ECE419 Design Document

Zexuan Wang (998851773)
Tzu-Yin Tai (999593319)
Suya Liu (998147964)
Gui Ming Tang (998229331)
Yiming Kang (998676730)
TianYu Li (999107149)

In this document, we will describe our fully consistent implementation of a peer-to-peer Mazewar game. The implementation is designed for the high network bandwidth, very low parallelism in network, very low network latency, and low-end nodes scenario.

1. The Distributed Algorithm

The team has decided on a Token Ring Design for our overall algorithm. The nodes (clients) are connected in a circle, where each node can only communicate with its neighbouring nodes in order to reduce network load. The players will take turns placing an event onto the Token, which is passed around the ring until everyone has received the event, then next event can be placed onto the token. To ensure consistency, only one event will take place per complete ring traversal.

2. Communication Protocol

In our design, there are two types of packets that are used to communicate between nodes: Token Packet and ACK Packet.

The Token Packet contains a Token Sequence Number and an Event (if any). The Token Sequence Number stays the same while the Event traverses around the ring, until everyone has received and executed that Event. The last player to receive the Token, who is also the original sender of that Event, will increment Token Sequence Number by 1, empty the event, and send the Token to the next player. When an empty Token is received, a new Event can be added.

When a node receives the Token, it will send an ACK Packet to its previous node to notify the receival of the Token. If previous node doesn't receive an ACK Packet within a determined amount of time, timeout happens and another copy of the same Token will be sent out. Each client keeps track of the Token Sequence Number, so they can to ignore any duplicate Token Packets.

See the following pseudo-code for the handling of dropped packets as well as the above process:

a) Node that receives Token

```
if Token.seq_num < self.seq_number:
    drop Token
else:
    send ACK with Token.seq_number back to previous node
    if Token.Event == empty:
        if self.event_queue != empty:
            attach next event in self.event_queue to Token
    else if Token.Event.player_name == self.player_name:
        Token.sequence_number ++
        remove Token.Event
    send Token to next client
```

b) Sender of Token

```
while True:
    wait for ACK
    if timeout:
            send same Token to next node again
        if receive new Token previous node or ACK from next node:
            break
```

3. Locate, Join, and Leave a Mazewar Game

Initial Start of the game: In addition to a Token Ring, a Naming Server will also be implemented. All players first connect to the Naming Server, sending their name, IP and a ServerSocket port number that can be used by a player to connect to them. The Naming Server will first wait for N players to join the game (N is a predefined integer), then it broadcasts list of all members in the ring to each client. The client locate itself in the list and the next client in the list is its next client. The client reply the name server with ACK and wait for the messages from the previous client or the name server. Once the name server receives ACK from all clients, it randomly selects a client and sends a start packet. That client will begin sending a start message down the token ring to initialize all clients and when the start message returns, the ring is successfully connected. If there is missing ACK(s) after the name server broadcasts the client list, the name server will remove the dead clients and broadcast again until there is no missing ACK(s).

Joining an existing game: To join an existing game, a new player C first contacts the Naming Server with its name and connection information, then the Naming Server selects a place in the ring for it to enter - say after player A and before player B, inserts C into the client lists and send C the client lists. Once client C receives the client list, it sends a join command along with information about itself and its location in the client list to the next client in the list. When the next client receives a join command, it adds the joining client to its client list according to the adding location. It then passes down the join command. When a client processing a join command sends to the joining client, it attaches the entire game state with the command, hence initializing the joining client. For any client processing regular commands, it always sends the command according to the local client lists at the moment. This allows regular commands to still occur in the system when a client is joining. If the regular command happens before the join command, the new client will be updated with the most up to date game state. If the join command happens before a regular command, the new client is updated with the game state, then the regular command will reach the new client which updates the game state like other clients.

Leaving the game:

To leave a game, the client sends a leave request to the name server who will delete the client from its list. Once acknowledged, the client will send a leave a request to the next client. When a client receives a leave request, it deletes the leaving client from its local client list, and propagate the command. The subtle difference is that the client will propagate the leaving message according to the unmodified client list but use the modified list for subsequent regular messages. This ensures that the leaving client will receive the message it sent which entails all client has updated the client lists and can leave safely. Before the leaving message receives its own leaving command, it is still in the game and is processing all commands regularly.

4. Missile Handling

To ensure consistency of missile tick, we will be handling missile tick just like other events (moving or firing). Each node will be responsible for sending out a missile tick event (attached on the Token just like other events) for a period of time, and the responsibility of sending missile tick will be assigned to each node round-robin-ly to ensure decentralized implementation. If a node is responsible for sending out a missile tick event, then it will generate a missile tick event every 200ms and put it into its local queue. The missile tick event will then just behave like other events, hence ensuring consistency among all nodes.

5. Handling Node Failure

To handle node failure, we add a Heartbeat Protocol. At regular intervals, a Heartbeat packet is sent to the neighbouring players. If a player notices that it haven't received a Heartbeat from its neighbouring player for a certain amount of time, timeout happens and it notifies the naming server. If a failed client is detected, the client first notify the name server to delete the failed client. Then it removes the failed client from its local client lists. After that it sends a remove client command with the failed client attached to the next client in the list. Each client when receiving this message simply removes the failed client from the client list and propagate the message according to the most update client list.

On each message sent, if the client does not receive ACK from the next client, it indicates the client has failed. The sending client then notify the name server and deletes the failed client from the local client list. It then sends a remove client command with the failed client attached to the next client in the list. Everything else happens like described above. The removed client command can also occur concurrently with regular commands, other remove client commands, join and leave commands. Hence the client will propagate any commands without waiting for the command it sent to come back.

