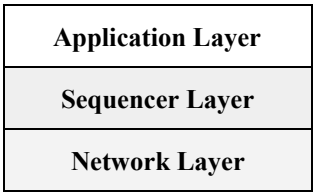
CIS 505 Project Final Report

Team Name: DVoter
Team Member: Dongni Wang (wdongni), Yisi Xu (yisixu), Wei Hu (huwei)
Project Name: dchat
Abstract: In this group project, our team (Dongni Wang, Yisi Xu, Wei Hu) implemented a fully distributed text-based group "chat" system built using C/C++ under Linux. We also finished extra credit tasks including implementation a decentralized totaling ordering multicast chat system, graphic user interface design etc.

1. High Level Design



Total ordering Multicast (Sequencer-based approach):
Special process elected as leader (sequencer)
Send multicast at process Pi:

- Send multicast message M sequencer

Sequencer:

- Maintain a global sequence number S (initial 0)
- When it receives a multicast message M, it sets S = S + 1, and multicast <M, S>

Receive multicast at process Pi:

- Pi maintains a local received global sequence number Si (initial 0)
- If Pi receives a multicast <M, S(M)> from sequencer, it buffers it until Si + 1 = S(M)

Membership Management
Each node has a Member (user) object, which holds a list of MemberListEntry and corresponding group membership information. The Member object provides interfaces for updating the membership information. The membership structure contains all the knowledge the current node knows about the existing group.

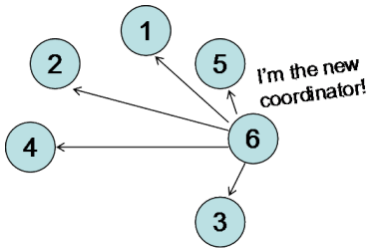
Failure Detection
Leader will perodically send heartbeats to the other member nodes, if the leader does not hear back from a node (with a fixed number of retries), the leader will initiate a leader election; Similarly each member node will perodically pings the leader, if the leader does not response, the member node will initaite a leader election. Note that it is possible that a node is mistakenly detected to be fail and start a new leader election. It's ok to start an election even no node fails. As long as the election finishes, the membership changes will be reflected in each participating nodes. The new leader will be elected to perform the sequencer duties.

Leader Election (Bully Algorithm)
All processes know other processes' ids
When a process finds the coordinator has failed (via the failure detector)

- If it knows its id is the highest, it elect itself as coordinator, then sends a *Coordinator* message to all processes with lower identifiers. Election is complete.
- Else it initializes an election by sending *Election* message.
    - Only to processes that have a higher id than itself

- If receive no answer within timeout, calls itself leader and sends *Coordinator* messages to all processes with lower id
- Else wait for *Coordinator* message

A process receives an *Election* message replied with *OK*, and start its own leader selection protocol (unless it has already done so)



3. System Workflow

Thread 1: Listening on socket, handle message and listening for next message.
Thread 2: Listing for user input from the command line, put a chat message into a buffer (this buffer queue is for the purpose of holding messages when a election is started) queue when a chat message is captured.
Thread 3: Fetch messages from the buffer and send it to the leader.
Thread 4: Whenever a message is put into the queue to be displayed, the thread fetch it from the ready message queue and display it to the terminal.
Thread 5: Perodically send heartbeats. Multicast heartbeats to all group member the node is the leader. Sent heatbeat to leader if the node is not the leader. When failures is detected this thread initiate a leader election.
(Possible Threads): When Thread 1 receive a *ELECTION* message (from some processes with lower process id), a new thread will be started to initiate a leader election

3. Detail Implementation

Class Design:
Project: dchat

- **stdincludes.h**
- **dchat.cpp (Application Layer)**
- **DNode.h**
- **DNode.cpp**
- **Member.h (Member Data Structure)**
- **Member.cpp**
- **DNet.h (Network Layer : UDP)**
- **DNet.cpp**
- **Queue.h (Data Structure)**
- **Queue.cpp**
- **Handler.h**
- **Handler.cpp**

Network Level Implementation
Classes:

- **DNet.h DNet.cpp**

API:

- int DNinfo(std::string & addr);

```
int DNsend(Address * toaddr, std::string data, std::string & ack, int times);
int DNrecv(Address & fromaddr, std::string & data);
```

Description:

DNinfo function gets the local address where the node is currently listening to. DNrecv read messages send to the port that the node is listening to and pass the message to a *Handler* class, then send back response message. DNsend will send the message to the given address and gets the response message.

