1. ***Language*** - C++
2. ***Message Format -***

Each message sent has the following format -

**<Tag>:<GSeq>:<LSeq>:<Content>**

* Tag - Tag field in the message is used to identify the type and the subtype of the message. Further, it identifies if a message if a data message or an acknowledgement. The Tag field if of 3 bytes in length and had the following format -

**<type><subtype><\_/A/N>**

Each of the above fields is of 1 byte length. The type field is used to identify the type of message or to broadly classify the message as chat message, notification message, Sequence request message, Election request message or a HeartBeat message. Each of the aforementioned types have sub types and the subtype field is used to identify the subtypes. The third field is used to identify if the message is a content message or an Acknowledgement message in which an \_ identifies a data message, A identifies an acknowledgement and an N identifies a negative acknowledgement. The tables below summarize the various Tags that will be used in the project.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No.** | **Type/Condition** | **<type>** | **<subtype>** | **<\_/A/N>** |
|  | **Notification Type -** |  |  |  |
| 1 | Join – request | N | 0 | \_ |
| 2 | Join – Success Response | N | 0 | A |
| 3 | Join – Fail/Error Response | N | 0 | N |
| 4 | Leave – before leaving/ crash detection | N | 1 | \_ |
| 5 | Leave – Acknowledgement | N | 1 | A |
| 6 | Announcement - message | N | 2 | \_ |
| 7 | Announcement - Ack | N | 2 | A |
|  | **Chat Type -** |  |  |  |
| 8 | Chat – chat message | C | 0 | \_ |
| 9 | Chat - Ack | C | 0 | A |
|  | **Sequence Type -** |  |  |  |
| 10 | Sequence - Request for new sequence | S | 0 | \_ |
| 11 | Sequence - Request Ack | S | 0 | A |
| 12 | Sequence - Response | S | 1 | \_ |
| 13 | Sequence - Response Ack | S | 1 | A |
|  | **Election Type -** |  |  |  |
| 14 | Election – Broadcast election start | E | 0 | \_ |
| 15 | Election - Broadcast Ack | E | 0 | A |
| 16 | Election – Request | E | 1 | \_ |
| 17 | Election - Response/Ack | E | 1 | A |
| 18 | Election – Leader Broadcast | E | 2 | \_ |
| 19 | Election – Leader Broadcast Ack | E | 2 | A |
|  | **Heartbeat Type -** |  |  |  |
| 20 | HB | H | 0 | \_ |
| 21 | HB – Ack | H | 0 | A |

Further, for most of the above cases, further information is stored in the content part of the message which are explained in the content section below.

* GSeq - This field contains the global sequence of a message which is used to enforce total ordering in the system. From the above table the message types Notification and Chat use the global sequence number. For other message types this field is is either absent or contains a default value of 0.
* LSeq - This field contains the local sequence number which reflects the number of messages sent from the sender to the receiver client. This number is unique based on the sender-receiver pair and the message. This field is used by all the message types described above.
* Content - This field contains the content of the message. For each type and subtype of the message, the content field will contain different types of information. The content is described by the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Type/Condition** | **<tag>** | **<Content>** |
|  | **Notification Type -** |  |  |
| 1 | Join – request | N0\_ | username |
| 2 | Join – Success Response | N0A | Case 1 - expected count of participant list  Case 2 - ip:port:seq:username  Here Case 1 is used to accept a join request and case 2 is used to send over the participant list to the new joinee |
| 3 | Join – Fail/Error Response | N0N | Error code |
| 4 | Leave – before leaving/ crash detection | N1\_ | Error code to indicate reason of leaving |
| 5 | Leave – Acknowledgement | N1A | - |
| 6 | Announcement - message | N2\_ | Message |
| 7 | Announcement - Ack | N2A | - |
|  | **Chat Type -** |  |  |
| 8 | Chat – chat message | C0\_ | Message |
| 9 | Chat - Ack | C0A | - |
|  | **Sequence Type -** |  |  |
| 10 | Sequence - Request for new sequence | S0\_ | MessageID |
| 11 | Sequence - Request Ack | S0A | MessageID |
| 12 | Sequence - Response | S1\_ | MessageID:SequenceNum |
| 13 | Sequence - Response Ack | S1A | MessageID:SequenceNum |
|  | **Election Type -** |  |  |
| 14 | Election – Broadcast election start | E0\_ | ElectionID |
| 15 | Election - Broadcast Ack | E0A | ElectionID |
| 16 | Election – Request | E1\_ | ElectionID |
| 17 | Election - Response/Ack | E1A | ElectionID |
| 18 | Election – Leader Broadcast | E2\_ | ElectionID |
| 19 | Election – Leader Broadcast Ack | E2A | ElectionID |
|  | **Heartbeat Type -** |  |  |
| 20 | HB | H0\_ | - |
| 21 | HB – Ack | H0A | - |

1. **Client Code Structure -**

The clientis a multi threaded program with each thread performing a different operation. The following threads will be used in the client -

* **Sequencer -** This thread assigns sequence number to chat messages. A sequence number is allocated to a chat message depending on two fields: global sequence number and client sequence number.

