CIS 505 Distributed Systems
Project Final Report

Group Member: Bofei Wang, Di Wu, Hangfei Lin

Introduction

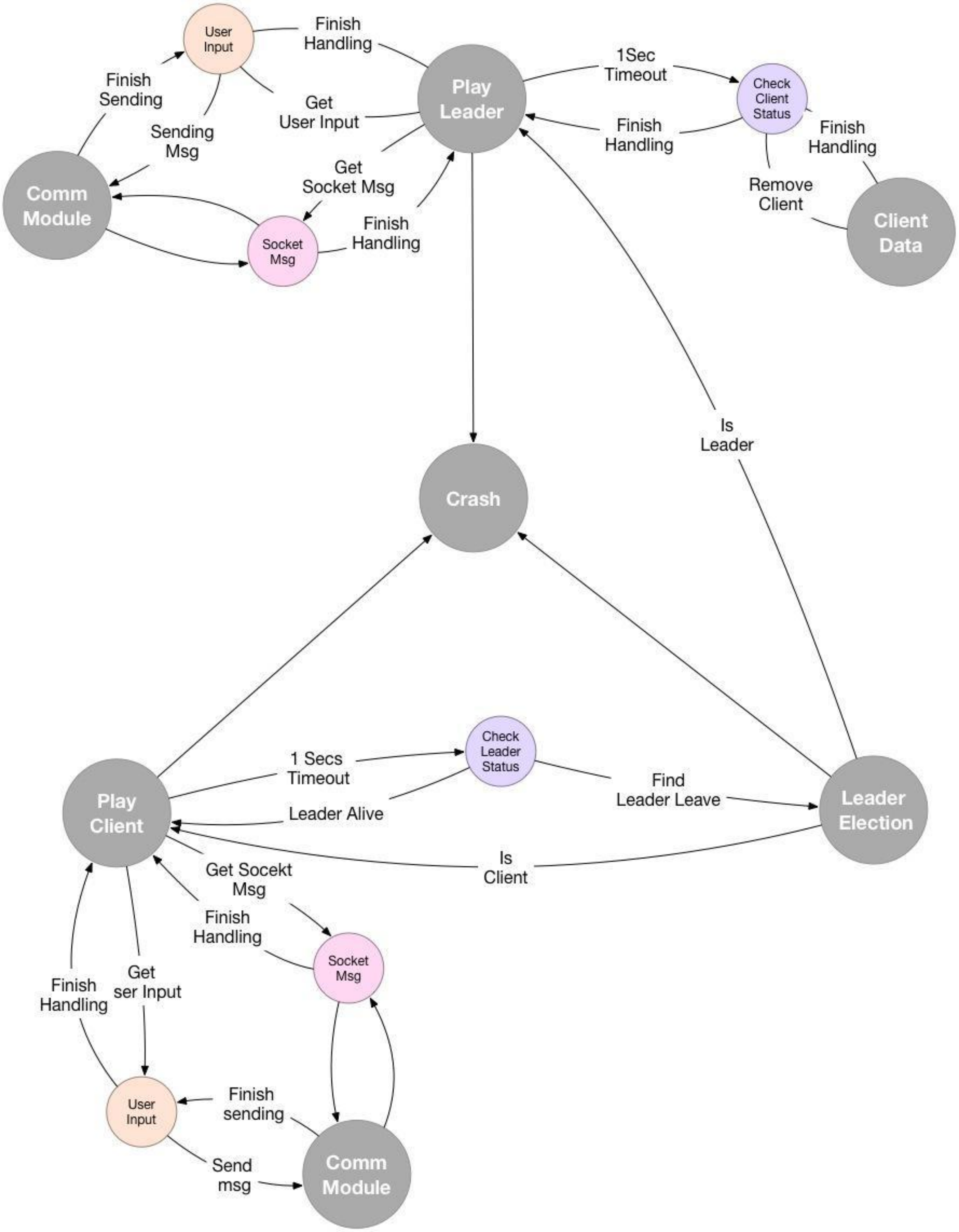
In this course final project, we would develop a distributed chat system with C/C++ under Linux. The main challenges and difficulties we would be facing and we would concentrate on is message passing, and fault tolerance.
This project proposal is organized in the following way: Architecture, Protocols, Modules, and Software Design Description.

