Approach 1 : Multithreaded

- A new thread is created for each client. Main thread accepts new connections using the listening file descriptor.
- There are five types of messages - JOIN, LIST, UMSG, BMSG and LEAV.
- For each client, we have a corresponding struct of type Client to store the name of the client and the socket corresponding to it, and the corresponding thread ID.
- A global data structure a linked list of such client entries is maintained, and atomic access to it is provided using a mutex variable.
- Each thread executes a function called serviceRequest, whose job is to read the incoming message, classify it as one of the above mentioned 5 types, process it accordingly, and update / use the global data structure (if required) to send back the reply.

Approach 2 : Event driven

- All sockets are set to be non blocking, and I/O notifications are received using the epoll API.
- A message queue is used as FIFO queue to queue the events.
- An epoll instance epfd is initially created to receive I/O notifications. Edge triggering is used (EPOLLET).
- Then, we create two threads :
    - First thread, repeatedly calls epoll_wait, receives the ready list of events, and pushes it into the message queue.
    - Second thread, repeatedly reads from the message queue, and services each request using a global data structure for clients which is similar to the one described in approach 1. If the event is on the listening fd, it accepts the new connection, and adds connfd to interest list of epfd instance using epoll_ctl().

Client :

The approach being followed for the client program is :

Client takes in 4 command line arguments : port number of server, N, M, T, where N is the number of concurrent connections, M is the number of messages exchanged, and T is the total number of connections. Now, assuming N to be 10, M as 5, and T as 25, then first, 10 processes will make connections to the server, and each will exchange 5 messages with the server. Whenever one of these processes finishes sending these 5 messages, it will terminate, and another process will then initiate a connection with the server, to ensure that at any moment, there are 10 concurrent connections with the server. This will go on until the total processes created reaches the value of T, which is 25.

Performance Comparison:

We have measured the multithreaded server w.r.t. the client discussed above based on following two parameters:

- **Latency** measures the end-to-end time processing time. In a messaging environment, we determine latency by measuring the time between sending a request and receiving the response. Latency is measured from the client machine and includes the network overhead as well.
- **Throughput** measures the amount of messages that a server processes during a specific time interval (e.g. per second). Throughput is calculated by measuring the time taken to processes a set of messages and then using the following equation.

*Throughput = number of completed requests * size_of_a_request / time to complete the requests*

We got the following results:

Approach 1:

Concurrent Clients	Average Latency (Clock Ticks)
10	713.3
20	222.575
40	454.675
60	513.33
80	364.02

Concurrent Clients	Throughput (Bytes per clock ticks)
10	0.432276657061
20	0.304375396322
40	0.471651763781
60	0.499653018737
80	0.379446640316

Approach 2:

Concurrent Clients	Average Latency (Clock Ticks)
10	558.05
20	205.875
40	176.2125
60	141.125
80	120.816666667