Design Document
Distributed Chat Client for Unix

Team Members

- Rohit Dureja (rohit@seas)
- Ritika Khandeparkar (ritikak@seas)
- Shanjit S Jajmann (sjajmann@seas)

Software Design:

Our design stresses on being distributed ensuring that the same code runs on all the clients using the chat system. To make this simple and easily operable, we identify the number of arguments entered by the user and decide whether or not they wish to start a new chat or join the old one.

We describe the following functions for our chat client to perform:

- Join chat:
    - As a sequencer: In this case, joining is simple since this client is the first to start the chat. We simply set a variable, which determines that this client is currently the leader/sequencer and have a thread which waits on incoming messages.
    - As a client: In this case, we establish a connection with the leader, and send a message stating our intent to join the chat. Also, a variable in reset to indicate that we are not the leader of this chat. We then proceed to start threads only to receive incoming messages and send messages typed in by the user. The client must now wait to receive an ACK from the sequencer which will also contain a message with the names and IP addresses of all current users.

- Send messages:
    - For both sequencer and client: We have kept this interface similar since it allows it to be less complicated. Also, since we are using a totally ordered multicast algorithm, this allows in ensuring that the ordering is kept correctly. This involves sending a message with our determined format to the sequencer.
    - We have also decided to use an acknowledgement system so that the sender receives an ACK from the sequencer for every message that they send. If they do not receive an ACK, they will resend the message. On the sequencer side we have designed mechanisms to ensure that even if it received the message the first time it does not re-display it.

- Ordering at the side of the sequencer:
  We display messages in the order that the sequencer receives them to ensure a global ordering scheme. In regard to this, the sequencer holds the messages to be sent out in a

list which is ordered in the time order in which it receives the messages at the socket. Since everyone else receives messages in this order, we ensure total ordering.

- Participation:
    - We have decided to have a “heartbeat” system to ensure that everyone is still present in the conversation. In our system, we have a thread on the sequencer side that will send out “heartbeat” messages every 5 seconds to check who is still present on the call.
    - This involves a sort of 2 Phase Commit Protocol, where the sequencer sends out a heartbeat the first time expecting an ACK from the client. If it does not receive an ACK within the next 2 seconds it resends the heartbeat message. At this point, if the client gets the message it must send the ACK. If the sequencer still does not receive an ACK it assumes the client has crashed or died and removes it from the list of clients currently on the chat. We realize that this is subject to the two generals problem, however we feel that any more than sending the heartbeat message twice would lead to too much unnecessary traffic on the network. We also have a scheme wherein, if we receive a message (not heartbeat but conversation message) from the client in this time, we also assume them to be active so the heartbeat is not resent and we do not wait for an ACK.

- Acknowledgements: There are two types of ACK’s in our system
    - ACK for heartbeat: This is sent by the client to acknowledge that they are alive and using the chat system actively.
    - ACK for message: This is sent by the sequencer to acknowledge that it has received the conversation message from the client. If the client does not receive this message, it resends the message to the sequencer.

- Leader election:
    - For leader election, we suggest a scheme as follows. If a client sends a message to the sequencer and does not receive an ACK in response, it will wait 2 seconds and resend the message. If it still does not receive an ACK it will assume that the leader is no longer active and call an election. For election we have decided to use the bully algorithm, electing the oldest client as the sequencer. We basically store the entire list of clients as a linked list going in order of the sequence in which they joined the chat. To elect a new leader, we use the client at the top of the list and select them to be the new leader. The client calling the election then sends a message to the new leader informing him of his new role in the system. The leader will then send a “heartbeat” message and the clients receiving the heartbeat message now realize that they must send their messages to this new leader.

