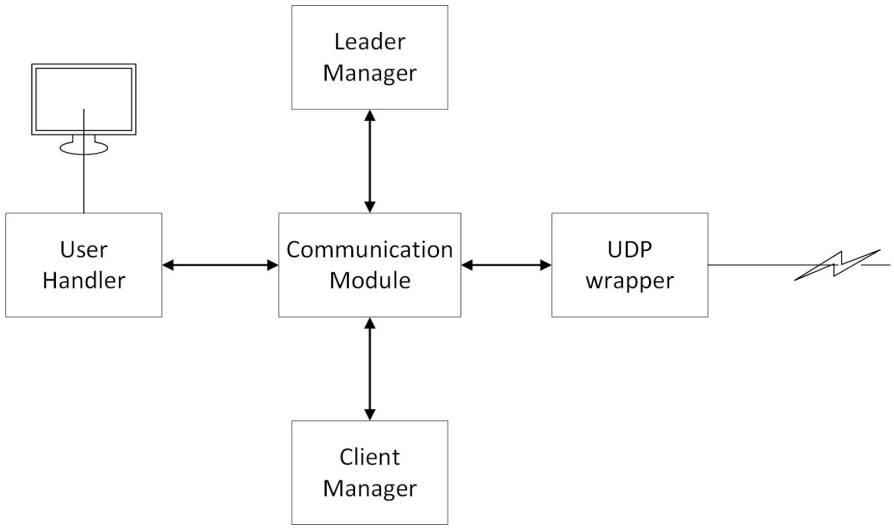
Project Milestone 1

Bipeen Acharya, Hung Nguyen and Krzysztof Jordan

Design Overall

We divide our distributed chat software into a number of components to handle different functionalities. Figure 1 illustrates them and how they communicate with one another in our software. The client manager is a very simple module which maintains a list of clients and any of their attributes. Similarly, the user input handler handles local user input and allows the client to send messages the server. The leader manager is responsible for ensuring that a leader is always present, and for holding an election (to determine a new leader) in case a leader leaves or crashes. The UDP Handler provides an abstraction for the internal implementation of UDP communications allowing other modules to have a black-box interface for sending and receiving messages. The communication module is the central component of our program that coordinates all communication between different modules/managers. This module ensures that information is exchanged between different modules in a timely manner.



Module Descriptions

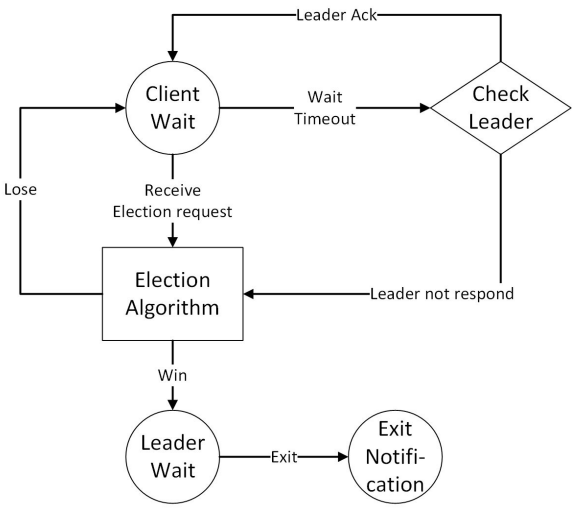
Now we describe each of the modules in our software. First, we provide a state machine to describe the different states a client in our software can be in. We then provide the API of each module.

Leader Manager

The leader manager module handles selecting a leader/sequencer amongst the clients. The clients normally remain in the **client wait** state and periodically check if the leader is active. If the leader does not respond, or if it receives an election request from some other client, the Bully Election Algorithm is run. A win for a client makes it a leader and it goes to the **leader wait** state.

Election Algorithm:

We plan to implement the Bully Algorithm to hold an election in case of a leader exit or leader failure. Our design requires clients to periodically query the Leader Manager to communicate with the leader. When a client detects a leader exit/failure, it broadcasts an election message to the rest of the clients in the client list (after querying the client management module). The IP addresses and the port of the clients are used to break ties. If during testing, two clients have the same IP address (which could happen since we might be running the clients on the same machine), we use the port number to break ties by allowing the client with the higher port number to be the leader.

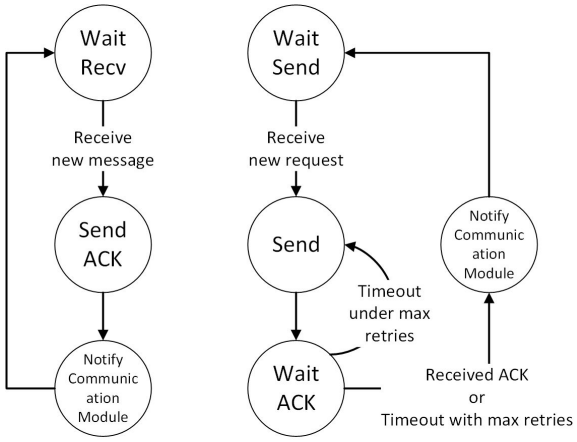


Leader Manager API

- [out] get_leader
- [out] is_leader

UDP Handler

The UDP handler will provide two operations to any other module. Firstly, it will receive any UDP message on a specific port, send back an acknowledgement of having received the message and pass along the message to the communication module (which will decide on the appropriate action to take on based on the message content.) It will also permit other modules to send messages to any other clients and state whether the send was successful. It will retry sending a message some amount of times before it times out.



