ECE419H1S – Lab 3 Design Document

Mark Sutherland – 997332989 *Scott Whitty – 997584378*

March 24, 2014

Overall Description

Our design for the decentralized Mazewar implementation is founded on the Token Ring algorithm and summarized in Figure 1.

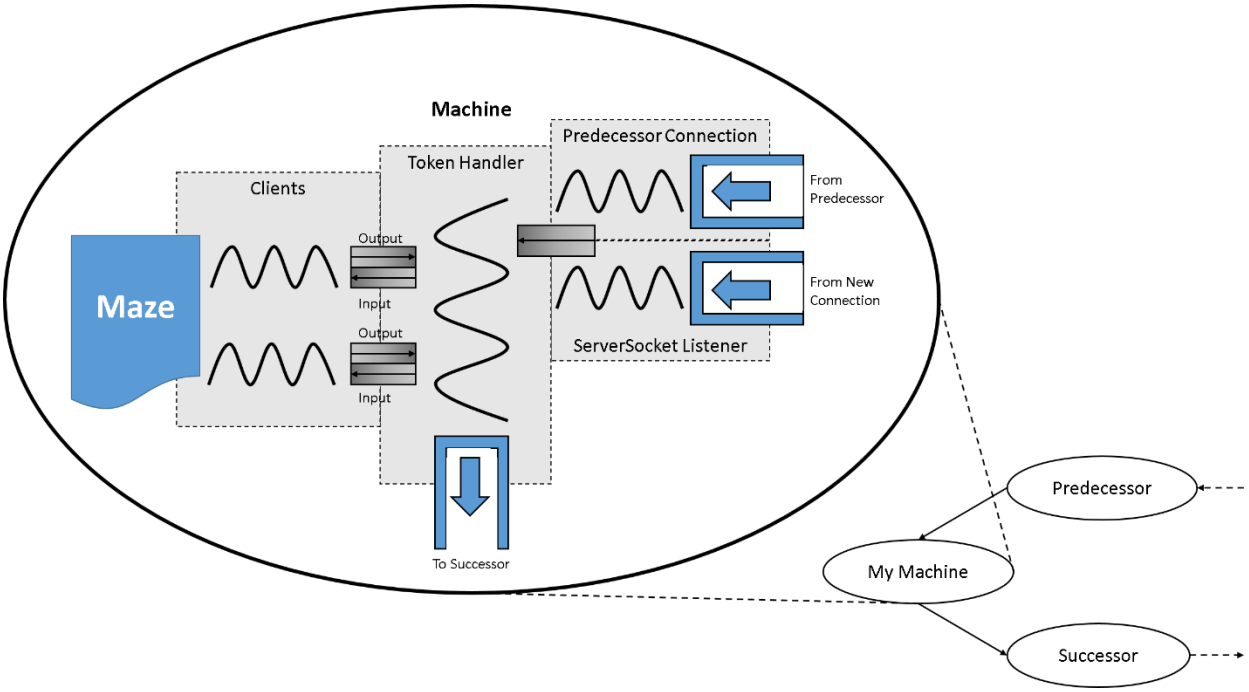

Figure 1: Overall Architecture for the Decentralized Mazewar Implementation

The key features of this design are as follows:

- **Client Threads:** Each Client (Remote, GUI and Robot) is hosted on an independent thread. They interact with the Maze via synchronized methods, and they interact with the Token Handler Thread via the input/output buffers which are allocated to them when they join.
    - When a Local Client (either GUI or Robot) generates an event, it is put in its output buffer and it immediately sleeps on its input buffer until the Token Handler pulls the action out of the token. This means each Local Client may have *only one event in flight at a time*.
    - Remote Clients sleep on their input buffers, process events from the Token Handler, and then immediately sleep on their inputs again.
    - Client threads are de-allocated and stopped upon a Leave event from the Token Handler, with the exception of a GUI Client which will wait until the Token Handler signals it again since a GUI Client leaving also constitutes closing the

process and the Dynamic Leave protocol must be executed before the process can be killed (see next section).

- **Token Handler:** The main thread for handling network communication. Given a token from the Predecessor Connection, it will pop off all events from the Token's queue in order and send them to its Clients. If the event corresponds to a Remote Client, the event simply goes back onto the queue. If the event was for a Local Client, the Token Handler checks that Client's output buffer for an event and if an event is present, it puts that event in the token instead. If no event has been requested by a Local Client when the Token Handler checks its output buffer, a NOP event is inserted into the token.
  Once all events in the Token have been processed, it is passed to the Machine's successor in the network.
  The Token Handler is also responsible for the dynamic join/leave protocol, as detailed in the next section. During this, it is fed a new socket by the ServerSocket listener.

Dynamic Join/Leave Protocol

Our solution implements a dynamic join/leave protocol with respect to the Token ring. The following shared policies are enforced during both protocols:

- Whenever a new connection is opened to a machine A, the machine B will send a special packet indicating one of two scenarios:
  - B wants to become A's successor in the ring
  - B is A's successor and is leaving the ring
- The Token has a special "Predecessor Location" field which, whenever non-NULL, means that the recipient must switch its predecessor connection to a new socket to that location and clear the location in the Token

Both the Join and Leave protocol have corner-cases related to the Ring network containing a single node; the details of these cases are omitted here.