Data Structures

As all of our implementation is using C++11, we have used the Standard Template Library extensively. We have employed data structures responsible for holding, storing and transferring most of our data. The main data structures we have employed so far are:

- the send message queue - this queue is used to store messages the user wants to send to the chat group.

```
// --- send message queue --- //
class message_information
{
      public:
      struct sockaddr_in address;
      char packet[BUFLEN];
};
queue<message_information> send_message_queue;
mutex send_message_queue_mtx;
```

- the recieve message queue - when a user node receives any message over the network it is added to this queue.

```
// --- receive message queue --- //
class message_information
{
      public:
      struct sockaddr_in address;
      char packet[BUFLEN];
};
queue<message_information> receive_message_queue;
mutex receive_message_queue_mtx;
```

- the active client list vector - this is the vector map which keeps track of all the active users in the group chat. As we have used a vector data structure, the node at location 0 is by default the leader. When the leader leaves, the underlying vector node; i.e at position 1 in the vector, is pushed up and made the new leader accroding to our leader election algorithm.

```
// --- list of nodes in the group --- //
class node_information
{
      public:
      struct sockaddr_in address; // stores socket information;
ip, port
```

```
        bool status; // active or inactive
};
vector<node_information> nodelist;
mutex nodelist_mtx;
```

- the display queue - we do not write messages on to stdout as they arrive as all messages are not supposed to be displayed to the user. Instead, we store them on a queue.

```
// --- display message queue --- //
class message_information
{
        public:
        char packet[BUFLEN];
};
queue<message_information> display_message_queue;
mutex display_message_queue_mtx;
```

All our data structures are protected with the use of mutexes which are employed using the C++11 mutex library.

Multithreading

We decided to create a multithreaded system using the threads library in C++11 since we felt this was the best way to ensure that we have our several processes running simultaneously. At present there are a total of five threads running:

- the send message thread which is responsible for reading messages from the send queue and sending them over the socket interface.
- the receive message thread which accepts messages from the socket interfaces and pushes them on to the received message queue.
- the parse received message thread which reads a message from the receive queue and parses it. Depending on the contents on the message the threads takes necessary actions.
- the display message thread which reads messages from the display queue and sends to stdout for display to the user.
- a user input thread which reads input from stdin.

As of now, these five threads runs at both the leader and client nodes. However, they differ in their functionality. We have used mutexes which ensure mutual exclusion when different threads are accessing the same data structure.

However, we still have to implement algorithms to eliminate busy waiting as some of our threads continuously query data structures for new data. We plan to use conditional variables with unique locks to eliminate busy waiting.

Underlying Protocol

We have a fairly simple underlying protocol that helps in achieving desired functionality. We generate an UDP packet that is sent across the network. As of now, the UDP packet has the structure

Control Sequence	Sequence Number	ACK Number	Payload
1 byte	1 bytes	1 bytes	256 bytes

The control sequence tells the node what action to perform. The sequence number and ACK number fields are used to implement reliable UDP and bear with packet loss. The payload is the data which is to be sent across node.

Control Sequence valid values:

- 10 - indicates new user with username as payload
- 11 - leader acknowledges acceptance of new user with IP address and port numbers of existing users as payload.
- 12 - if the new user had queried a non-leader to join, the non-leader replies with 12 and the leader IP address and port number in the payload.
- 20 - when a node posts a message to the chat group. This message can be from the non-leader to leader for sequencing or from the leader to group members for display.
- 30 - the node which detects that the leader has dropped, sends out a message with this control sequence with the ip and port of the new leader.

We still have to implement the reliable UDP protocol which makes use of sequence and ACK numbers.

Division of work

We have worked together to come up with the comprehensive design of the project. However, for the implementation we have decided to divide the workload based on specific components of the project. We regularly meet and collaborate to allow the design to work together.

Ritika (**Chat messages and ACK**): In charge of writing threads for sending and receiving of conversation messages. This includes ensuring the messages are sent and put into an ordered list at the side of the sequencer and that ACK's are sent out for every message received.

Shanjit (**Joining chat and maintenance messages**): In charge of the table containing IP addresses for every client that needs to be maintained in the order in which they join the chat (since that order is the basis of our leader election protocol). Also ensuring that join messages are handled correctly at the sequencer and the client receives a response with a list of the current clients on the chat.

Rohit (**Heartbeat messages and Leader Election**): Writing code to ensure that the heartbeat system works correctly and we check that a message is sent every 5 seconds to check participation of everyone on the chat system. Also, in charge of ensuring that leader election is called correctly and that there is seamless transition to the new leader.