Architecture

The architecture of the chat system is basically modeled as two roles, leader and client. Although there are some small differences between them, the main components and functionality are the same. The leader and the client could be modeled as four modules: messgae handling component, cooridnation component, status report component, leader election component.

The finite state machine for client and leader is as follows:

cis505_FSM


User Input
Finish Handling
Play Leader
1Sec Timeout
Check Client Status
Finish Sending
Get User Input
Finish Handling
Finish Handling
Sending Msg
Get Socket Msg
Remove Client
Comm Module
Client Data
Socket Msg
Finish Handling
Is Leader
Crash
1 Secs Timeout
Check Leader Status
Find Leader Leave
Play Client
Leader Alive
Leader Election
Is Client
Get Socekt Msg
Finish Handling
Socket Msg
Finish Handling
Get ser Input
Finish sending
User Input
Send msg
Comm Module

Protocols

The clients will use the UDP messages as follows,

NORMAL:

The messages' state is NORMAL, when users enter their messages with keyboards. the client will send the message in the format as "NORMAL, <timestamp>, <username>, <message>" to the leader. "timestamp" is assigned by the system and "username" is the name of user who sends the message. The leader will parse the message and then send a message in the same format to each member of the group.

JOINREQUEST:

The message will be sent by the user who is trying to join the group. The format of the message is like "JOINREQUEST, <timestamp>, <username>, <IP>, <portnum>". "username" is the name of user who is trying to join the group."IP" and "portnum" are the user's address and the number of the port that his/her system is listening on. If a client received the message, it will forward the message to the leader. And the leader will parse the message and prepare to send response back.

JOININFORM:

The leader of the group will send the message like "JOININFORM, <timestamp>, <username>, <IP>, <portnum>" to each member of the group when it received a "JOINREQUEST" that defined above. "username" is the name of new user who just join the group."IP" and "portnum" are the user's address and the number of the port that his/her system is listening on. When clients received the "JOININFORM", they will parse the message and store the information of the new member.

JOINRESPONSE:

The leader of the group will send the message like "JOINRESPONSE, <timestamp>, <leader_name>, < leader_IP>, < leader_portnum>, <num of group member>, <user1_username>, <user1_IP>, <user1_portnum>,…,<userN_username>, <userN_IP>, <userN_portnum>" to the client who is trying to join the group. "num of group member" is the number of members of the current group and the followings are the information of each member. "<userN_username>" is the user N's username and <userN_IP>, <userN_portnum> are his/her address and port number the system is listening on. The new user will parse the message and store the information of leader and each member.

CLIENTLEAVE:

When client is going to leave the groups chat or crashed. The client will send the message "CLIENTLEAVE, <timestamp>, <username>" to the current leader. "username" is the name of user who is leaving the group chat. After receiving the message, the leader will delete the information about the client and send the same message to each member who is still in the group. And the clients will delete this user’s information, too.

LEADERLEAVE:
When the current leader is going to leave the chat. The leader will send the message “LEADERLEAVE, <timestamp>” to all of the members that are in the group. Then clients will delete the leader’s information and prepare for election.

CLIENTSTATUS:
The clients will send the message “CLIENTSTATUS, <time>, <username>, <IP>, <Port>” to the current leader every fixed seconds. The time is the real time, not the timestamp assigned by the system. “username”, “IP” and “Port” are the client’s name, address and port number, respectively. The clients send the messages to the leader periodically so that the leader can realize the clients are still alive.

LEADERSTATUS:
The message “LEADERSTATUS, <time>” is sent to each member by the current leader. This is similar to “CLIENTSTATUS” that is also sent periodically. If clients have not got the message from leader for a long time, they will realize that there is something wrong with the connection.

ELECTION:
When the current leader leaves the group or crashed, the clients will send the message “ELECTION, <timestamp>, <username>, <IP>, <portnum>” to the clients that have a “bigger” name than itself. If a client received the “ELECTION” message and its username is “bigger” than the sender’s, it will send back a refuse message and send it own “ELECTION” message to other clients. Otherwise, it will just ignore the message and do nothing.

LEADERINFORM:
When a new leader is elected, the new leader will send “LEADERINFORM, <timestamp>, <leader_name>, <leader_IP>, <leader_portnum>” to other clients. When other clients get the message, they will update the information about the leader they stored.

