University of Toronto

Faculty of Applied Science & Engineering

ECE419 Distributed System

Design Document

Implementing Mazewar in Decentralized fashion

Yifan Wang

997036666

Chenzhi Shao

Supervised by

Professor Cristiana Amza

February 24, 2013

Table of Contents

List of Figures

List of Codes

1 Introduction

This document aims to describe the general design and algorithm used in developing a decentralized version of Mazewar game implemented in Lab 2 of ECE419 Distributed System course. In addition, possible inconsistency of game state will be described and corresponding solution will be addressed. Further more, additional design that enables dynamic player leave and join as well as Robot client will be discussed.

2 Design

2.1 Component List and Discription

Naming Service In order for players to communicate with each other without a central server processing all of their request, they must know how many players are there, and how to communicate with them. This is where the Naming Service component comes in. This service maintains a hashmap of all currently connected players as well as their input/output stream. This hashmap is updated on player join/leave, and is supplied to newly created player for them to recognize current players. See Section 2.3 for details usage of this hashmap.

Client Broadcaster For other players to acknowledge movement of local player without a centralized broadcast server, the player must have some way of deliver this information. In our design, we decentralize the role of the broadcast server by spliting it onto each current player. On each local player action such as move, turn or fire, a broadcast packet (See Section 2.2 for packet structure) will be sent to all current player detailing the movement of the local player. The broadcaster will obtain a hashmap containing all current players from the Naming Service, and loop through all the output stream to deliver each packet.

Client Listener To receive packets broadcasted from other players, a seperate thread runs continuously to listen on input streams of each different player through a selector. When a packet arrives, it temporarily put on hold until it can be determined that the ordering is correct. After that, the packet is processed, and the action specified in the packet is performed to update the local environment.

Client Holdback Queue Since there is no centralized component that can maintain order of the overall system, client must determin locally what the ordering of arriving packet should be. This however could result in inconsistent game state. As a result, this design implements a queue that holds incoming packet until it can be determined that the next packet played back from the queue is consistant accross the game. See Section 2.4 for details on how the ordering is determined.

2.2 Communication Protocol

To determin the communication protocol between clients, many options were exploited. First, a one to one socket for each pair of players were considered. In this design each player creates a socket for every existing player, and allow them to connect and send packet. To deal with blocking on listening on the input stream, a thread is created for every input stream. Clearly there is high overhead in this method that the number of thread needed is high, and will easily run into scaling issue, not to mention the possible difficulty of dealing with race condition.

The second method looked at was using DatagramSocket class. This class allows broadcasting packet to all party listening on a particular socket without having to loop through all the output stream. However, this class extends an UDP implementation, and does not garentee message delivery. One possible solution was to employ at-most-once message delivery algorithm described in class, but without centralized server, the complexity increases dramatically. In addition, the problem of creating a single thread for each input stream is still undealt.

The third method that was chosen was to use a Selector calss. This class allows the client to use one thread to handle multiple channels. Figure 1 shows how an overview of how selector can be used by one thread to monitor several sockets. With this method, broadcasting packets will be handled with a single thread, as well as listening for packets.

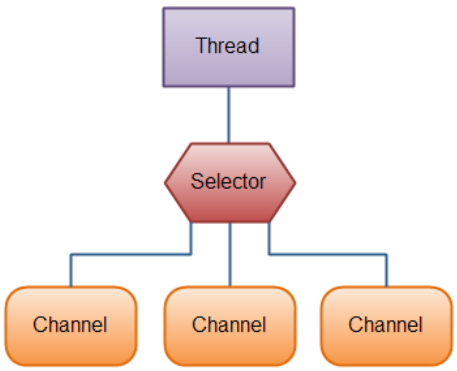

Figure 1: A thread using Selector to handle 3 Channels

Packet Structure The packet sent between the clients needs to contain information about all aspect of the game. This includes player creation, action, acknowledgement, error handling as well as action sequencing. Code 1 shows the proposed packet structure.

Code 1: Structure of packets exchanged between players

```
public class MazewarPacket implements Serializable {
    public static final int NULL = 0;
    public static final int ERROR_DUPLICATED_CLIENT = 1;
    public static final int ERROR_DUPLICATED_LOCATION = 1;

    public static final int REGISTER = 100;
    public static final int REGISTER_SUCCESS = 101;

    public static final int ADD = 103;
    public static final int ADD_SUCCESS = 104;

    public static final int MOVE_FORWARD = 105;
    public static final int MOVE_BACKWARD = 106;

    public static final int TURN_LEFT = 206;
    public static final int TURN_RIGHT = 207;

    public static final int FIRE = 300;
    public static final int KILLED = 301;

    public static final int FIRST = 400;
    public static final int PROPOSE = 401;
    public static final int FINAL = 402;

    public static final int QUIT = 400;

    //packet origin identifier
    public String owner;
    public String killed;

    //packet content
    public int type = MazewarPacket.NULL;
    public HashMap<String, DirectedPoint> mazeMap =
                            new HashMap<String, DirectedPoint>();
    public HashMap<String, Integer> mazeScore =
                            new HashMap<String, Integer>();
    public HashMap<String, Integer> mazeSequnce =
                            new HashMap<String, Integer>();
}
```

The packet origin identifier identifies the client who generated the packet, and in the case where two clients are involved (eg. killing), the source and target of the clients. Packet contents contains a type variable that identify the action the packet aims to perform, such as moving or turning. MazeMap variable are used to supply clients with a new location of a particular player in case of respawn or new player join. MazeScore variable is used to update new joined player with current score table. MazeSequnce is used to maintain a total order of packet deliver accorss, the detail of this design is presented in Section 2.4.

