BGN: 2191A

P5

Q5

a. The nodes in the system are the clients (users) and the 5 servers. Each node stores a Lamport timestamp which can be set using “setTimestamp” and retrieved using “getTimestamp”.

   Client nodes can send 3 types of message: follow, unfollow, and listFollowers, each containing a user to reference, the user making the request (which we assume can be authenticated), and a timestamp. They can also contain a callback but this is only used once a response has been received – the callback is not transmitted.

   Server nodes (if they are running) will send acks in response to all client messages. In the case of listFollowers, wherein a response is required, the ack contains the response.

   The algorithm on the client side maintains a message queue with earlier messages at the front. Each message in the queue is sent to all 5 servers. If less than 3 acks are received, then the message is replaced at the **front** of the queue. This is so that a later message cannot be successfully sent before a retry of an earlier message.

   If enough acks are received and the message requires a response (i.e., listFollowers), then the client checks that at least 3 responses are the same as one another. If not, the message is again replaced at the front of the queue. If 3 responses are indeed the same, this “consensus” response is passed to the message callback.

   All of the messages are idempotent so retrying them has no additional side-effects.

   I am assuming that the servers have a way of synchronising themselves over a long enough period of time (e.g., once per day).

   I am also assuming that all acks contain information which can identify the message they are acknowledging, thereby allowing the client to check whether an ack was received for a specific message.

```
// Repeatedly tries to send each idempotent message m in a queue to all 5 servers until it receives an ack
from at least 3
func sendMessagesToServers() {

          while (message_queue not empty) {
                    // Get the earliest message in the queue
                    m = message_queue.pop_first();

                    // Send message to all servers
```

```
                    T = getTimestamp();
                    for (i from 1 to 5 inclusive) {
                              send(server_i, m, T + i);
                    }
                    T = T + 5

                    // Wait for acks
                    sleep(100ms);

                    // Collect acks
                    // We assume that any response data (e.g., the list of followers) will be contained
within the ack
                    acks = [];
                    for (i from 1 to 5 inclusive) {
                              did_get_ack, ack, ack_T = receieveAck(server_i, m);
                              if (did_get_ack) {
                                        T = max(T, ack_T);
                                        acks.append(ack);
                              }
                    }

                    setTimestamp(T + 1);

                    // If not enough acks, retry message IMMEDIATELY
                    // (i.e., put it at the front of the queue not the back)
                    // This ensures that e.g. an unfollow message does not get run before a retrial of a
follow message which came earlier
                    if (num_acks < 3) {
                              message_queue.put_in_front(m);
                              sendMessagesToServers();
                    }

                    // If the message requires a response, check whether any 3 of the acks are the same
                    // (e.g., if they contain the same lists of followers)
                    if (m.callback is not null) {
                              ack = consensus(acks);                    // null if no 3 acks are the
same
                              if (ack is null) {
                                        // Retry message IMMEDIATELY
                                        message_queue.put_in_front(m);
                                        sendMessagesToServers();
                              } else {
                                        message.callback(m);
                              }
                    }
}

func follow(u2) {
          m = generateFollowMessage(u2);
          message_queue.enqueue(m);
          sendMessagesToServers();
}

func unfollow(u2) {
          m = generateUnfollowMessage(u2);
          message_queue.enqueue(m);
          sendMessagesToServers();
}

func listFollowers(u2, callback) {
          m = generateListFollowesMessage(u2);
          m.callback = callback;
          message_queue.enqueue(m);
          sendMessagesToServers();
}
```

b. This pseudocode works by listing all of the current user's followers and sending a "chirp" message to the server for each of them. This message includes the recipient user and the body of the chirp. This body encodes (perhaps with JSON) information such as the time of day of the sending of the chirp, the text of the chirp, potentially references to another chirp if this is a reply, etc. Like the other messages, the chirp message is assigned a Lamport timestamp when sent.

   Since the chirp message is no longer semantically idempotent, I am assuming that out-of-order messages are rejected on the server-side with no ack sent. This prevents chirps being sent more than once if the ack takes too long to be received.

   I am also assuming that when a chirp is received by the client, its Lamport timestamp is updated. This ensures that replies are always delivered after the message to which they are replying.

   Finally, I am again assuming that each client knows their own id ("me" in the pseudocode)

```
func postChirp(body) {
        listFollowers(me, (followers) => {

                for each (follower in followers) {
                        m = generateChirpMessage(follower, body);
                        message_queue.enqueue(m);
                }

                sendMessagesToServer();

        });
}
```