CIS505 ChatSystem Documentation

Milestone3

Team member: Yuanjie Chen, Siyu Qiu, Zhi Li
Pennkey: yuanjiec, siqiu, zhili
Programming Language: C/C++
Tasks accomplished: regular version + extra credit 5.1, 5.2, 5.3, 5.4, 5.5

System workflow

thread 1: Receive

Receive message, enqueue and notify other thread which is blocked when the queue is empty.

thread 2: TypeMsg

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

thread 3:

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.

thread 5:

Check parser. May wake up by thread 1. Call parser method from parser.h.
Then parse the request, parsing the request name, call corresponding method in ChatNode.h

Low-level Communication Design

// multicast.h
string 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
string stub_receive();
// listen on socket binded; send ack msg back to sender; forward msg received to parser
string 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)
string stub_create(); // create a chat and bind chosen port (random) for listener
string getlocalinfo(); // get current IP and port
string get_in_addr(struct sockaddr *sa); // get information from socket addr
void stub_checkSendRate(); // check client's send rate to user, and slow down send rate if soar above threshold, and cancel the delay when rate drops low.

int stub_getSendMsgNum(); // check how many messages have this client sent to the current leader

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.

Update since Milestone2:

- update stub_send
    - now stub_send could choose to send infinite times, send at most once and at least once
    - enable msg send queue by first enqueue msg before sending, and dequeue on receiving "OK" ack back
    - if not receiving "OK" (timeout), then will continue to resend SENDMAX(3) times iteratively and return "ERROR"
    - if receiving "RESEND" then keep sending until receiving "OK"
    - every time stub_send tries to resend, it first dequeue the current msg then enqueue a second time on reinitializing the same msg, however the id of the same msg may change.

- update stub_receive
    - stub_receive could now check the size of packet received and compare with packet header

    - send "RESEND" request on receiving unintact msg
    - send "OK" ack on receiving full & intact msg

- update stub_connect
    - now stub_connect use stub_send(Tip, Tport, msg, request = 1) to check if destination available by sending msg only once
    - will receive "ERROR" on not being able to connect
- **datastructure change**
    - add in struct clock_send
    - use targetIP:targetPORT as key to store how many valid messages have been sent to the target
    - num will only update on sending non-resend messages(resend < 1)
    - use value of num(different among targets) as sequence number as part of msg header
    - add in struct msg_monitor
        - use receivedIP:receivedPORT as key mapping to sequence number received
        - a vector holdback is used to store those sequence num that are way ahead of it's supposed to receive
        - latest is used as a classifier that seq received below latest will be defined as duplicate
        - latest only updates on receiving the msg with correct seq number(current latest+1)
        - on receving msg with correct seq number, it will then try to clear the member in holdback vector one by one incrementaly (e.g. if 3,4 are in holdback vector, now 2 arrives, it will then trigger the erase of 3 and 4, and update value of "latest" before erase), keep latest always the seq of latest valid msg received.

    - update stub_receive to adapt the msg_monitor datastructure
        - first check if data is intact
            - if intact: reply "OK", then if sender already in monitor list, update seq num and holdback vector accordingly.
            - if not intact: then simply reply "RESEND" as ack.

    - update stub_send to adapt the clock_send datastructure
        - first check if it's a resend request
        - (either triggered by not intact resend request "RESEND" or resource temporarily unavailable)
        - if resend (request >= 1) use same msg without adding new header to it
        - if not resend (request == 0)

    - update sequence number accordingly, use the same sequence number for sending that msg in the header
    - adding new header to it, format: data size(fix length == 5) + IP:PORT + "STA" + sequence number(target specific) + "END@" (@ is for parser). Overall something like: 00045192.168.1.100:26411STA00002END@what's up

- **Traffic Control (Extra Credit 5.1)**
    - change the bool value of **checkRate** to true in **multicast.cpp** and build the program, then the client program will automatically track the rate of sending to the leader periodically, slow down the rate of sending by adding delay to each send when the rate is above the threshold, and cancel the delay when the rate drops below the threshold. There'll be no limitation for leader's sending rate.
- **Encryption (Extra Credit 5.4)**
    - change the bool value of **encrypted** to true in **multicast.cpp** and build the program, then the transmission process will be fully encrypted using encryption techniques based on XOR and additional substitution techniques, then will be decrypted on the receiver side.
    - hashkey can be changed, for now it's "happy". Since the sendto function of UDP tranmission requires message to be of const char* type, using XOR will result in the encrypted message to contain multiple '0's, and the first of which will be seen as a null terminator thus make the message sent unintact. The solution is to substitue all '0's with ! and append the position of substitution to the tail of encrypted message. All messages will be encrypted (including the tail part). The tail will then be used in the receiver's decryption function.

