CIS 505 final project

Milestone 2

Team member: Yuanjie Chen, Siyu Qiu, Zhi Li
Pennkey: yuanjiec, siqiu, zhili
Programming Language: C/C++

System workflow

thread 1: Receive

Receive message, call parser method from parser.h.
Then parse the request.
According to request name, call corresponding method in ChatNode.h

thread 2: TypeMsg

listen to the keyboard
call stub_send
user send msg to leader
leader multicast msg

thread 3: CheckQueue

Check whether message queue is empty or not. If it is empty, the leader could exit. If it is not empty, dequeue the message and multicast it to all users in the group and then exit.

thread 4: HeartBeat

Leader: check if current users are alive, if not then update userlist and multicast.
User: check if leader is alive, if not then raise leader election.

Low-level Communication Design

// multicast.h
char* stub_connect(char* Tip, char* Tport);
// use stub_send to send msg to potential node to test if ip:port is valid, then use stub_create to bind a port for listener
char* stub_receive();
// listen on socket binded; send ack msg back to sender; forward msg received to parser
char* stub_send(char* Tip, char* Tport, char* msg);
// send msg to destination ip:port; timeout if not receiving any ack msg back after 5s(arbitrary)
char* stub_create(); // create a chat and bind chosen port (random) for listener
char* getlocalinfo(); // get current IP and port
void* get_in_addr(struct sockaddr *sa); // get information from socket addr

General Design

In our design, low-level communication is separate from high-level communication (message generation, total ordering, leader election etc.). Its major responsibility is to send given

message to the specific destination given IP and port, as well as listen on specific port and forward messages received to high-level parser, which will then call high-level functions to handle different requests/scenarios. The interaction between low-level communication and high-level message processing is similar to stub in RPC. Here in our implementation, multicast.c acts as the stub, which is in charge of message sending and receiving, while other cpp files are in charge of high-level functionalities.

Fault Tolerance
Since we need to use UDP for message transferring, we have to design a small application-level protocol that would handle possible packet loss and reordering. The steps are as follows.
example msg: **[size][sequence] RequestName#selfip_selfport_content**

1. Datagrams less than MTU (plus IP and UDP headers) in size (say 1024 bytes) to avoid IP fragmentation.
2. Fixed-length header for each datagram that includes data length and a sequence number, so we can stitch data back together, and detect missed, duplicate, and re-ordered parts.
3. Acknowledgements from the receiving side of what has been successfully received and put together.
4. Timeout and retransmission on the sending side when these acks don't come within appropriate time. Current implementation is to first check if destination node is alive after the first timeout, then react correspondingly.

Extra
Encryption will be done through low-level communication. The idea is to hash the message (after adding header mentioned above) before sending the message, then decrypt on receiving the message.

High level Communication Design

Message Protocol
When sending messages, user send the message to the leader and then leader multicast the message to all users in the chat group. Each user use a hold-back queue implemented by min_heap. And when user receive the message from the leader, it compares the total number from the receiving message and its local receivedNumber for total ordering.
format: RequestName#selfip_selfport_content

Data Structure
User Class:
variable:
IP: ip address of a user

nickname: nickname of a user
ID: it is used for leader election module, which choose the highest ID as the new elected leader
total: it records the current number of messages the leader has received. When leader multicast messages, the total will also be sent to each client. It is used for total ordering.
leader: bool variable to check whether this user is leader or not.

method:
getter and setter for each variable
multiple constructors

```
class User
{
private:
	string IP;
	string nickname;
	int port;
	int ID;
	int total;
	bool leader;
public:
	string getIP();
	void setIP(string IP);
	string getNickname();
	void setNickname(string nickname);
	int getPort();
	void setPort(int port);
	int getID();
	void setID(int ID);
	int getTotal();
	void setTotal(int total);
	bool getIsLeader();
	void setIsLeader(bool isLeader);
	User(string IP, string nickname, int port, int ID, int total, bool isLeader);
	User(string IP, string nickname, int port);
	User();
};
```

variable:
userlist: a list of current alive users
me: local user
rNum: number of msg received

holdback: small end message queue
BlockingQueue msgQueue;

method:
createChat(User user)
A new user create a chat. Then the chatNode object add the user to the userlist and set this user to be the leader.

