CIS 505
Project 3 Milestone 3

Group 34
Aayushi Dwivedi Aniruddha Rajshekar Ankit Mishra

Group Members

Name	PennKey
Aayushi Dwivedi	**aayushi**
Aniruddha Rajshekar	**arajs**
Ankit Mishra	**mankit**

Assumptions

We have done aggressive testing for upto 4 members per group. So we assume that the chat system willl work fine for group sizes in the range of 1-5.

Aayushi Dwivedi Aniruddha Rajshekar Ankit Mishra

Language

We used C++ as our programming language. C++ provides more options when it comes to selection of data structures than C does. We exploited this fact to develop a more usable code.

Code Structure

The client is a multi threaded program with each thread performing a different operation. Following are the most important functions and threads used by the chat system:

1. **Sequencer :** This function 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.
2. **Heart Beat Thread:** 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.
3. **Election Thread:** 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.
4. **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.
5. **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.

Protocols

The chat system runs over UDP, hence events like lost, duplicate or re-ordered messages keep happening during a chat session. In order to handle such events, we came up with certain protocols. We cover each of them in brief in the following sections.

1. Message Reliability Protocol

- Handling Lost Messages or Message Reliability: Sometimes messages can get let lost or dropped during transmission. In order to ensure a reliable transmission, we use several buffers to keep track of different types of messages being transmitted over the network.

Message reliability at sequencer's end: After assigning a global sequence number to each sequence request, the sequencer multicasts the sequenced messages. The sequencer maintains a buffer called txBuffer (refer data structures section for exact details) to keep track of acknowledgements received for each multicast message and a thread is used to check this buffer after a fixed amount of time. For the first two thread cycles, all the members from whom an acknowledgment has not been received the sequencer retransmits the message. If some member fails to acknowledge the retransmitted messages even after the third retransmission, that member is removed from the buffer.
Message reliability at client's end: All clients use a seqBuffer (refer data structures section for exact details) to keep track of all out going sequence requests. If the sequencer fails to respond with a sequence number for that request, the client re-sends the request to the sequencer for sequencing. This ensures that all the messages on the clients side get sequenced.
Hence, the above two procedure follows *at least once* semantics for fault-tolerance.
In addition to this each client maintains a buffer called holdBackQ(refer data structures section for exact details) to store all the sequenced messages received from the sequencer. This queue is used to display messages in total order. The queue is sorted based on the global sequence number assigned to each message and the messages are displayed in exact order. Out of order sequence numbers are held back in the queue until the correct ordered message arrives.

- Handling Duplicate Messages : Message duplication can occur when the sequencer or a client retransmits a message. To avoid problem of duplication, we chose maps as our data structure. Each map uses a unique key for hashing. This helps in avoiding duplicated.

- Handling Reordered Messages: Total ordering is maintained by using a centralized sequencer. The sequencer maintains a map called hold_back_queue (refer data structures section for exact details). All out of order requests are pushed it into the inner map, using local sequence number of the request as key and the content as value. Once the in order request arrives, the sequencer assigns sequence number to all those held back requests which meet the following criteria: local sequence of the request = last_client_seq -1. This technique helps in ordering the messages.

2.Election Protocol

The person who initially starts the chat is the default leader. Being a leader requires that member of the chat room to carry out following tasks:

- Keep track of all the members and information related to them;
- Notify all the members of any updates in the group;
- Sequence all chat and notification messages;
- Carry out the heart beat;

All in all the leader is the backbone of the entire chat system. Hence, if for some reason the leader expires, a new leader must be elected immediately to prevent total break down the system. We achieve this by implementing the bully algorithm for leader election.

The election thread is used for leader election to elect a new leader if the old leader 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.

3.Sequencing Protocol

Our system was designed to implement total ordering of messages, implying all members in the chat room see the messages in the exact same order. In order to do so, we use a **centralized sequencer**. This sequencer is used to assign sequence numbers to all the chat messages that are to be delivered to all the members in the chat room. In addition to normal chat messages, it sequences notifications as well.