Join Protocol

If machine **C** wants to join the ring, and assuming machines **A** and **B** are already in it and **B** is **A**'s successor, the following steps are executed:

1. **C** contacts the DNS server and chooses **A** as its predecessor.
2. **C** initiates a connection with **A** and sends it the port its SeverSocket Listener is listening on.
3. When **A** next receives the token, it adds the port that **C** is listening on to the Token, sends the token to **B**, and replaces its successor connection with the one **C** initiated.
4. When **B** receives the token and recognizes **C**'s location, it replaces its predecessor with a new connection to **C**.
5. **C** sees the connection **B** initiates and sets its successor to **B**.
6. When **A** next receives the Token, it will append the Locations of all Clients to the Token before processing any events.

7. When **C** receives the Token next, it will create Remote Clients for each location **A** appended to the Token, and adds a "Client Join" event to the Token for all its Local Clients.

When a machine's Token Handler sees a "Client Join" event for a Remote Client, it will allocate a new Client thread and an input/output buffer for it.

Leave Protocol

If machine **C** has predecessor **A** and successor **B**, and its GUIClient wants to leave, the following steps are executed:

1. When **C** receives the Token next, it places a "Client Leave" event in the Token for each Local Client (since Robots never generate a Leave event).
2. When a Token Handler Thread receives a "Client Leave" for a Remote Client, it removes it from the Maze and de-allocates its associated objects.
3. When **C** receives the Token back again, it notifies **A** that it is leaving; **A** will not pass the Token if it is received before **B** is connected to it.
4. **C** places **A**'s location into the Token and passes it to **B**. **C** is now free to de-allocate its memory and de-render the Maze.
5. When **B** receives the Token with **A**'s location, it opens a new connection to **A** and becomes **A**'s successor in the Ring.

Robot Clients

Since the Dynamic Join protocol is de-coupled from a new Client joining the Maze, support for Robot Clients becomes trivial. Upon a GUI Client detecting an "R" key press, it will attempt to spawn a Robot Client using a hard-coded list of possible names. This uses the same "Client Join" packet as seen in step 7 of the Join protocol.

Further Discussion & Questions

Q: What are the strengths and weaknesses of the join/drop protocol?

- The strength of our startup protocol is that it doesn't have one centralized "entry point" to the token ring that always must be up and responding. Any client that is inside the ring can support a join request and create the new sockets required for connection.
- However, this client must be up and in the game for the entire duration of the join protocol, if it tries to leave or disconnect before the entire join procedure has finished, then the ring will be broken and the game will hang.
- Additionally, multiple joins occurring simultaneously is not currently supported, although it could easily work for certain cases with large token rings. (i.e. Two different clients are supporting two join requests that are far enough apart in the token ring that they do not share any connections that would need to be changed concurrently.)

Q: Evaluate your design's performance on the current platform.

- This design works well for small numbers of clients in a high speed ethernet LAN, since actions can only enter the network of players when we first get the token. Since the token is passed around regardless of whether each connected player has a message available, the number of

messages per action is technically amortized over the amount of "available" messages in the network. For example: (N is number of clients)

- If only one client in the ring has a new action, the number of messages to render this action is O(N), since the token must go all the way around the ring.
- Best case is that each one of the N clients has a new message when they get the token, so the number of messages is O(1).
- In the case where every client is idling, the token will pass around at a constant rate, and no new client actions will be taken. In this case there is significant network overhead for no events being drawn on the screen!

Q: How does your implementation scale with the number of players? With high-latency networks? With thin clients?

- However, for a small # of players in on a high speed network, there is not a noticeable delay that we were able to notice when playing the game.
- For large numbers of players (assuming that they are all active and generating events), the token ring solution generates less messages on average than a totally ordered multicast (TOM) protocol, since TOM will generate N^2 messages for each action taken! However, although there are less messages overall, there will likely begin to be significant latency between a player pressing keys and events actually being rendered, since the network delays are additive over the entire token ring.
- This means that the token ring algorithm is preferable on a high-packet loss and low bandwidth network (assuming we support TCP retransmission so we do not lose the token), since the O(n) upper message bound is preferable in cases we want to minimize retransmissions.
- With thin clients in the mix, our protocol becomes less optimal since it is heavily based on multi-threading, and a thin client will slow down significantly when it has to perform tasks with lots of network messaging and multithreading.

Q: Discuss any inconsistencies that may occur. How does your design handle them?

- There should be no game inconsistencies with player locations or scores, since the token ring enforces the exact same event ordering at every node in the ring. However, there are occasional instances where the application hangs due to a bullet rendering issue (the same one that was present in Lab 2). The thread that renders the bullets is completely separate from the threads that implement the client actions and work with the token, and there occasionally arises a deadlock where it does not render the bullets properly.