CIS 505: Project 3 April 24, 2016

Team Distributed Sisterz:
Matt Howard (mhowar)
Jae-Joon Lee (jjlee)
Doron Shapiro (dorons)

Final Report

Reliable Messaging Protocol Analysis

All `dchat` data is sent over a reliable messaging protocol (RMP) on the network. RMP is a reliable, ordered, datagram-oriented protocol that works as follows:

- When a node `A` wants to send a node `B` a message `m`, it first constructs a `DATA` packet with the desired message and a random ID number `t`. It sends `DATA<m,t>` to `B` and stores a duplicate in a send buffer.
- When `B` receives a `DATA<m,t>` packet from `A`, it stores it in a receive buffer for `A` and replies to `A` with an `ACK<t>` packet.
- When `A` receives an `ACK<t>` packet from `B`, it replies with a `SYN_ACK<t>` packet. If the ID number `t` matches the message in its send buffer, it removes the message from the buffer and becomes ready to send again.
- When `B` receives a `SYN_ACK<t>` 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 `ACK_TIMEOUT`, its corresponding message is retransmitted. After it has been retransmitted `NUM_RETRIES` number of times, it is discarded.

We can analyze what happens when each of the messages in the protocol is dropped or duplicated:

- When a `DATA<m,t>` packet from node `A` is dropped, `A` will time out on receiving an `ACK<t>` packet and retransmit up to `NUM_RETRIES` times. If an `ACK<t>` is subsequently received, the protocol will continue as planned. Otherwise, `A` will correctly report send failure.
- When a `DATA<m,t>` packet from node `A` is received twice by node `B`, it will respond to both messages with an `ACK<t>` packet. Node `A` will respond to both `ACK`'s with `SYN_ACK`'s. However, upon `B`'s receipt of the first `SYN_ACK` packet, it clears its receive buffer, so the second `SYN_ACK` is discarded. Note that receiving a duplicate message restarts all related timeout timers.
- When an `ACK` packet from node `B` to node `A` is dropped, one or both of the following may occur:
    - Node `A` will time out on receiving an `ACK` packet and retransmit the original message, reducing to the case where a `DATA` packet is duplicated.

    - Node `B` will time out on receiving a `SYN_ACK` packet and retransmit the `ACK` up to `NUM_RETRIES` times. If a `SYN_ACK` is subsequently received, the protocol will continue as planned. Otherwise, the connection between `A` and `B` is down, `A` will properly report send failure, and `B` will properly discard the partially sent message.
- When an `ACK` packet from node `B` to `A` is duplicated, this reduces to the case where a `DATA` packet is duplicated.
- When a `SYN_ACK` packet from node `A` to `B` is dropped, `B` will time out and retransmit its `ACK` packet up to `NUM_RETRIES` times. If a `SYN_ACK` is subsequently received, it will return the original message to the user. Otherwise, it will correctly discard the message. In this case, however, node `A` will have incorrectly reported that the message was sent.
- When a `SYN_ACK` packet from node `A` to `B` is duplicated, this reduces to the case where a `DATA` packet is duplicated.

Chat Protocol Description

Here are all the possible commands that the chat program could send or receive on top of the RMP protocol.

Joining commands

- `ADD_ME <NICKNAME>`
    - Sent to request membership in a group
- `PARTICIPANT_UPDATE @<LEADER_NICKNAME>:<LEADER_IP>:<PORT> [<A_NICKNAME>:<A_IP>:<A_PORT> ...]= <JOIN_LEAVE_MESSAGE>`
    - Sent to all chat clients after a change in participants. The `JOIN_LEAVE_MESSAGE` will be of the form "Alice joined on 192.168.5.81:1923" or "Eve left the chat or crashed" or “Ira is the new leader”
- `JOIN_FAILURE`
    - Joining user should retry after short time period
- `LEADER_ID <LEADER_IP>:<PORT>`
    - Joining user should contact the specified leader
- `JOIN_NICKNAME_FAILURE`
    - Group addition failed, user must pick a new nickname

Message exchange commands

- `MESSAGE_REQUEST <SENDER_NICKNAME>= <MESSAGE_PAYLOAD>`
    - Sent to the leader to request that a message be broadcast to all participants
- `MESSAGE_BROADCAST <MESSAGE_ID> <SENDER_NICKNAME>= <MESSAGE_PAYLOAD>`
    - Sent by the leader to all participants with a message payload
    - Where `MESSAGE_ID` is an non-negative integer and comes from the leader's logical clock