Each member in the chat room assigns a unique local sequence number to each of its messages. The member then requests the Sequencer to assign a global sequence number to its chat message. It does so by generating payload with the following format:

S0_:0:LSeq:IP_PORT_Key:Chat

where,

- S0_ is the payload identifier
- 0 is the initial global sequence number assigned to the unsequenced chat message
- LSeq is the local sequence number of the chat message given by the source member
- IP_PORT_Key is the unique identifying key of the source member
- Chat contains the actual content of the chat messages

After identifying it as a sequencing request, the leader calls the sequencer which carries out rest of the task. The algorithm for sequencing is as below:

- Check whether this is the first sequencing request from this key
    - if yes:
        - increment the global sequence number by 1, assign it to this request
        - add this key to sequencers hold back queue map and set last_client_seq corresponding to this key equal to the local sequence number of this message
        - Call sendSequenced()
    - if no:
        - check the inner map, the data structure holding all the out of order sequence requests and content corresponding to it.
        - If this map is empty and the last_client_seq for this key is exactly 1 less than this request's local sequence, then it implies all the requests from this key have been

sequenced. Increment the global sequence number by 1 and assign it to this request. Update the last_client_seq. Call sendSequenced().

- If the above is false, it implies that the current request has been received out of order.
    - Push it into the inner map, using local sequence number of the request as key and the content as value.
    - Wait until the next in order request arrives. Once the in order request arrives, assign sequence number to all those held back requests which meet the following criteria: local sequence of the request = last_client_seq -1; and call sendSequenced().

sendSequenced(): This function is used to broadcast the sequenced messages to all members in the chat room. sendSequenced() is responsible for differentiating between a normal chat message and a notification message. If the request is for a normal chat message, sendSequenced() simple multicasts it to all members in the participant list, which in turn put the message into their holdBackQ before displaying it. Otherwise, it identifies it as a notification message and forces all the members to display the notification message without pushing it into their holdBackQ. Hence, all requests are prioritized.

4.Heart Beat Protocol

We use the heart beat protocol to keep track of active members. It is essential for two purposes: first, to help the leader know how many members are active in the chat room; second, to help the members in the chat room detect the demise of the leader. To implement this protocol, we use a **heartbeat thread**.

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.

Data Structures

We mostly used maps and structures to create more complex data structures. This choice of data structures helped us to add several clients at the same time, gave us the freedom and flexibility to try different mechanisms. Use of maps and pointers gave us greater control over the code which in turn helped in debugging the code simpler.

In the following paragraphs we discuss in brief about some of the important data structures used in the implementation.

- **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;
  ```

- **Message :** Information regarding each chat message is stored in a special structure with the following format:

  ```
  struct message
  {
  string content;
  string senderKey;
  int localSeq;
  int globalSeq;
  int ackCount;};
  ```

- **Heart Beat Map :** HeartBeat Map is used by the leader 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:**
  Following are the data structures used in the implementation of the sequencer:

- **generatedGlobalSeq**: 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 map: the map is used to hold all the client_seq numbers that were received out of order and the corresponding chat message to that sequence number.

  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;
          map<int, string> client_msgs;
  };
  ```

- **Reliability:** To implement reliable delivery of messages, we use following data structures:
  - **holdBackQ:** 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 `struct message` defined above.
    Following is the structure of the `holdBackQ:`:
    ```
    std::map <int, struct message * > holdBackQ;
    ```

  - **seqBuffer**: It is a map to hold unsequenced messages with key as the local sequence number for that sequence request and value as `struct message` defined above. Whenever a member of the chat room the sends a sequence request to the sequencer, it pushes the request to the seqBuffer. If for some reason the sequencer fails to receive a sequence request, seqBuffer is used to retransmit a new sequence request.
    Following is the structure of txBuffer:
    ```
    std::map <int, struct message * > seqBuffer;
    ```

  - **txBuffer**: It is a map to hold sequenced messages with key as the global sequence number for that msessage and value as `struct txmessage` given below. Whenever a member of the chat room misses the broadcast sequenced message sent by the sequencer, txBuffer is used to retransmit that message.

    Following is the structure of txBuffer:
    ```
    std::map <int, struct txMessage * > txBuffer;
    ```

where,

```
struct txMessage
{
	string content;
	string senderKey;
	int localSeq;
	int globalSeq;
	int ackCount;
	std::set <string> ackList;
};
```

Difference between `struct message` and `struct txMessage` is that of an additional structure:
`std::set <string> ackList;`
txBuffer uses this set to keep track of acknowledgements sent from members after receiving a sequenced chat message.