6. RobotClient

We also plan to implement RobotClient. They are treated exactly like regular clients, except that their events are automatically generated.

7. Evaluate the portion of your design that deals with starting, maintaining, and exiting a game – what are its strengths and weaknesses?

In our solution, the basic method of maintaining a ring of clients consistently is to use a local client list at each client. Each client will propagate commands always according to its local client list and who is the next member on the list. For each ring event throughout the game session: join game, leave game, and detected a failed client, the most up to date client list is located at the client where the event occurred or one client downstream. And since the messages in the system is routed according each client in a unidirectional fashion, one can always ensure that the messages are routed to the right client immediately after such ring event occurred even if not everyone has the knowledge of the most up to date client list. The next most important job is then to update all the members in the ring that such event occurred so everyone has a consistent view of the client list. This is done through join, leave and remove client messages. Each of the message will add or remove a client on the local client list and nothing else (it does not affect the game state). Since each action can uniquely identified, they are atomic actions and can occur concurrently to each other and to regular game messages (in this case, concurrency means two messages can propagate in the system at the same time regardless of order, but each client still processes message one after the other). This means that a join message can be sent immediately without stopping regular game messages. And regardless of whether a join message or a regular game messages happens first, the joining client will eventually be game state consistent with everyone else. The biggest advantage of this design is that multiple client can join, leave, or dropped out of the ring at the same time without require all client to stop transmission of regular game messages. This also does require coordination and broadcasting from the name server. The name server can just be updated once to ensure it has the most up to date client list.

To see how each ring message is concurrent:

1. Join - a client issued a join is not in the ring until the client before the joining client has received the joining message. At which time two cases will happen 1) the client process join message first and initialize the new client, after that the new client is like any other client and can process any command 2) the client process any other message (including another join message) first and ignores the new client because it has no knowledge there is a new client. Then it process the join message and initialize the new client with the most up to date game state.
2. Leave - Although a leave message deletes the client from the game state, we do not care the whether other event happens first or not because the client will be gone. A client will not leave until it receives its own leave message, by which time, no one has the knowledge of that client exists in the ring.
3. Remove a client - Only upstream client can detect failures in a client, and upstream client will remove the failed client from the local client list. After that, any message will be routed to the alive client and not the failed client by the client that made the detection. When remove client causes other client to remove the failed client, the other client will not use this information directly because they are not one client upstream of the failed client. Remove a client message is here simply to keep everyone's client list consistent.

8. Evaluate your design with respect to its performance on the current platform (i.e. ug machines in a small LAN). If applicable, you can use the robot clients in Mazewar to measure the number of packets and packet sizes sent for various time intervals and number of players. Analyze your results.

We measure the round-trip time (RTT) between UG machines with ping. Each packet is 64 bytes which is similar to an event packet sent over the network, the resulting RTT between ug machines are (measured from ug138):

```
--- ug139.eecg.toronto.edu ping statistics ---
50 packets transmitted, 50 received, 0% packet loss, time 49002ms
rtt min/avg/max/mdev = 0.243/0.321/0.930/0.110 ms

--- ug140.eecg.toronto.edu ping statistics ---
50 packets transmitted, 50 received, 0% packet loss, time 49998ms
rtt min/avg/max/mdev = 0.239/0.386/1.020/0.100 ms

--- ug141.eecg.toronto.edu ping statistics ---
```

```
50 packets transmitted, 50 received, 0% packet loss, time 50100ms
rtt min/avg/max/mdev = 0.246/0.476/6.851/0.894 ms
```

The RTT is about 0.4ms between UG machines. Assuming O(N2) performance with N players, and a maximum of 0.5s (500ms) delay acceptable. Equating the 0.39 * N^2 = 500 gives the maximum players supported is around 35. This delay is higher if one connects to ug machines from outside of the U of T network. This could not be verified however because ping response is disabled on ug machines.

9.How does your current design scale for an increased number of players for the type of environment you chose? You can give examples for environments that might fit the hypothetical scenario you chose e.g., game is played on desktops or mobile devices, wired/wireless. Use Big Oh notation in your performance analysis.

Our design targets the environment of high network bandwidth, low network parallelism, low network latency and low-end nodes. It will be suitable for players playing the game on mobile device like cell phones and tablets in a fast local network, EECG lab, for example. The token ring design is suitable to play on mobile devices since each player only needs to receive packets from its previous node in the ring and send packets to the next packet, minimizes the amount of traffic for each individual node. It requires a low network latency because every packet needs to go around the entire ring cycle to ensure consistency, thus having a performance of O(n) where n is the number of players in the token ring. The design scales linearly with the number of players, where an additional player will add one more extra stop in the token ring, which may not be very ideal, but should be tolerant giving a very low network latency.

10. Evaluate your design for consistency. What inconsistencies can occur? How are they dealt with?

There are 3 sources of inconsistencies that the design covers. First is the packet drop between clients, where a packet drop introduces a missing action. This is properly handled by packet retransmission. The second inconsistency occurs because of packet re-ordering. Re-ordered packets causes actions to be executed in different sequences. The token-ring design by nature eliminates the problem, because token-ring guarantees that only one packet is transmitted at the same time. In addition, sequence numbers are used to keep track of the expected packet, hence there will be no re-ordering issue. Incoming The third source of inconsistency is bullet travel intervals. In the original design bullets are handled independently among clients, introducing inconsistency. This is handled by handling bullet movements the same way client actions are handled. This introduces ordering between bullet movements and client actions, hence ensuring consistency.