Election commands

- `START_ELECTION <SENDER_NICKNAME>`
    - When a non-leader realizes a leader cannot be reached, it broadcasts this out
- `ELECTION_STOP`
    - When a node sees that an election has been started by an "inferior" node
- `ELECTION_VICTORY <SENDER_NICKNAME>`
    - Sent when a user believes themselves to be the new leader

Heartbeats

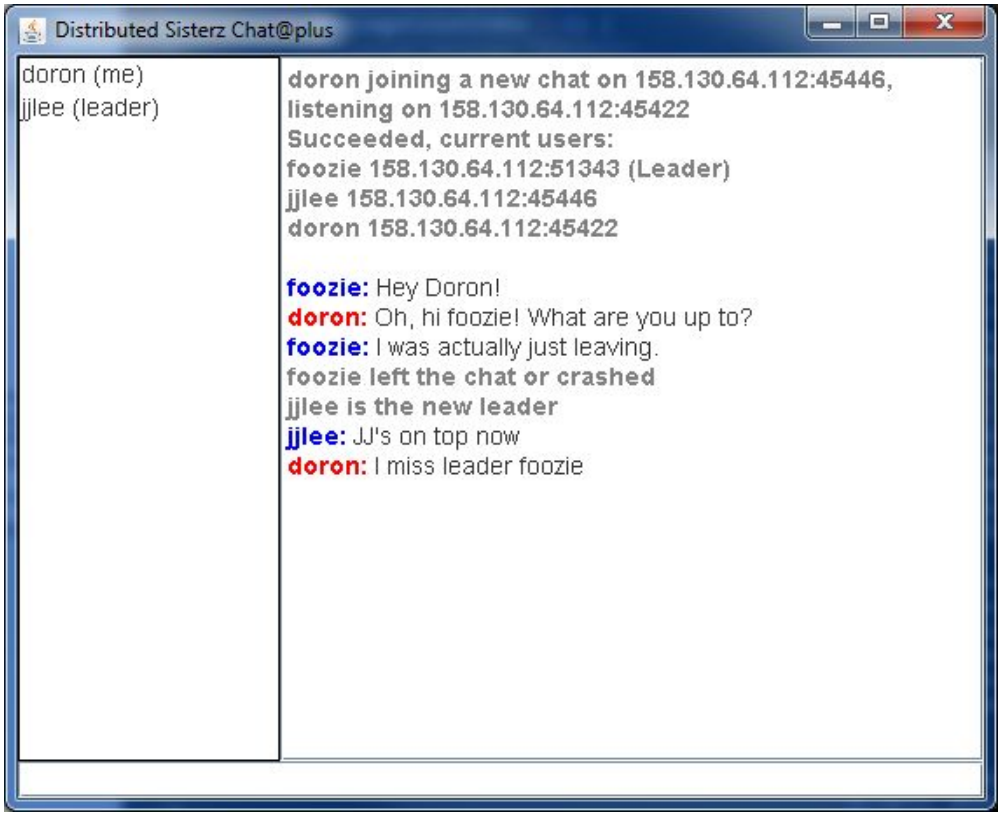
- `HEARTBEAT`
    - Sent from all participants to the leader on a regular interval to detect leader failure
    - Sent from the leader to all participants on a regular interval to detect participant failure

Assumptions

We restrict usernames to be less than or equal to 20 characters and messages to be less than 1000 characters.

GUI Extra Credit

Our graphical user interface consists of a Java Swing application. The Swing application consists of a chat history window, a list of participants, and a message entry box. The Java application acts as a wrapper for the CLI dchat program and interprets dchat's standard output to create a rich user experience. We use the Java ProcessBuilder to exec a dchat process and use multiple threads to process IO events.



Screenshot of our GUI

The GUI can be built and run inside the GUI directory using `make build` and `make run`. We were able to run it on speclab using X11 forwarding.

Encryption Extra Credit

We implemented a Vigenère cipher to provide encryption for our chat messages. To aid in debugging, the cipher is currently used only to encrypt user message data, although expanding this to encrypt metadata as well is trivial.

Building Running Distributed Sisterz dchat

Inside of the main directory, run `make` to build. Then, start the chat using the ./`dchat` binary. The usage is:

`dchat <NICKNAME> [<ADDR:PORT>]`

The `<ADDR:PORT>` argument is only used for joining an existing chat.