Modules and Implementation

Message Handling and Coordination Module

The message preprocessing/sending/receiving module will be responsible for normal communications among clients and leader. It will receive and parse the message, then send corresponding messages to leader or clients based on the type of the messages.

The coordinator module will be responseble for the clients' and leader's leave or crash. When a new client join a group chat, the coordinator send the new user's information to the leader and other clients. And when leader crashes or leaves, the coordinator will inform all clients and send "ELECTION" message to the leader election module. The modules are implemented in two classes called "leader.c" and "client.c" described as follows.

Protocols

Protocols are those described above.

leader.c:

The first member of the group or the client that won the election would be the leader of the group. We are planning to create a class called leader. It has 4 fields, string leader_name, string leader_ip, int leader_port and map<string, string> group_member_info. The group_member_info is a map. Its key is the username of the clients in the group and the value is "IP:port" that is a string contains the client's address and port number that it is listening on. The leader has two methods, "init_group_info(string name, string server, int port_num)" and "do_leader()"; When a user wants to create a new group chat, he/she needs to call "init_group_info()" first. Then the fields, string leader_name, string leader_ip and int leader_port will be assigned with the values that passed in with the init_group_info() method. And it will also add the first key value pair into the group_member_info map. Then the system will call do_leader() method. do_leader() will not return or exit until the user wants to leave or the system crashed. When do_leader is called, the system will keep listening on the port and the keyboard. If the user entered messages with keyboard, the leader will send the "NOMAL" type messages to each member of the group. If the leader received some message from the socket, it will parse the message and do something based on the type of the message. The leader will forward the "NORMAL" messages to each member, send "JOINRESPONSE" message to the user that is trying to join the chat, send "JOININFORM" message to each member when new users joined the chat, send "LEADERLEAVE" message to members when the leader is leaving the chart and send "LEADERSTATUS" to members to report its state. The leader will also periodically update the group_member_info map and delete the members that seem to be inactive.

client.c:

We create a class called "client" for this module. The class has 7 fields, string local_name, string local_ip, int local_port, string leader_name, string leader_ip, int leader_port, and map<string, string> group_member_info. "local_name", "local_ip" and "local_port" is the client's username, address and port number respectively. "leader_name", "leader_ip" and "leader_port" is the current leader's username, address and port number respectively. And map<string, string> group_member_info contains the members' information including the leader. The key of the map is the member's name and the value is the "IP:Port" string that contains the user's address and port number. The class has two methods. The first one is int join(string server_name, int port_num, string username, string local_ip, int local_port). The system will call join when the user wants to join in an existing group chat. "server_name" and "port_num" is the address and port number of the leader or client in the group. And "username", "local_ip" and "local_port" is the username, address and port number of the user that is joining the group. The “join” method will send a “JOINREQUEST” message to the client or leader and wait for the “JOINRESPONSE” message. If the client does not get the message within a specific time, “join()” will return -1, which means the user cannot join the group. Otherwise, the client will parse the “JOINRESPONSE” message and store the leader’s and other members’ information in the map. If the client joins the chat successfully, then the system will call “do_client()” method. The method will not return or exit until the user wants to leave or the leader leaves or crashed. It will keep listening on the port and the keyboards. If it gets messages from keyboards, client will send “NORMAL” message to the leader. If it will send “CLIENTSATUS” message to the leader periodically to inform leader that it is still active. If the client gets “JOINREQUEST” message, it will forward the message to the leader. If it received “NORMAL” message, it will parse the message and print the text on the screen. If the client got a “ELECTION” message, it will compare the sender’s name and its own name. If its own name is “bigger”, it will send a refuse message to the sender and send its own “ELECTION” message to the clients with “bigger” usernames. When it got “JOININFROM” message, it will add the new user’s information to the map. And it will update the leader_name, leader_ip and leader_port when got “LEADERINFOM” message. It will record the time when got “LEADERSTATUS” message from the leader and check if the leader is still active.

Status Reporter(Updater) Module

In the chatting system we design, it is necessary to show the message that some client/leader in the group might crash. If the client is crashed, the leader needs to multicast this message to all the clients. If the client cannot hear from leader, it might need to start a new leader election. We thus need a reporting mechanism to make sure that the client and leader can know the most up-to-date status of each other.