High-level Communication Design

Leader election detail design

// Update since Milestone 2

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

We use **Chang and Roberts Ring algorithm** to elect new leader
checkAlive():
Leaders ping other users every 10 seconds. If other users are down, erase that user in the userlist and then multicast the new userlist to other users.
Other user ping leader every 10 seconds. If leader is down, call the leaderElection function to elect new leader. we use Chang and Roberts Ring algorithm to elect new leader.

leaderElection():
Send **my ID** to the next neighbor. If next neighbor is the leader who is down, then send to the next one. And set my participant as true. This implementation has achieved the purpose raised in **Extra Credit 5.5**.

sendUID(int id):
Get id from pre neighbor. If I has participated in electing new leader and the id received is bigger than my id, then ignore. If I has participated and the id received is smaller than my id, then send my id to the next available neighbor. If I has participated and the id received is equal to my id, that means my id is the smallest id and it has been passed around the circle. Then call setNewLeader function to set myself as the new leader.

setNewLeader():
This method should only be called in the new elected leader side. Ping the former leader to find it. Then erase the former leader in the userlist. Then set myself as the new leader with the correct information, such as total number, next id number etc. Then multicast the new userlist to other users.

Message Queue

leader side:

- receive message queue:
    - Implementation: blocking queue
    - Mechanithem: two threads together visit this queue
        - enqueu thread: enquee every message/request from all users
        - dequeue thread: blocked when the queue is empty while pop out message whenever there are things inside

all user side:

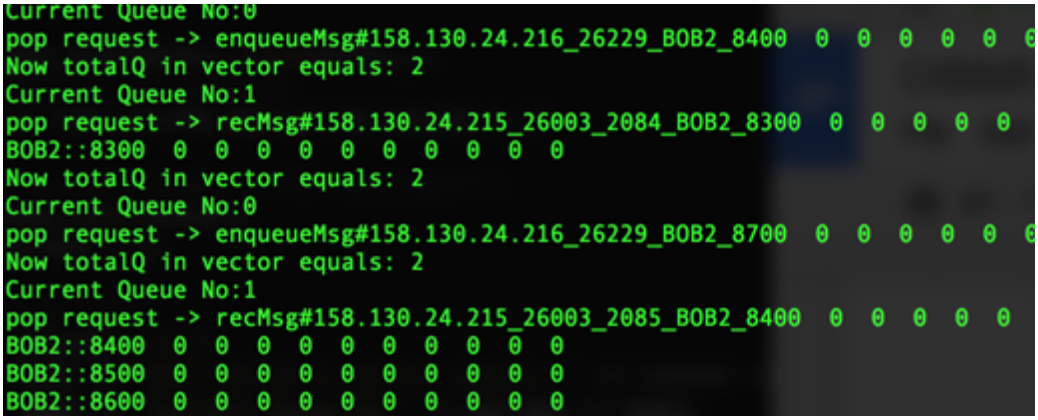
- hold back queue:

    - To realize message total ordering, every user has a holdback queue to store coming messages so that messages are shown in order.
- message priority queue: **(Extra Credit 5.3)**
    - We implement a priority queue to store message for each user.
    - For the time when there are a large volume of messages come, sort those requests according there type. Deliver the request to upper layer according after sorting.
    - detail implemenation of priority is assigned at parser.cpp, mycomparison.cpp, blocking_priority.cpp;

```
if(req.compare("sendUID") == 0){
    comparator = 1;
}else if(req.compare("multicastUserlist")==0){
    comparator = 2;
}else if(req.compare("updateUserlist")==0){
    comparator = 2;
}else if(req.compare("deleteUser")==0){
    comparator = 3;
}else if(req.compare("addUser")==0){
    comparator = 4;
}else if(req.compare("sendLeader")==0){
    comparator = 4;
}else if(req.compare("connectLeader")==0){
    comparator = 4;
}else if(req.compare("recMsg")==0){
    comparator = 5;
}else if(req.compare("enqueueMsg")==0){
    comparator = 5;
}else if(req.compare("newUser")==0){
    comparator = 6;
}else if(req.compare("exitNotice")==0){
    comparator = 6;
}
```

- fair queuing **(Extra Credit 5.2)**
    - The implementation is based on round robin algorithm, assuming that the processing time of each request is about the same, then when burst of incoming messages happen on the leader end, to ensure fairness, one message of each user will be poped out from the queue, and after one round, user with empty message will be erased from the current queue. This process will continue until no user is in the queue, and then block until a new message arrives.
    - The implementation is based on vector of priority queue. Blocking will happen if no user is available in the vector, and start round robin other wise.
    - proof of working can be seen in the following screenshot: when burst of incoming messages happen on leader side, our implementation will use

round robin algorithm and pop one message each from different user.