Membership Management

Classes:

Member.h Member.cpp

class Member has the following varibales

```
std::vector<MemberListEntry> memberList;
MemberListEntry * leaderEntry;
```

API:

```
bool isLeader();
```

getLeaderAddress/getLeaderName

updateLeader/addMember/deleteMember/printMemberList

Description:

Member class store all the information that the current node knows about the group membership. This class also provides common modification interface for updaing the membership information.

Multicast Communication, Failure Dectioon, Leader Election

Classes:

DNode.h DNode.cpp Handler.h Handler.cpp

API:

DNode.h DNode.cpp

```
// Leader Election
void nodeLoopOps();
void startElection();
void updateElectionStatus(int new_status);

// Membership Update
int introduceSelfToGroup(std::string joinAddress, bool isSureLeaderAddr);
void initMemberList(std::string member_list);
void addMember(std::string ip_port, std::string name, bool toPrint);
void deleteMember(std::string ip_port);

// Multicast message
void multicastMsg(std::string msg, std::string type);

// Message routnine
void addMessage(std::string msg);
void sendMsg(std::string msg);
void sendMsgToLeader();
int sendNotice(std::string type, std::string addr);
void multicastNotice(std::string type);
```

Handler.h Handler.cpp

```
std::string process(Address & from_addr, std::string msg); // return ack message
if (recv_msg.find("#") == 0) { // Start with a #
        // D_M_ADDNODE/D_M_MSG/D_LEAVEANNO
} else {
        // D_CHAT/D_JOINREQ/D_HEARTBEAT/D_ELECTION/D_COOR
}
```

Description:

The Handler has a reference to DNode class and the sequencer layer, which handles multicast communication, failure detection and leader election messages.

holdback_queue * m_queue: This is a holdback queue that deliver messages in global order.

Whenever user receive a message from a member node, it will check it again the sequence number of that node which the leader has seen. The leader will ensure that message delived also follows the order of how user input the order. If the leader receive a chat message whose sequence number is not expected, it will acknowledge a resend request to the member (which is essential is the sequence number that the leader has seen for the member). The leader will then resend the messages that could possibly be lost. This is the at least once semantic.

Message Format:

```
#MSG#SEQ#username::Message, multicast from the sequencer
#ADDNODE#SEQ#ip#port#name, multicast from the sequencer
#LEAVEANNO#seq#name#ip:port
CHAT#Seq#username::Message
JOINREQ#PORT#Name
JOINLEADER#LEADERIP#LEADERPORT#LEADERNAME
JOINLIST#initSeq#LEADERNAME#ip1:port1:name1:...
HEARTBEAT#port
ELECTION#ip:port, start election
COOR#name#ip:port, announce leadership
```

Application Level Workflow

To start a group chat,

```
Node * node = new DNode(name);
```

To join a group chat,

```
Node * node = new DNode(name, ip_port);
```

Then,

```
node->nodeStart();
// Thread: Listening for input
std::thread thread_sendMsg(sendMsg, node);
// Thread: Receive chat messages
std::thread thread_recvMsg(recvMsg, node);
// Thread: Receive chat messages
std::thread thread_displayMsg(displayMsg, node);
// Thread: Track heartbeat
std::thread thread_heartbeat(heartBeatRoutine, node);
// Thread: Send message to leader
std::thread thread_sendMsgToLeader(sendMsgToLeader, node);
thread_sendMsg.join();
thread_recvMsg.join();
```

```
thread_displayMsg.join();
thread_heartbeat.join();
thread_sendMsgToLeader.join();

// Clean up and quit
node->nodeLeave();
delete node;
```

* Extra Credit implementation

5.1 Traffic Control

Note: using our implementation of the chat system, only the chat messages delievery rates are dependent on users. Therefore, the only type of messages that would potentially cause traffic congestion in our current system is D_CHAT. On this note, we mainly take care of the D_CHAT messages. But the traffic control system shall be easily extended to control the full incoming messages.

Implementaiton Details:

The sequencer would keep a dictionary to record the number of messages received from each member in a fixed time interval. And the sequencer will also have a thread to perform the checking task: at the end of each time interval, the sequencer would traverse the dictionary to check if any of the members have sent more messages than the max number of messages allowed. If a member has exceeded its limit, the sequencer would send out a traffic control message to tell that member to slow down. Otherwise, the sequencer would send out a traffic control message to tell the member that his message delivery rate is acceptable. Upon receiving a traffic control message, the members would update its message sending rate (enforced by asking the sending thread to sleep or not to sleep).

