CIS 505: Project 3
Team: Distributed Sisterz
Design Document

Program Flow (based on user-driven events):

- Event: Starting a new chat
    - Start listening as leader, initializing
- Event: Node *N* joining a chat
    - From *N*'s perspective, given a known member *M* of the group:
        - Loop:
            - Send *M* an ADD_ME message.
            - Wait until *M* replies:
                - If the reply is a PARTICIPANT_UPDATE message, the join was successful.
                - If the reply is a JOIN_FAILURE message, wait a short period
                - If the reply is a LEADER_ID message:
                    - Send the leader an ADD_ME message
                    - If the reply is a PARTICIPANT_UPDATE message, the join was successful.
                    - If the reply is a JOIN_FAILURE or LEADER_ID message, wait a short period
                    - If the reply is a JOIN_NICKNAME_FAILURE, exit and prompt the user to choose a new nickname
            - After 10 failed attempts, exit program
    - When a member of the group *M* is contacted by a new node *N*:
        - If M is leader:
            - If *M*'s name is in use, send JOIN_NICKNAME_FAILURE message
            - Otherwise, *M* adds *N* to list
            - *M* multicasts new list of nodes to network, including *N* (PARTICIPANT_UPDATE message)
        - If *M* is not leader:
            - *M* sends "Leader is L" message to *N*
        - If *M* is in election:
            - *M* sends *N* JOIN_FAILURE
        - Wait for an announcement from the leader that *N* has been added
        - Reply to *N* with the location of the leader
        - If in leader election or request to leader triggers election, reply to *N* with JOIN_FAILURE message
- Event: Exchanging messages
    - When a node *N* wants to send a message *m*:
        - *N* sends the leader the message *m*
        - If a leader election begins, place *m* in a send queue

    - When the leader receives a message *m*:
        - It assigns the message its current (logical) clock value *t* and increments the clock
        - For each member of the group:
            - Send the member the tuple *<m, t>*
    - When a non-leader *A* receives a message from the leader *L*
        - Print it
    - When a node receives a START_ELECTION message from *N*:
        - Compare own nickname to that of *N*
        - If own name is higher:
            - Reply to *N* with an ELECTION_STOP message.
            - Follow procedure for starting a new election.
- Event: A node *N* leaves the chat
    - From *N*'s perspective:
        - Just exit.
    - If *N* is not a leader:
        - From the leader's Perspective:
            - Tries to send a message to *N*
            - Message times out
            - Leaders announces *N* has left (with a PARTICIPANT_UPDATE message)
        - From everyone else's perspective:
            - Gets PARTICIPANT_UPDATE message and updates local state
    - If *N* is a leader:
        - A node *M* will try to send *N* a message *m*
        - Message times out
        - *M* broadcasts a START_ELECTION message to the group
        - If *M* receives an ELECTION_STOP message from a peer, wait until an ELECTION_VICTORY message is received and re-send the message *m* to the new leader. *(M->leader)*
        - If *M* receives an ELECTION_STOP message but times out waiting for an ELECTION_VICTORY, resend the START_ELECTION message.
        - If nobody replies to the START_ELECTION message after a timeout, broadcast an ELECTION_VICTORY message.
- Heartbeats?

Node State (For a node *M*):

- Nickname *(M->nickname)*
- IP address *(M->ip_address)*
- Port # *(M->port_num)*
- Logical clock (sequence number for each message) *(M->seq_num)*
- List of other nodes *(M->participants)*

- Is Leader *(M->is_leader)*
- Reference to leader *(M->leader)*
- Message queue *(M->msg_queue)*

Message Types:

- PARTICIPANT_UPDATE:
    - Contains a list of rows of the following form:
        - <nickname, IP address, Port #, is_leader>

Transport Layer: Reliable Message Protocol:

- A datagram-oriented protocol that guarantees that messages will be delivered exactly once.
- `int RMP_get_address_for(const char *address, int port, rmp_address *address_dst);`
    - Parses the given IP `address` and `port` number into the rmp_address in *`address_dst`.
    - Returns 0 on success and -1 on error.
- `int RMP_sendTo(rmp_address address, const void *buffer, int numBytes);`
    - Attempts to sends `numBytes` bytes of the specified `buffer` to the given `address`.
    - Will send up to 65,502 bytes of data at a time, all remaining bytes will be truncated.
    - Will block and retry sending until the recipient acknowledges the message OR at least the timeout value of 100ms is reached.
    - Returns the number of bytes sent or -1 on error.
- `int RMP_setup_receiving_socket(rmp_address address);`
    - Sets up an RMP socket that can be listened on for incoming connections on the given `address`.
    - Returns a socket file descriptor or -1 on error.
- `int RMP_listen(int socket_fd, void *buffer, size_t len, rmp_address *src_addr);`
    - Blocks for a message to be received on `socket_fd` and places up to `len` bytes of it in the given `buffer`. Also places the sender's address in `*src_addr`.
    - Returns the number of bytes received or -1 on error.
- Implemented over UDP with the following packet format:
    - ```
      | Type   | ID number | Data |
      | 1 byte | 4 bytes   | ...  |
      ```
    - Type is one of `DATA`, `SYN_ACK`, or `ACK`
- When a node A wishes to send a message to another node B, the following protocol is used:
      ```

- When A wants to send B a message, it first constructs a `DATA` packet with the desired message and a random ID number. It sends the message to B and stores a duplicate in a send buffer.
- When B receives a `DATA` packet from A, it stores the ID number in a receive buffer for A and replies to A with an `ACK` packet containing the ID.
- When A receives an `ACK` packet from B, it replies with a `SYN_ACK` packet with the same ID number. If the ID number matches the one in its send buffer, it removes it from the buffer and becomes ready to send again.
- When B receives a `SYN_ACK` packet from A, it removes the corresponding ID from its receive buffer.
- If any item remains in a send or receive buffer for more than the time (TIMEOUT/NUM_RETRIES), its corresponding message is retransmitted. After it has been retransmitted NUM_RETRIES number of times, it is removed.