User operation detail design

showCurrentUser():
print the name, id and port of each user in the current userlist

multicastUserlist():
This method will only be called within the leader. The leader multicast the new userlist to all other user via specific message format so that other users could receive and understand how to parse.

reqLeader(string Tip, int Tport):
This method is called when a new user want to join the chat. It will send my ip and port to the leader via the target connect ip and port.

sendLeader(string Tip, int Tport):
Traverse the userlist to get the leader ip and port. Then send the target ip and port back to the requested new user.

connectLeader(string Tip, int Tport):
Send my ip and port the connect with leader.

addUser(string ip, string name, int port):
This method should only be called within the current leader. Construct a new user and push it in the userlist and then call multicastUserlist.

updateUserlist(vector<User> newuserlist):
Use the received newuserlist to update the current userlist.

userExit():
This method is called when standard input capture a ctrl-D signal. If I am the leader, then erase myself in the current userlist and set the next neighbor as the new leader because it has the smallest id. Then multicast new userlist. If I am not the leader, then send my ip and port to the leader, let the leader to erase my information in the userlist.

deleteUser(string Tip, int Tport):
Erase the user identified by Tip and Tport. Then multicast the new userlist to all other online user.

Message delivery detail design
sendMsg(string message):
Send the message to leader and let leader enqueue the message.

enqueueMsg(string msg):
Enqueue the received msg to the msgQueue.

checkMsgQueue():
Check msgQueue. If the msgQueue is not empty, dequeue it and then multicast it to all online user.

multicastMsg(string message):
Send the message with the total number to all other online user. The total number is used for total ordering.

recMsg(string name, int total, string msg):
If the total number is equal to my receivedNumber, it means that this message is the one I expect to receive. Then increment rNum and call showMsg to show it. Then use while loop check the holdbackQueue which is a priority queue. If the peek element is equal to the rNum, then dequeue it and show it. If not, just break.
If the total number is not equal to my receivedNumber, just enqueue the message in the holdback queue. It means that this message arrives early than the message I expect to print out instantly

showMsg(string name, string msg):
Just print out the msg to the standard output.

// Former document in Milestone 2
Data Structure for basic functionality
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();

};
```

User Manual

Regular Part

1. On speclab, use make to build the executable file dchat, make sure the value fed to function “getlocalinfo” is “em1” for speclab in order to get real ip address.
2. **Create** a chat: ***$dchat client1***
   You’ll see the ip:port on which you’ve created your chat, as well as a userlist containing just yourself. Chat to yourself freely :)
3. **Join** a chat: ***$dchat client2 x.x.x.x:xxxxx*** #x refers to ip:port
   If you’ve entered in the correct ip:port of anyone in an existing chat (either leader or normal user), you’ll be joining the chat successfully, returned with your ip:port, and shown a userlist containing all live users in the chat as well as leader information, also all existing users will be notified with “NOTICE $yourname JOIN in the chat”. Then you could chat freely.
4. **Exit** a chat: normally it’s preferred to exit a chat with **Ctrl + D**, then all existing members will be notified with “NOTICE $yourname EXIT the chat”, and an updated userlist will be multicasted from leader to all existing members and shown on screen. If you’re currently a leader, Ctrl+D will also trigger the assignment of a new leader before the program exits, and other members will receive the same notification as above. You could also exit with **Ctrl+C** or **Ctrl+Z**. If you’re currently the leader, other member will sense your absence and raise leader election. Otherwise, leader will sense the absence of a current member, remove it from the current list, then multicast the new userlist to all existing members.

EC Part *(full program provided in /EC)*

1. **Traffic Control:** change the bool value of **checkRate** to true in **multicast.cpp** and build the program, then the client program will automatically track the rate of sending to the leader periodically, slow down the rate of sending by adding delay to each send when the rate is above the threshold, and cancel the delay when the rate drops below the threshold. There’ll be no limitation for leader’s sending rate.
2. **Encryption:** change the bool value of **encrypted** to true in **multicast.cpp** and build the program, then the transmission process will be fully encrypted using encryption techniques based on XOR and additional substitution techniques, then will be decrypted on the receiver side.
3. Low-level communication **debug mode**: change bool value debug to true in multicast.cpp, then make and run the project, and now you could see what’s happening for different requests in low-level communication.