Implementaiton Testing:

To demonstrate the traffic control mechanism, we could print out the traffic control messages sent by the sequencer and received by the chat members, the log messages regarding the length of send delay interval on the sender side to the terminal. Also, the display of the broadcast messages from the sequencer could indicate the delivery rate change.

Code Related:

The code is in the folder named “dchat_congestion”.

Most of the three layers are the same as in the basic version. While I have made the following additions:

1. **Traffic-checking Thread** (dchat.cpp 25, 146; DNode.cpp 484-512)**:** while the sequencer is alive, it would periodically check the delivery rate of each member and send messages to notify members about whether the delivery rate shall be reduced or resumed (if previously reduced).
2. **Hashmap message_counter_table** (DNode.h 45-60, 92): the structure to hold the counter of received chat messages, which will be updated upon receiving new chat messages and will be reset after the sequencer has finished one check.
3. **Delievery Rate Adjustment** (Handler.cpp 78-86, DNode.cpp 279): force the chat-message-sender to sleep/not sleep for a fixed time interval before sending new chat message to the sequencer.

5.2 Fair Queuing

Note: To easily test the fair queueing function, we have to slow down the sequencer side by manually force it to sleep a fraction of one second before handling any messages. Otherwise the message handler could be too fast to show any difference and we would have to copy and paste lots of chat messages.

Implementaiton Details:

On the sequencer side, it will keep a hashmap of FIFO message queues with key being the address of each chat group member, and a counter indicating the index of member in the member list for Round Robin algorithm. Upon

receving a new message (other than join request or heartbeat) from members, the sequencer will push the message into the associated queue of the sender. Then it will check the message queues according to the order of member list, starting from where the counter is pointing to. Next the sequencer will perform the message handling tasks whenver it has found a queue that has messages not yet handled in it.

Implementaiton Testing:

To demonstrate the fair queueing, we slow down the server side handling rate (DNet.cpp 275). Then we can have two members A, B trying to flood messages (by copy and paste) to the sequencer at the same time, and A and B would alternate to deliver chat messages.

Code Related:

The code is in the folder named "dchat_fair".

Most of the implementation is the same as in the basic version. While I have made the following additions:

1. **Buffer Messages by Sender Address** (DNet.cpp 241-248): upon receiving a non-heartbeat, non-join-request message, the sequencer would buffer up the message and its address related information.
2. **Process via Round Robin** (DNet.cpp 274-315, Handler.cpp 12-41, Member.h 137-144): upon buffering a newly received message, the sequencer would try pop a message from the buffered queues to process. The sequencer would keep an integer to track where the last session of Round Robin finished at. For example, if the last process task poped a message from member A, the current task should try pop message starting from the member after A, say, member B and then C, D ,E, etc.
3. **Avoid the Sequencer Chat Privilege** (DNet.h 44-48, DNode.cpp 271, Handler.cpp 35-41): originally when the sequencer wants to send a chat message, it immediately broadcasts the message to every member. To ensure that the sequencer does not have such chat privilege, we make the sequencer to push to a buffered-message queue, and the chat message will later be processed via Round Robin..

5.3 Message Priority

Note: To test the priority function (heartbeat, join request, etc. are prioritized compare to chat messages) we slow down the sequencer side by manually force it to sleep 0.3 second before handling any messages. Otherwise it is too fast to tell the difference.

Implementaiton Details:

Message Priority is accomplished by assigning priority score according to the message types upon receiving new messages, and pushing the new messages into a priority queue before processing. Next, the message with the highest priority score is poped from the priority queue and hand to the handler to further process the message. For now, we assign a priority score of 0 to D_M_MSG and D_CHAT messages, a score of 9 to D_HEARTBEAT messages and election related messages D_ELECTION and D_COOR, and a score of 5 to other messages about message joining or leaving, etc. And these scores could be easily adjusted at DNet.cpp 282 (int DNet::findPriorty).

Implementaiton Testing:

The demonstration is a bit tricky because the handling is usually too fast to show the diffience of priorities. To demonstrate, we first slow the sequencer down, then flood the sequencer while have a new member join the chat group. By printing log messages to the console, we can see that even if the chat messages is received and pushed to the queue earlier than a join request/heartbeat message, the chat message would be poped and processed after the sequencer finish processing the more important messages. See below for a screenshot of priority demo.

Code Related:

The code is in the folder named "dchat_priority".