2.3 How a player Locate, Leave and Join

This section puts all the component together to describe the life cycle of the game play. First of all, when a player wishes to join the game, a query is sent to the naming service which replies with a list of currently connected player, and the socket address they are listening on. The new player then establishes connection with all provided sockets, and wait for the other players to acknowledge this join request. When the existing player receives a new player request, it register the new connection locally, and replies with the its own current location. After the new player have received reply from everyone else it then starts to process any action related packet in the hold-back queue based on sequence number.

When a player decides to leave, it sends a broadcast packet with type QUIT as in Code 1, and wait for acknowledgement from all players. When all the player acknowledges the quit, the player can then gracefully exist. This to maintain total order as described in Section 2.4.

2.4 Maintaining Consistency

Consistency is a major issue in a distributed game in this case. To demonstrate possible scenario of inconsistent state, we will look at several cases. First of all, consider player A and player B both made a move to position (12,7). Player B will see its own movement first and the broadcast packet from player A second, and declare that player B has arrived at (12,7) first. Similarly player A will think itself arrived first, resulting in two different game state. Second of all, we look at two players that are right next to each other firing at each other at the same time. This is quite a frequent scenario since there are many corners that leads to players facing each other. Similar to the first case, the local players might see different order of event resulting in differnt player being vaporized on different machine. Thirdly, when a new player joins dynamically, it must captures the current game state, and put all subsequnce movement on hold before the current game state and be processed and displayed. This again requires an ordering of event accross the game.

To solve this issue, we realized that an ordering needs to be established. Several types of ordering in a distributed system were looked at. First, the FIFO ordering only guarantees that if process A produces a happens-before relation, then this relation is captured on all other processes broadcasted to, it provides no information on order accorss processes. Secondly, the Causual ordering were looked at, which guarantees if message m -> m' where -> is the happened-before relation induced, then any other process will also

deliver m before m'. This again does not garentee two process will have same order of execution. Finially we arrive at total ordering, which garentees that the order of execution will be maintained throughout the system.

Total ordering leads to a consistant game state at all times accross all players, and should be applied as part of this design. We should note that total ordering does not garentee fairness in real time. For example, if player A and B are next to each other and fires at one another, if player A fired first, and player B second, total ordering does not garentee that player A will always kill player B. Instead, it garentees that if player B killed player A, everyone in the game will see the same state. To achieve total fairness, a phyical clock synchronization is needed.

3 Implementation

This section presents the implementation of design proposed in section 2.

Component Implementation

Naming Service Similar to the server component implemented in lab 2, the naming service will create a socket that is embedded on the client code. The naming service will be created first and start listening on this socket, and when a new client is created, they will attempt to establish a connection at this socket first.

The naming service will spawn a new thread on client connection. The thread will simply put the new client name, and socket address into its local hashmap, and reply the up-to-date hash map to the client, after which, the socket will be closed, and properly cleaned up.

While the naming service do form a central point of failure, its processing load is orders of magnitude lower than other component of the game, allowing it to scale up when player number becomes large.

Client Broadcaster Opposite to lab 2 server, the broadcast component will be a thread on the client side, constantly broadcasting every local action to every player registered to the hashmap locally kept, which was obtained from naming service when the player first joined.

Client Listener The listener on the client will be a thread that uses a Selector to monitor a list of channels as shown in Code 2.

Code 2: Sample code of Selector monitoring a list of channel

```
Selector selector = Selector.open();

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, SelectionKey.OP_READ);


while(true) {

  int readyChannels = selector.select();
```

```
if(readyChannels == 0) continue;


Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

  SelectionKey key = keyIterator.next();

  if(key.isAcceptable()) {
      // a connection was accepted by a ServerSocketChannel.

  } else if (key.isConnectable()) {
      // a connection was established with a remote server.

  } else if (key.isReadable()) {
      // a channel is ready for reading

  } else if (key.isWritable()) {
      // a channel is ready for writing
  }

  keyIterator.remove();
}
```

Client Holdback Queue This queue will be a priority queue that will automatically sort the packet inside with priority of sequence number decided with algorithm specified in 3.

Consistency Implementation

As described in the book /textitDistributed Systems Concepts and Design, there are two methods of implmenting total ordering, a sequencer, and the ISIS method. Since sequncer requires a centralized component that needs to broadcast a global sequence number, this is equivelent to lab 2. To decentralize the sequencer, we use the ISIS method described below.

1. Client p broadcase ¡m,i¿ with message m and unique id i to everyone.
2. On receiving m for the first time, m is added to the Holdback queue which is a priority queue and tagged as holdback. Reply to sender with a proposed priority, i.e., a sequence number /textitseq number = 1 + largest seq number heard so far, suffixed with the recipients process ID to break tie. Note that the priority queue is always sorted by priority.

3. Client p collects all responses from the recipients, calculates their maximum, and re-broadcase original message with a final priority for m.

4. On receiving m (with final priority), the receiver mark the message as OK, reorder the priority queue, and deliver the set of lowest priority messages that are marked as deliverable.