. It should be noted that most POSIX operating systems will automatically close sockets that are still open when the application closes.

# The bind() and connect() APIs

Once a socket file descriptor is created, the socket must be bound, or connected, depending on whether the socket is creating the connection (the server), or is connecting to an existing bound socket (client). When communicating via TCP or UDP, binding a socket dedicates a port for the socket. Ports 0-1024 are reserved for specific services and are often managed by the operating system (requiring special privileges to bind). The remaining ports are user-defined and often may be bound without privileges. Determining which port to use is dependent on the implementation. Some ports are predetermined for a specific application, or the application can ask the operating system for an open port to use, which has the added complication of communicating this newly-allocated port to potential client applications.

**The bind() API takes the following form:**

int bind (int socket, const struct sockaddr \*address, socklen\_t address\_len);

The socket integer parameter is the socket file descriptor that was previously provided by the socket() API. The address parameter tells the operating system which port to bind to, and which IP address to accept incoming connections from, usually INADDR\_ANY which tells the operating system that an incoming connection may be accepted from any IP address. Finally, the address\_len parameter tells the API what the total size of the address structure is.

The total size (in bytes) is needed for the address structure because different structures are supported depending on the socket type you're using. For example, an IPv6 socket has a larger IP address compared to an IPv4 socket. In this chapter, we will discuss IPv4, which uses the sockaddr\_in{} structure, which defines the following fields:

* sin\_family: This is identical to the socket domain, which, in the case of IPv4, is AF\_INET.
* sin\_port: This defines the port to bind to, which must be converted into network byte order using htons().
* sin\_address: This defines the IP address to accept incoming connections from, which must also be converted into network byte order using htonl(). Often, this is set to htonl(INADDR\_ANY), indicating connections are accepted from any IP address.

While servers use bind() to dedicate a port for the socket, clients use connect() to connect to an already-bound port. The connect() API has the following form:

int connect(int socket, const struct sockaddr \*address, socklen\_t address\_len);

It should be noted that the parameters for connect() are identical to bind(). Like bind(), you must provide the file descriptor returned by the call to socket(), and like bind(), you must provide, in the case of IPv4, a pointer to a sockaddr\_in{} structure as well as the size of the sockaddr\_in{} structure.

When filling out the sockaddr\_in{} structure, you would use the following:

* sin\_family: This is identical to the socket domain, which, in the case of IPv4, is AF\_INET.
* sin\_port: This defines the port to connect to, which must be converted into network byte order using htons().
* sin\_address: This defines the IP address to connect to, which must also be converted into network byte order using htonl(). For loopback connections, this would be set to htonl(INADDR\_LOOPBACK).

Finally, both bind() and connect() return 0 on success or -1 on failure, setting errno in the event of an error.

# The listen() and accept() APIs

For TCP servers, two additional APIs exist that provide the server with a means to listen for and accept incoming TCP connections listen() and accept().

The listen() API has the following form:

int listen(int socket, int backlog)

The socket parameter is the file descriptor returned by the socket() API, and the backlog parameter limits the total number of outstanding connections that may be made. In the examples in this chapter, we will use a backlog of 0, which tells the API to use an implementation-specific value for the backlog.

If listen() succeeds, 0 is returned, otherwise -1 is returned and errno is set to the appropriate error code

Once your application is set up to listen for incoming connections, the accept() API may be used to accept a connection once it is ready.

The accept() API has the following form:

int accept(int socket, struct sockaddr \*address, socklen\_t \*address\_len);

Like the other APIs, the socket parameter is the file descriptor returned by the socket() API and the address, and the address\_len parameter returns information about the connection. nullptr may also be provided for both the address and address\_len if the connection information is not needed. Upon successful completion of the accept() API, a socket file descriptor for the client connection is returned, which may be used to send and receive data to and from the client.

If accept fails to execute, instead of a valid socket file descriptor being returned, -1 is returned, and errno is set appropriately.

It should be noted that both listen() and accept() are only needed for TCP connections. With a TCP connection, the server creates two or more socket descriptors; the first one is used to bind to a port and listen for connections, while the second one is the socket file descriptor for the client, which is used to send and receive data. UDP, on the other hand, is a connectionless protocol and thus the same socket that is used to bind to a port is also used to send and receive data with the client.

# The send(), recv(), sendto(), and recvfrom() APIs

To send information to a server or client after opening a socket, POSIX provides the send() and sendto() APIs. The send() API has the following form:

ssize\_t send(int socket, const void \*buffer, size\_t length, int flags);

The first parameter is the socket file descriptor for the server or client you wish to send data to. It should be noted that the socket must be connected to a specific client or server to work (such as communicating back to a server, or to a client opened using TCP). The buffer parameter points to the buffer you wish to send, length defines the length of the buffer you wish to send, and flags provides various different settings for how you wish to send the buffer, which in most cases is just set to 0. It should also be noted that when flags is set to 0, there is typically no difference between the write() function and the send() function, and both may be used.

If a server is attempting to communicate with a client using UDP, the server won't know who or how to send information to the client as the server binds to a specific port, not to a specific client. Likewise, if a client using UDP doesn't connect to a specific server, it will not know who or how to send information to the server. For this reason, POSIX provides sendto(), which adds the sockaddr{}structure to define who and how you wish to send the buffer. sendto() has the following form:

ssize\_t sendto(int socket, const void \*buffer, size\_t length, int flags, const struct sockaddr \*dest\_addr, socklen\_t dest\_len);

The only difference between send() and sendto() is that sendto() also provides the destination address and len parameters, which provide the user with a way to define who the buffer is sent to

To receive data from a client or server, POSIX provides the recv() API, which has the following form:

ssize\_t recv(int socket, void \*buffer, size\_t length, int flags);

The recv() API takes the same parameters as the send() API, with the difference being that the buffer will be written to (which is why it's not labeled const) when data is received, and the length field describes the total size of the buffer and not the total number of bytes received.

Likewise, POSIX provides a recvfrom() API, which is similar to the sendto() API and has the following form:

Ssize\_t recvfrom (int socket, void \*restrict buffer, size\_t length, int flags, struct sockaddr \*restrict address, socklen\_t \*restrict address\_len);

Both the send() and sendto() functions return the total number of bytes that were sent, while the recv() and recvfrom() functions return the total number of bytes received. All of these functions return -1and set errno to an appropriate value in the event of an error.

Please find the attached Src files.



**Testing procedure**:

**Real-Time Uses**