Most of the implementation is the same as in the basic version. While I have made the following additions:

1. **Priority Assignment and Prioritized Buffer** (DNet.cpp 239-303): upon receiving a new message, the message type would firstly be extracted and a priority score would be then assigned based on the type of the incoming message. Then the sequencer would buffer up the message and its address related information in a priority queue based on the priority score assigned by a mechanism described in Implementation Details section.

2. **Process by Priority Order** (DNet.cpp 260-280): In this process, the sequencer pops a message from the priority queue to process and send back asnwers (ACK or others) accordingly. This handling usually fast (even we slow the sequencer down manully), thus we could re-utilize the old socket.
3. **Message-processing Thread** (dchat.cpp 78-87, 147, 154, DNode.cpp 42): The queue is mainly used for demonstration, as it is easier to adjust how much we want to slow down the sequencer to display the priority mechanism. The implementation should be functional without the processing thread, but rather making the call within DNet.cpp.

```
pushed message: CHAT#20323#546#Bob:: f
slept: 300
poped message: CHAT#20323#546#Bob:: f
Bob:: f
pushed message: CHAT#20323#547#Bob:: g
pushed message: HEARTBEAT#158.130.24.208:20379
# priority : important
slept: 300
poped message: HEARTBEAT#158.130.24.208:20379
slept: 300
poped message: CHAT#20323#547#Bob:: g
Bob:: g
pushed message: CHAT#20323#548#Bob:: h
slept: 300
poped message: CHAT#20323#548#Bob:: h
Bob:: h
```

5.4 Encrption

Message is encrypted before being sent out in network layer. Similarly, each received message is decrypted after received in network layer. The encryption method we used is substitution encryption. The encryption key is agreed across all the users. The key is defined as ENCRYPTKEY in stdincludes.h.

5.5 Decentralized total ordering

Instead of using the centralized sequencer, use a decentralized total ordering algorithm, using the clients as participants. Elected leader is not used here. Processes collectively agree on the assignment of sequence numbers to messages in a distributed fashion. All-to-all heartbeat failure dector is implemented for the membership management. (Sources code in the <*decentralized_extra_credit*> folder)

All-to-All heartbeat failure dection

For simplity, we only implementated an all-to-all failure dector for detecting node failure

Decentrilized P2P Multicast Protocol

Each process maintain two variables:

int P = 0; // P(q,g) is largest sequence number proposed by q to group g:

std::pair<int, int> A; // A(q,g) is thelargest agreed sequence number q has observed so for for group g.

Step 1 (sender to all receivers): Process q multicast message m. Each received message is put in the hold back queue of the receiver and marked as undeliverable.

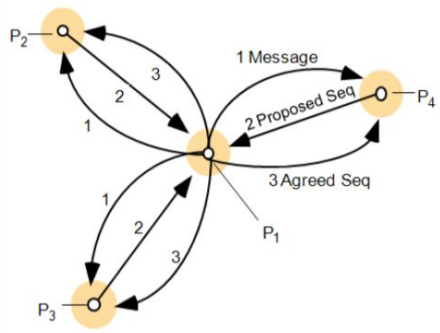
Step 2 (receiver back to sender): The receiver assigns a proposed timestamp (including its process id) to the message and returns to sender, which must be larger than any timestamp proposed or received by that process in the past. It is made unique by including process identifier as a suffix to the timestamp.

Step 3 (sender to all receivers): Sender chooses largest proposed timestamp as final timestamp for message and informs destinations. Receivers assign final timestamp to message in hold-back queue and mark message as deliverable. Hold-back queue is reordered in timestamp order and when the message at the head of the hold-back queue is deliverable, it is delivered.

Messages:

#define D_PROPOSE "PROPOSE" // PROPOSE#msg

```
#define D_COMMIT        "COMMIT"        // COMMIT#id#pid#key
#define D_JOINREQ       "JOINREQ"       // JOINREQ#PORT#Name
#define D_JOINLIST      "JOINLIST"      // JOINLIST#LEADERNAME#ip1:port1:name1:...
#define D_M_CHAT        "CHAT"          // CHAT#msg
#define D_M_ADDNODE     "ADDNODE"       // ADDNODE#ip#port#name, multicast from the sequencer
#define D_HEARTBEAT     "HEARTBEAT"     // HEARTBEAT#ip:port
```



Implementaiton Testing:

In order to validity our implementation, we print out the log files to the terminal for the two phase protocol. We print out the timestamps includeing the finalized maximum timstamp for each multicast request. To test it we have a special client which gives a higher sequence number and this message should be displayed with the higher sequence number.

