CIS 505 - Software Systems
Project 3 - The Iterators

Shashank Agani Munivenkata Reddy	**agshash**
Parul Bhalla	**pbhalla**
Aakriti Singla	**aakritis**

Introduction

As part of the project, we will be implementing a fully ordered multicast chat server with a central sequencer/leader (chosen from among the current group members). The leader will be responsible for sequencing messages as well as determining who the current active clients are, detecting eliminating any inactive/crashed users from the chat (and making sure that the remaining clients have an up to date list of those currently in the chat). A new client can add to the system by sending request either to the leader or any active client. Each client will communicate with the leader from time to time. Along with this, various recovery mechanisms will be implemented, like message failure recovery, election of new leader in case the leader crashes or exits, etc.

The various modules implemented for the project are as follows:

- ***Naming Module*** - When a new user tries to join a chat by connecting to any currently active client, the current client will check
  - If he is a leader of a group/ chatroom, then add the new user to the group for which he is a leader
  - If he is not a leader of a group, he will reply back with the IP address and port number of the leader of the group that he is part of.
- ***Total Ordering/ Sequencer Module*** - The sequencer/leader maintains a queue of all the incoming messages. As soon as he gets a message from any client, he assigns it a sequence number and enqueues it. He dequeues the message and multicasts it to all the users in the chatroom. However, before he multicasts it, he has to ensure that all the clients have acknowledged the previous message. We will be using sequence numbers with messages. This will allow us to holdback messages in a holdback queue before all the messages before the current message is delivered.
- ***Recovery Module*** -
  - *Message Failure Recovery Module* - To ensure message recovery, we will use sequence numbers for each message being received by the server. Server will assign a unique sequence number (by adding 1 to the previous sequence number) and send back the message to client. Both will exchange acknowledgments to ensure message delivery. If the ACK is not received from the server, it will be retransmitted. A sample example is described later in the document.
  - *Node Failure Handling Module* - Have a separate thread to send a heartbeat message from the leader. If you don’t receive an ACK from the client within a specified time (reliable delivery is ensured using Sequence numbers), client is

assumed to be dead and we remove it from the set of users in the chatroom and also broadcast this removal of user to all the others in the chatroom. When the client doesn't receive a heartbeat message from the leader within a specified time, he invoke the ***Election Module***. The user with the highest ip (heuristically the most recently joined ip) gets elected as the winner and this gets broadcasted to all the other users. Then, all the other users will send an acknowledgement accepting the new leader. Also, to ensure that all messages from previous leader were received by all the clients, while election, clients will cross check the number of messages received and shares the required missing messages during inter-communication in election.

Basic Protocol

➔ ***Message Types***

- NORMAL CHAT MESSAGE
- ACK (Acknowledgements for messages)
- STATUS (addition of new user, removal of user etc)

➔ ***Data Structures***

```
class User {
        string ip
        int port
        string name
}

class ChatRoom {
        set<sockaddr_in> // Set of users in the chatroom
        user leader
}

class Message {
        string message;
        int sequence_num;
}
```

Queue<message> : This is maintained on the sequencer and client to ensure ordered messages and reliable delivery.

Map<user_ip, last_received_sequence_num> : This is maintained to ensure reliable delivery. It ensures that each message is delivered to each client in the correct sequence.

Design Specification for Clients

➔ ***Format of UDP Messages***

We will be using particular codes for particular requests/ services. For instance, the code `new_chat_room` will be representing the `new_chat_room` service.

Sample Overview of using Sequence numbers:

Client sends HELLO
Server sends HELLO 1 back to client assigning a sequence number
Client sends ACK 1
Server sends ACK 1

[Note: If the client doesn’t receive ACK from the server, he retransmits it.]

➔ ***Sending/ Receiving of Messages***

- **`new_chat_room`**`(leader_ip, name)`

  Start the UDP server on a particular port number. Initialize an empty set of users and set the leader object (name, ip)

- **`join_chat_room`**`(chatroom_ip, client_ip, name)`

  Client: `new-join-request chatroom_ip client_ip name`
  Peer-Client/ Leader: `new-join-request leader_ip`

  Resolve the leader of the chat room by the naming module and update the map with the leader of the group.

  After this, the newly added user’s ip is multicasted to all the other users of the group.

  Leader: `new-join-approval client_ip name`

- **`leave_chat_room`**`(chatroom_ip, client_ip, name)`

  Client: `new-join-request chatroom_ip client_ip name`
  Peer-Client/ Leader: `new-join-request leader_ip`

  Resolve the leader of the chat room by the naming module and update the map with the leader of the group.

  After this, the newly added user’s ip is multicasted to all the other users of the group.

  Leader: `new-join-approval client_ip name`

- **`new_message`**`(source_ip, message)`

  Client: *`new-message source_ip message`*