reqLeader(string Tip, int Tport)
Client request leader information from other client using target IP and port.

sendLeader(string Tip, int Tport)
Send leader information back to new added client

connectLeader(string Tip, int Tport)
Connect to leader. add new client to userlist

updateUserlist(vector<User> vector)
Each client received the new userlist from the leader and then update the new userlist.

addUser(string ip, string name, int port)
Add new user to userlist and then multicast new userlist to other clients

void deleteUser(string ip, int port)
Delete user from userlist

multicastUserlist()
Multicast new userlist to other clients

sendMsg(string msg)
send message to leader

multicastMsg(string msg)
leader multicast message to all users

electLeader()
send its own ID to neighbor user. and use chang robert ring algorithm to elect new leader

multicastLeader()
send leader information to all user in the chat group

exitChat()

if I'm leader, delete myself from userlist, set a new leader, multicast new userlist it to all users. Then check message queue, if it is empty, exit. If it is not empty, wait until all messages has been multicasted to all users

if I'm not a leader, request deleteUser method from leader, leader will multicast userlist.

ChatNode Class:

```
class ChatNode
{
private:
	ChatNode(){};
	static ChatNode* node;
	vector<User> userlist;
	User me;
	int rNum;
	BlockingQueue msgQueue;
	mutex userlistMutex;
	mutex meMutex;
	mutex rNumMutex;
	mutex totalMutex;

	struct Compare{
		bool operator() (int a, int b){
			return a > b;
		}
	};
	priority_queue<int, vector<int>, Compare> holdback;

public:
	static ChatNode* getInstance();
	vector<User> getUserlist();
	void setUserlist(vector<User> userlist);
	User getMe();
	void setMe(User user);
	int getPNum();
	void setPNum(int number);
	int getRNum();
	void setRNum(int number);

	void createChat(User user);
	void reqLeader(string Tip, int Tport);
	void sendLeader(string Tip, int Tport);
	void connectLeader(string Tip, int Tport);
```

```
        void updateUserlist(vector<User> vector);
        void addUser(string ip, string name, int port);
        void deleteUser(string ip, int port);
        void multicastUserlist();

        void sendMsg(string msg);
        void multicastMsg(string msg);
        electLeader();
        multicastLeader();

        exitChat();

};
```

Leader Election Algorithm

Chang and Roberts Ring algorithm

With the Chang & Roberts algorithm, the coordinator selection is performed dynamically as the message circulates, so the election message contains only one process ID in it - the highest numbered one found so far. Think about the ring algorithm: if our goal is to pick the largest process ID to be the coordinator, there is no point in ever affixing smaller process IDs to the list in the message.

Every election message starts with the process ID of the process that started the message. If a process sends an election message, it identifies itself as a participant in the election regardless of whether it initiated the election or forwarded a received election message,
To pick the surviving process with the highest-numbered process ID, any process that detects the death of the current leader creates an election message containing its process ID and sends it to its neighbor, as was done in the ring algorithm. Upon receiving an election message, a process makes the following decision:

If the process ID of the message is greater than the process ID of the process, forward the message to the neighbor as in the ring algorithm. The current process is not a contender to become the coordinator since the election message tells us that a process ID with a higher number exists.

If the process ID of the message is less than the process ID of the process then replace the process ID of the message with the process ID of the process. Then forward the message to the neighbor as in the ring algorithm. This is the opposite of the previous decision. The current process has a higher process ID than any other process that encountered this election message, so it has a chance of becoming named coordinator.

If the process ID of the message is less than the process ID of the process and the process is already a participant, then discard the message. If the process is a participant, that means it already sent an election message to its neighbor with either itself or a higher-numbered process ID. Forwarding this message on would be redundant.

If the process ID of the message is the same as the process ID of the process, that means that the message has circulated completely around and no higher-numbered process encountered the message since it would have replaced the message's ID with its process ID. The process therefore knows it is the coordinator and can inform the rest of the group.