Each client maintains the sequence number of requests it has previously sent to the sequencer. Whenever the client raises a request to the sequencer, it includes this sequence number (client\_seq) in its request. The sequencer maintains a field called last\_client\_seq. This field is updated whenever the sequencer assigns a sequence number to a new chat message and is equal to the client\_seq of that request.

If the sequencer receives an out of order request, ie, with client\_seq not equal to last\_client\_seq +1 , it puts that request in a hold back queue. The requests in the queue are assigned a sequence number only when client\_seq of that request is one greater than last\_client\_seq for that client. This approach helps is correct numbering of messages.

* **Heart Beat -** The heartbeat thread is periodic thread that is responsible for keeping track of the active members of the group and to detect crashes. It executes in two ways based on weather the client is a sequencer or a regular participant.

For the sequencer the heartbeat thread sends a heartbeat message with tag H0\_ to all the participants and goes to sleep for a short duration. In the mean time, the network thread collects heartbeat acknowledgements and updates them in the heartBeatMap, described below. When the thread resumes, it checks the heartBeatMap to see if all the processes have responded or not. The processes that haven't acknowledged the heart beat are assumed to be dead and the sequencer then goes on to inform all the active participants. Then the sequencer resets the heartBeatMap, sends a new heartbeat message and the process repeats itself.

For a regular participant the heartbeat thread checks if it has received a heartbeat from the sequencer or not. The network thread is responsible for receiving the heartbeat, acknowledging it and (via shared variables) informing the heartbeat thread that a heartbeat was received. The heartbeat thread sleeps for a short duration and when it resumes, it simply checks if a heartbeat was received or not.If it wasn't received then the sequencer is assumed to be dead and the heartbeat thread is responsible for starting a leader election by triggering the election thread.

* **Election -** The election thread is used in case of leader election to elect a new sequencer if the old sequencer crashes. The election thread is triggered by either the heartbeat thread or the network thread. If the heartbeat thread realizes that the sequencer has crashed, it will start a new leader election by triggering the election thread. In this case the election thread will then send an election broadcast to all clients, informing them of the impending election so that they can start queuing their user messages till the election has concluded. Then it sends an election request to all the clients that have a higher IP:PORT than itself to start an election based on the **bully algorithm**. When the election concludes, the new leader broadcasts its identity to every client in the group.

However, the Election thread can also be triggered by the network thread, when it receives an election request from any other client. In such a case the election thread is responsible for responding to the election request and then proceeding with the same process as above to conduct leader election based on the bully algorithm.

* **User Thread -** This thread is used to receive message from the console from the user and send it to the appropriate client. This thread is mainly used to service the chat messages that the user inputs while the program is being executed.
* **Network Thread -** This thread is used to receive messages from the network on the socket and main port of the client. The message is first received and verified. Then the message is broken into four fields i.e. Tag, GSeq, LSeq and Content. Based on the Tag, the client responds to the message.

1. **Data Structures -**

* **Participant -** The participant data structure is used to hold all the information about a participant. It has the following format -

struct participant

{

struct sockaddr\_in address;

int seqNumber;

string username;

};

The participant structure is used in the participant list and to hold the details about the sequencer.

* **Participant List -**  Each client maintains a participant list which contains all the participants in the form of a hash map with IP:PORT as the unique key and struct participant as the value.

std::map <string, struct participant \* > participantList;

* **Heart Beat Map -** HeartBeat Map is used by the sequencer to keep track of the clients that are alive, as explained in the heartbeat thread section. It is a hash map that has IP:PORT as a key and a bool value to indicate if the process is alive or not. It is defined as following -

std::map <string, bool > heartBeatMap;

* **Sequencer-** 
  + Global sequence number: It is a simple integer which is incremented by one every time a new sequence number is assigned.
* Hold back queue: The sequencer uses hash map with IP:PORT of the client which requested a sequence number as the unique key and a structure consisting of last\_client\_seq and a vector of all the client\_seq numbers that were received out of order.

Following is the structure of the **hold\_back\_queue**:

std::**map**<***string***, ***struct*** LastSeen> hold\_back\_queue;

where, LastSeen has the following structure:

***struct*** LastSeen

{

***int*** last\_client\_seq;

std::**vector**<***int***> client\_seq\_nos;

};

* **Hold Back Queue-**  Each member in the chat room maintains its own list of chat messages to be displayed. The hold back queue is used to maintain total ordering across all the members. It is implemented using **map** data structure. The key is unique sequence number of the chat message and value is the actual message to be displayed.

Following is the structure of the **hold\_back\_queue**:

std::**map**<***int***, ***string***> hold\_back\_queue;