5.6 Chandy/Lamport's 'snapshot' algorithm

Description:

Snapshot is initiated by typing "take snapshot" in command line from any active member. After receiving confirmation from user, the process which initiates snapshot will record its own current process state and multicast markers to all the other members in the chat group. It then starts recording any in-flight message until the completion of one snapshot.

Once a process i received a marker from process k:

(1) If this is the first time it sees a maker, it will record its current process state, set the state of the the channel between it and k (Cki) as empty and multicast markers to all the other processes. It then records all the in-flight messages from all other processes except k.

(2) If process i has already seen a marker from k', it will mark the state of Channel Ck'i as all the messages it receives from k' since it saw the marker for the first time.

Snapshot is regarded as completed once (1) all the processes have received a marker, and (2) all the incoming channels have received a marker (which means states of all the channels have been recorded).

Code is in a folder named "snapshot_extra_credit".

Class:

Snapshot.cpp Snapshot.h Channel

API:

Channel:

```
std::queue<std::string> messages;
```

```
void addMsg(std::string m);
std::string retrieveMsg();
void clearChannel();
bool isEmpty() ;
void copyChannelMsg(Channel *c);
int getMsgCnt();
```

Snapshot:

```
DNode *ssnode;
Member *ssmember;
int channelNum;
int channel_marker_cnt;
std::string marker_from_addr;
Channel *marker_from_channel;
std::deque<std::pair<time_t, std::string>> msg_queue;
std::unordered_map<std::string, bool> channel_markers;  // <address, received_marker> pair
std::unordered_map<std::string, Channel> channels;  // <address, channel> pair

Snapshot(std::string name, std::string addr, DNode* node);
void recordState(DNode* cur);
void recordChannelMarker(std::string from_addr);
bool getChannelMarkerReceived(std::string addrKey);
bool receivedAllMarkers();
Channel* getChannel(std::string addrKey);
void setMarkerFromAddr(std::string from_addr);
Channel* getMarkerFromChannel();
std::string getMarkerFormAddress();
int getChannelNum();
Member* getMember();
DNode* getNode();
std::unordered_map<std::string, Channel> getChannels();
void setChannels(std::unordered_map<std::string, Channel> c);
void setMsgQueue(std::deque<std::pair<time_t, std::string>> q);
std::deque<std::pair<time_t, std::string>> getMsgQueue();
```

Message format:

```
#define S_NONE 0       // not in taking snapshot
#define S_RECORDING 1   // after first time see marker, start recording
```

Workflow:

When user types “take snapshot” in command line, global snapshot is initiated following the logic in above desctiption part (line 44, dchat.cpp). After completion of snapshot, a thread will be created which is responsible for writing the checkpoint to disk for future reference (line 777, DNode.cpp). The file name is in the format of “Checkpoint_[address]_[snapshot_number].txt”. Meanwhile, the program only records the most recent snapshot. When user type “show snapshot” in command line, the most recent checkpoint will be displayed on screen (line 46, dchat.cpp). This snapshot information contains process state at the point of taking snapshot as well as all the in-flight messages.

Process state information contains leader election status, sequence number, leader name and address, gourp member list and time of last contact with the members. All the history messsages are also included in process state with timestamp associated with each message. In-flight messages are captured for each channel in every process’s snapshot.

Sample snapshot screen output:
Process state:
Election status: 0

Sequence seen: 0

Leader name: Yisi
Leader address: 158.130.24.202:20328

Member list:
Username: Bob, address: 158.130.24.208:20304, last contact time: 2016-04-24 18:36:03

Channels:
Address: 158.130.24.208:20304
Messages:

Message history:
2016-04-24 18:35:39:
    Start chatting. Current users:
Yisi 158.130.24.202:20328 (Leader)

2016-04-24 18:35:48:
    NOTICE Bob joined on 158.130.24.208:20304
2016-04-24 18:35:56:
    Yisi::blah
2016-04-24 18:35:57:
    Yisi::blah
2016-04-24 18:35:58:
    Yisi::blah

End of snapshot.

5.7 GUI
Develop a GUI using C++ Qt cross-platform graphic user interface library. The appearance of GUI is as follows. It works just as the command line version except for that the chat messsages are displayed on the panel as well as the membership list. User enters chat message and hit Enter button to send message.
The source code is in the folder <*GUI_extra_credit*>.


MainWindow
dchat
Join Address
Username
Connect
MainWindow
dchat
Bob::Hello Alice!
Alice:: Hi Bob! CIS 505 is the best!
192.168.73.130:20446:Bob (leader)
192.168.73.130:20046:Alice
Alice 192.168.73.130:20046
Enter