Transmission Message Format and Payloads

This section describes in detail the design and usage of messages. We have used tables for easy reference.

Transmission Message Format

All transmission messages follow the same convention. Fixing the message format helped us to differentiate between different types of messages being communicated.
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 is a data message or an acknowledgement. The Tag field is of 3 bytes in length and has 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 are used in the project.

Table Describing Tags

S. No.	Type/Condition	<type>	<subtype>	<_/A/N>
	Notification Type			
1	Join – request	N	0	_
2	Join – Success Response	N	0	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 - Retransmission Request	S	4	_
13	Sequence -Retransmission Neg Ack	S	4	A
	Election Type			
14	Election – Broadcast election start	E	0	_
15	Election - Broadcast Ack	E	0	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 can be found in the following table.

The following table describes all the payloads used in the chat system. It gives a brief description of the usage, sender and receiver of the payloads. It can be used as a reference to better understand the code.

Table To Understand Message Payloads

S. No.	Payload Type	Payload Format	Payload Comments
1.	Notification	N0_:0:LSeq:key	This is a notification sequence request payload. Used to get sequence number from the sequencer. 0: inital global sequence no.
		N0A:GSeq:0:participantListSize	Ack received for notification sequence request. 0: Local sequence participantListSize: No. of participants who are to receive the notifications
2.	Chat	C0_:GSeq:LSeq:chatMesg	Multicast payload by the sequencer. The sequencer sends this payload to all members in the participant list after a request has been sequenced.
		C0A:GSeq:LSeq:_	Acknowledgemnt payload sent by each member to the sequencer upon receiving a sequenced chat message
3.	Heart Beat	H0_:0:0:-	Heart beat message payload sent by the leader
		H0A:0:0:-	Heartbeat message payload sent by each live member in the chat room
4.	Election	E1_:0:0:-	Start election payload. Used to initiate a new election. This payload i sent to all higher numbered IP:PORT combination.
		E1A:0:0:-	Acknowledgement payload sent by all the higher IP:PORT members who received the election request to lower numbered IP:PORT which started the election.
5.	Centralized Sequencing	S0_:0:LSeq:key:msg	Sequencing request payload. Sent by a member to the sequencer when a new/retransmitted unsequenced message(msg) is detected.
		S0A:GSeq:LSeq:msg	Sequenced payload. Sent by the sequencer to the member who requested sequencing of that message (msg)

		S4_:0:missingSeq:_	Retransmission request payload. Sent by the sequencer when it detects a missing local sequence number. missingSeq is the LSeq from that client that sequencer should be receiving
		S4A:GSeq:LSeq:_	Negative acknowledgement payload. Sent by the member who was requested by the sequencer to retransmit an unsequenced message. The member sends this message indicating this missingSeq was not found in its seqBuffer.
6.	Decentralized Sequencing	S2_:0:LSeq	
		S2A:proposedSeq:LSeq:msg	
		S3_:GSeq:0	
		S3A:GSeq:LSeq:responseTag:???	
		S3N:GSeq:LSeq	

Extra Credits

1. **Message Priority :**
   Our implementation considers priority messages in ordering the messages at the sequencers end. All notifications are given a higher priority than normal chat messages.
2. **Decentralized Total Ordering:**
   Implemented a distributed sequencing algorithm to take care of decentralized total ordering.

References

std::map
http://www.cprogramming.com/tutorial/stl/stlmap.html
http://www.cprogramming.com/tutorial/stl/iterators.html

std::map.find()
http://www.cplusplus.com/reference/map/map/find/

inet_ntop() and inet_pton()
http://beej.us/guide/bgnet/output/html/multipage/inet_ntopman.html

htons()
http://beej.us/guide/bgnet/output/html/multipage/htonsman.htm
https://www.sgi.com/tech/stl/hash_map.html

pthread
http://randu.org/tutorials/threads/

User Manual

Compile the code using the makefile. Type the following on the command line to compile:

```
$$ make dchat
```

To start a new chat session type the following on the command line:

```
$$ ./dchat  <username>
```

To join an existing chat, you need to know the IP and port number of any one of the member of that group. Once you have obtained these, type the following on command line:

```
$$./dchat  <your_username>  <IP:PORT_NO>
```