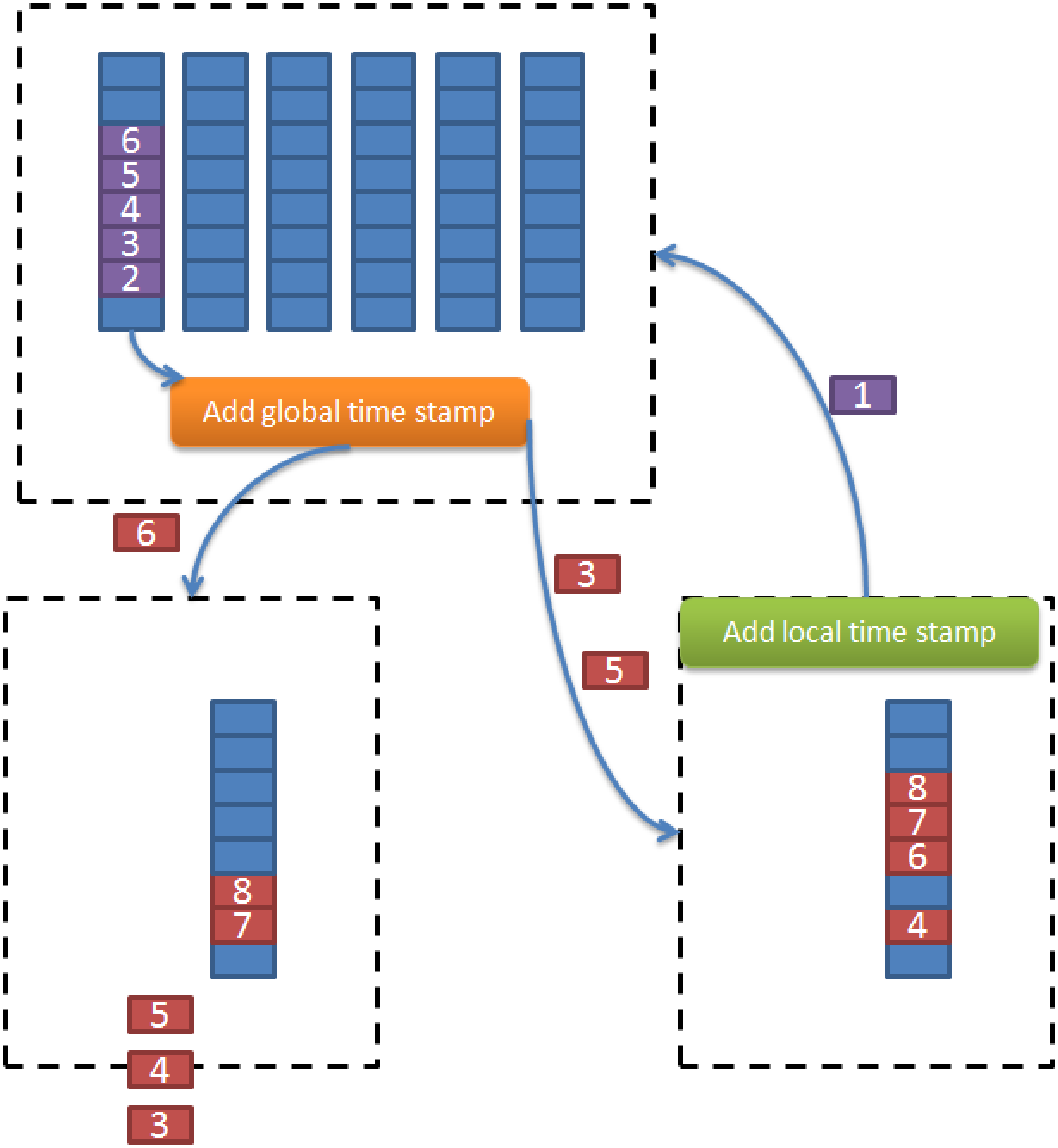
The basic implementation is: after the client joins the chatting group, it will send a CLIENTSTATUS message to the leader at a fixed rate (e.g. every 10s). When leader receives this message, it will update the status information stored locally. Then, it will send a LEADERSTATUS message back to the client. When client receives this message, it will also update the status information stored locally. From time to time (e.g. every 30s), the leader/client needs to check the local status information. If the leader find the time interval from the last updating time for some client has exceeds some threshold, the leader will remove the client from the group_member_info. If the client does not receive the updating message from the server, it will start a election by sending ELECTION message to the clients that has “bigger” number than it.