Communication Module

The communication module is the central component of our program. It will coordinate all communications between the different modules and ensure that any information is distributed correctly. There are two main actions performed by the communication module, sending and receiving.

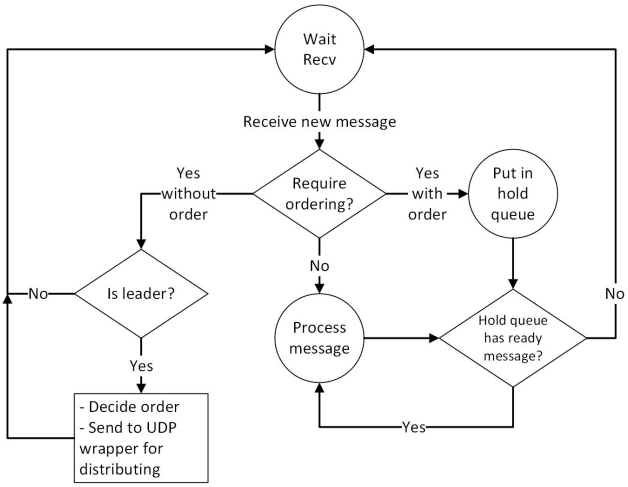
When a client is in the **wait recv** state and it receives a message, it decides whether an ordering of message is required. This decision is based on whether the messages are chat messages (which require ordering), or protocol messages like status messages or election messages (which do not require ordering). If the message does not require ordering, the message can be immediately be processed. If it does require ordering, then the client checks if it has already been ordered. If not, then we check whether this client is the leader (in which case it can decide the order and send for distribution), otherwise it can disregard the message

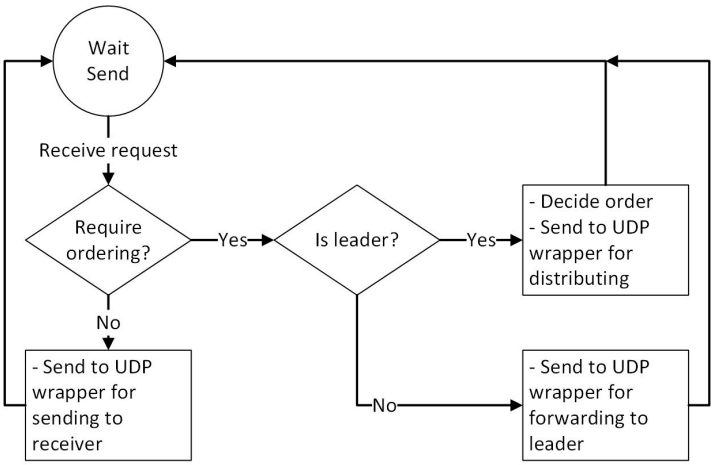
since the message was not intended for it. Otherwise if it is ordered, it is put in a hold queue and processed when the hold queue has a ready message.

On the other hand, when a client is in the **wait send** state**,** the module makes similar decisions for sending the messages.

Communication Module API

- [in] send_message
- [in] receive_message





Client Manager

The client manager is a wrapper for the client list data. It ensures sending status update to other clients through Communication module when join/leave chat group. In addition, it also updates and distributes the client list if receive join/leave notification from other clients.

Client Manager API

- [out] get_client_list
- [in] receive_notification

User Handler

The user handler will process input from the user in the form of chat messages and pass them along to the communication module. When the message is done being processed the communication module will send the message back for display. This layer will handle user prompts for exiting the program as well and make a best-effort attempt to properly close down the chat server by notifying the communications module.

Message Payload

The communication protocol consists different type of messages as described below:

1. Chat Message: for distributing user's chat messages within the system. This type of message requires total-ordering.

```
typedef struct _ChatMessage {
        int uid;
        string username;
        string data;
} ChatMessage;
```

2. Status Message: for distributing client's status within the system. It will be used whenever a client wants to notify its status or other clients' detected status. This type of message doesn't require total-ordering.

```
enum Status{CLIENT_JOIN, CLIENT_LEAVE, CLIENT_CRASH, LEADER_LEAVE};
typedef struct _StatusMessage {
        string username;
        Status status;
}
```

3. Election Message: for distributing message for election algorithm. This type of message doesn't require total-ordering.

```
enum ElectionCommand{ELECT_START, ELECT_STOP, ELECT_WIN};
typedef struct _ElectionMessage {
        ElectionCommand command;
} ElectionMessage;
```

4. Recover Message: for handling crash/error recovering including lost messages/leader crash, etc. This type of message doesn't require total-ordering.

```
enum RecoverCommand{MSG_LOST}
typedef struct _RecoverMessage {
        RecoverCommand command;
        int order;
}
```

The payload over the network is basically the union of all message types:

```
enum MessageType{CHAT_MSG, STATUS_MSG, ELECTION_MSG, RECOVER_MSG}
typedef struct _Payload {
        string ip;
        int port;
        int order;     // -1 for no total-ordering
        MessageType type;
        union message {
                ChatMessage chat;
                StatusMessage status;
                ElectionMessage election;
                RecoverMessage recover;
        }
} Payload;
```

Schedule

The schedule and tasks are managed at Trello.