The detailed implementation of this is that the leader will keep a map<STRING> in which the username as key and with in its class. For the client side, it will start a new thread for repotring status within a certain amount of time. For the leader side, it will also start a new thread to travers all the clients repoting status in the status table. It can handle all the reporting message in the main thread as all other message. Whenever it recieves a CLIENTSTATUS message, it will respond a LEADERSTATUS message.

Sequencer Module

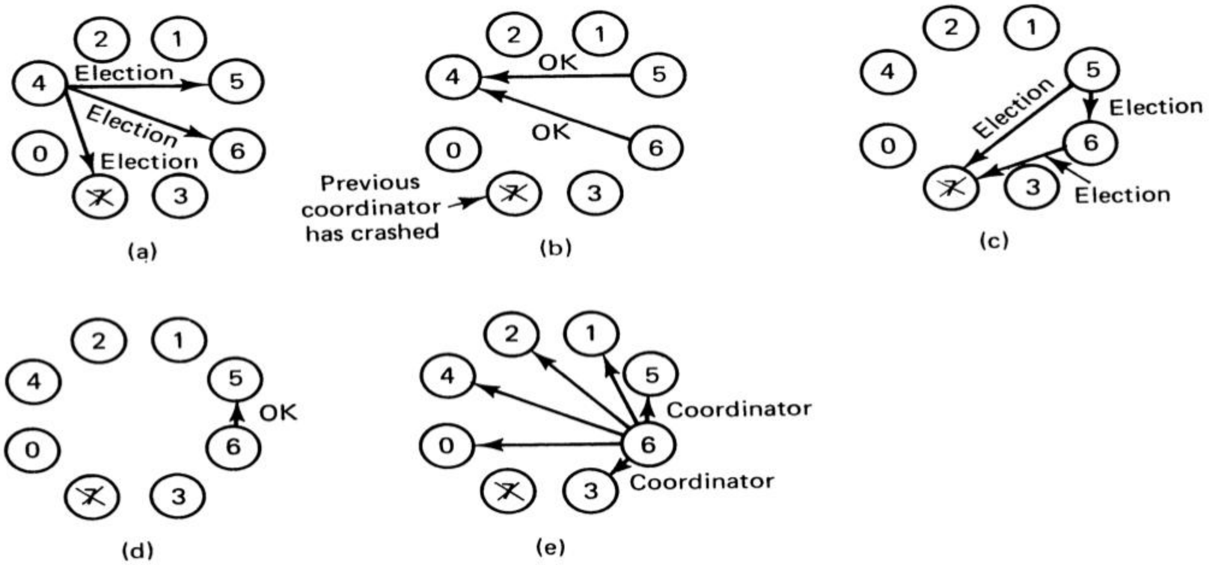
Since the distributed system is run on UDP, when messages are sent from the client to leader or leader to client, the sequence cannot be guaranteed. To solve this, we add a sequencing subsystem to the chatting. The sequencer preserves total ordering.

As is shown in the above figure, each message sent from the client to leader will contain a local time stamp. There is a multiple queue at the leader side and each queue is responsible for maintaining the message package received from each client. It also has corresponding expected time stamp for each client. If the message time stamp is larger than the expected time stamp, it will put the message into the corresponding client queue. Each time when there is a message, it will check the queue to see if there is remained messages. If it is the case, the server will dequeue all the messages that can be sent. Similarly, whenever there is a message sent from the leader to all clients, it will have a global time stamp. In the client side, it will also contain the expected global time stamp. If the received message's global time stamp is larger than the expected one, it will enqueue the message waiting for the smaller time stamp message coming.



Leader Election Module

In a distributed system, leader/coordinator might leave the system due to some normal reasons or crashes. When this happens, we handle this by electing a new leader among the remaining clients: one of the clients would become the new leader. There are various ways to implement leader election process. Here we use the classical "Bully Algorithm". An illustration from Distributed Systems is shown below.



Protocols

The protocols we used are shown in the "Protocols" chapter. It's indicated by "ELECTION" header and “LEADERINFORM” in the datagram.

Implementation

If the clients notice(by Status Report Module) that the leader leaves, the client would go into the leader election loop. This loop won’t end until the leader is elected and the chatting system attain normal communication status.

In the main function, once the client notice that the leader leaves, the client would fall into the election process, which is implemented in the LeaderElection class. Once the leader election finishes, the main function would do some clean up for different roles: clean the old data of previous chatting (message, and old timestamp). Then the main function would start the role transformation. If the client is not the new leader, it would just clean old data, set the new leader infomation, and try connecting the new leader. If the client is elected as the new leader, this new leader would clean old client data and set necessary leader data, and waiting for the clients’ connection.

Most of the work would be done in LeaderElection class. There is a main loop for in the function, which is in charge of listening for the socket connection from other clients and from user. If there is user input, it would friendly inform the user that it’s still leader election and be patient. The rest part is the actual implementation of the “Bully Algorithm”. We referenced the “Bully Algorithm” pseudocode from wikipedia below:

“When a process P determines that the current coordinator is down because of message timeouts or failure of the coordinator to initiate a handshake, it performs the following sequence of actions:

1. P broadcasts an election message (inquiry) to all other processes with higher process IDs, expecting an "I am alive" response from them if they are alive.
2. If P hears from no process with a higher process ID than it, it wins the election and broadcasts victory.
3. If P hears from a process with a higher ID, P waits a certain amount of time for any process with a higher ID to broadcast itself as the leader. If it does not receive this message in time, it re-broadcasts the election message.
4. If P gets an election message (inquiry) from another process with a lower ID it sends an "I am alive" message back and starts new elections.”

Some special situations and solutions

- If a process was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job.

Fault Tolerance and Handling

Message Lost: Client and Leader have queues to store messages they sent most recently and fields to indicate the next message’s timestamp they expect. Each message has its own timestamp. When the leader or clients get a message, they will check its timestamp firstly. If they find the message’s timestamp is larger than what they expect, they will send a request to leader or clients requiring them to send the message they expect.

Duplicate Messages: Neither client nor leader will print duplicate messages on their screens, since they will check the timestamps of the messages. If the timestamp is smaller than what they expect, it means that they have already printed the message before, the message is a duplicate one and they will not print it.

Crash/Leave of Client/Leader: The leader will send “Heartbeat” requests to each client every 3 seconds. Then active clients will response “ACK” messages to the leader. If a any client does not response “ACK” message within 30 seconds, the leader will assume that the client has left or crashed and send messages to other active clients to inform them to delete the information of that inactive client. Also the clients will send “Heartbeat” messages back to the leader, and the leader will response with “ACK” message. If the clients do not get the leader’s response with in 30 seconds, they will assume the leader has crashed and start to propose “ELECTION”.

Software Design Description:

For this course project, it's a medium-sized software. So to manage complexity and improve efficiency, we would first have an overall architecture design of the system and separate some of it into de-coupled modules. We would first assign the modules to each group member and reach an agreement on the interfaces of different related modules. As the software develops, we would frequently discuss, combine and test all the modules and the whole software. The detailed labor of division is as follows:

Overall coordination: Di Wu

- main function
    - Recieve and send messages between leader and clients
    - Correspond to different message types
- Interface:
    - In: when status report find that the leader is not responding
    - Out: send “ELECTION” message to Leader Election module

Status Reporter, Sequencer: Bofei Wang

- main fucntion
    - Client/sevre sends the status report to each other at a fixed rate.
    - If client/server has not reported in a certain amount of time, remove the item from the map
- Interface:
    - Out: send “ELECTION” message to Leader Election module

Leader Election: Hangfei Lin

- Leader election happens when the original leader leaves or the original leader crashes.
- Algorithm used to implement: Bully Algorithm
- Detailed Trigger Situation:
    - When a client finds that the leader is not responding.
- Interface
    - In: when status report find that the leader is not responding
    - Out: send results to coordinator

References:

Hector Garcia-Molina, Elections in a Distributed Computing System, IEEE Transactions on Computers, Vol. C-31, No. 1, January (1982) 48-59