University of Toronto

Faculty of Applied Science & Engineering

ECE419 Distributed System

Design Document

Implementing Mazewar in Decentralized fashion

Yifan Wang	Ivan Shao
997036666	996608489
David Zhu	Ji Guo
996961382	996677270
Feifan Chen	Hannah Mittelstaedt
997066909	996883975

Supervised by

Professor Cristiana Amza

March 21, 2013

Table of Contents

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. Furthermore, additional design that enables dynamic player leave and join as well as Robot client will be discussed.

2 Design

2.1 Component List and Description

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 hash map of all currently connected players as well as their hostname and port address. This hash map 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 hash map.

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 splitting it onto each current player. On each local player action such as move, turn, fire or kill, 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 hash map 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 separate thread is created to listen on each input stream from each player. When a packet arrives, it temporarily puts 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 determine 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 consistent across the game. See Section 2.4 for details on how the ordering is determined.

2.2 Communication Protocol

To determine the communication protocol between clients, many options were exploited. The first method we 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 a UDP implementation, and does not guarantee 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 second method is the one used in our 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 overhead in this method that the number of threads needed is high, but since modern operating system handle multi-threading quite well, this method should be efficient for our context.

Packet Structure The packets sent between the clients need to contain information about all aspects 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 MazewarPacketIdentifier implements Serializable {
    public int lamportClk;
    public String owner;

    @Override
    public boolean equals(Object id) {
        return (this.lamportClk == ((MazewarPacketIdentifier)id).lamportClk
        && this.owner.equals(((MazewarPacketIdentifier)id).owner));
    }

    @Override
    public int hashCode() {
        int hash = 1;
        hash = 31 * hash + lamportClk;
        hash = 31 * hash + owner.hashCode();
        return hash;
    }
}
public class MazewarPacket extends MazewarPacketIdentifier
implements Serializable, Comparable<MazewarPacket> {
    public static final int NULL = 0;
    public static final int ERROR_DUPLICATED_CLIENT = 1;

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

    public static final int ADD_NOTICE = 200;
    public static final int REPORT_LOCATION = 201;
    public static final int ADD = 202;

    public static final int MOVE_FORWARD = 300;
    public static final int MOVE_BACKWARD = 301;

    public static final int TURN_LEFT = 302;
    public static final int TURN_RIGHT = 303;

    public static final int FIRE = 400;
    public static final int INSTANT_KILL = 401;
    public static final int KILL = 402;

    public static final int QUIT = 500;

    //packet definitions
    public String newClient;
    public String ACKer;
    public String victim;
    public int seqNum;

    public InetSocketAddress address;
    public HashMap<String, InetSocketAddress> clientAddresses;

    public int type = MazewarPacket.NULL;
```

```
    public DirectedPoint directedPoint;
    public Integer score;

    // Additional variable to indicate number of clients that
    // is multicasted with this packet
    public int cardinality;

    @Override
    public int compareTo(MazewarPacket o) {
        if (this.seqNum == o.seqNum)
            return this.owner.compareTo(o.owner);
        else
            return this.seqNum > o.seqNum ? 1 : -1;
    }
}
}
```

The packet identifier class identifies a packet uniquely by its lamport clock and the client's name who generated the packet, and in the case where two clients are involved (e.g. killing), killer will generate the packet and victim's name is recorded in the victim field. Packet contents contains a type variable that identifies the action the packet aims to perform, such as moving or turning. ACKer is the name of the client that acknowledges an action packet. This field is null for action packets. Seqnum is the lamport clock for the packet when the sender sends it. In case of an action packe, it is the same as the lamportClk field. But in the case of an ack packet, it is usually different from the lamportClk field since lamportClk field is to identify the original action packet. Address is the ip address of the sender client to register at the naming service. Client addresses is a map returned from naming service that includes all the connected clients' names and ip addresses. Directied point variable is used to supply clients with a new location of a particular player in case of respawning or new player joining. Score variable is used to update new joined player with current score table. Cardinality variable is to indicate the number of connected clients at the moment of multicasting. LamportClk and seqnum field together are used to maintain a total ordering of packet delivery acorss all players, 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 components 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 registers the new connection locally, and replies with 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 waits for acknowledgement from all players. When all the players acknowledge the quit, the player can then gracefully exit. This is 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 lead 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 capture the current game state, and put all subsequent movement on hold before the current game state being processed and displayed. This again requires an ordering of events across 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 acorss processes. Secondly, the Causal 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 guarantee two processes will have same order of execution. Finally we arrive at total ordering, which guarantees that the order of execution will be maintained throughout the system.

Total ordering leads to a consistent game state at all times across all players, and should be applied as part of this design. We should note that total ordering does not

guarantee 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 guarantee that player A will always kill player B. Instead, it guarantees that if player B killed player A, everyone in the game will see the same state. To achieve total fairness, a physical clock synchronization is needed.

3 Implementation

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

3.1 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 hash map, 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 does 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 number of players grows.

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 hash map locally kept, which was obtained from naming service when the player first joined.

Client Listener The listener on the client will be a series of threads each listening on an input stream to one of the currently connected players.

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 3.2.

3.2 Consistency Implementation

As described in the book Distributed Systems Concepts and Design, there are two methods of implementing total ordering, a sequencer, and the ISIS method. Since sequencer requires

a centralized component that needs to broadcast a global sequence number, this is equivalent to lab 2. The second method was the ISIS method described below.

1. Client p broadcasts <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, seq-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-broadcast original message with a final priority for m.
4. On receiving m (with final priority), the receiver marks the message as OK, reorder the priority queue, and deliver the set of lowest priority messages that are marked as deliverable.

During lecture a third method was discussed using Lamport clock, acknowledgement and a holdback queue. The section below describes this implementation and why it was chosen for this design.

1. Client p broadcasts <m,l> with message m and local Lamport clock number.
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 an acknowledgement with the up-to-date Lamport clock.
3. Client p collects all responses from the recipients, and plays the message from the front of the holdback queue when all the acknowledgements from all peers are received.

This method is selected because it guarantees total system ordering as well as decentralized component, while being simpler in implementation than the ISIS method presented earlier.

4 Evaluation

4.1 Starting, Maintaining, Exiting

The strength in this design reflects from the fact that naming service is extremely simple, allows it to scale from 10 to 100s. On the other hand, it clearly forms a single point of failure.

4.2 Performance on Current Platform

The performance of this design will be re-evaluated once the lab is complete. However, based on estimation, the packet traffic will be approximately doubled from implementation in lab 2, and since lab 2 offered great responsiveness on LAN, estimation is that user responsiveness for this design will at least be playable. After the implementation of the lab, the responsiveness when running up to 6 clients was still satisfying. As expected, thanks to the LAN connection, our implementation guarantees a responsive, playable game with decent scalability.

4.3 Consistency

The consistency issue is raised by different processes with a mixed number of thread all concurrently sending and receiving packets. However, since we implement a consistency algorithm described in Section 3.2, total system order is guaranteed. As a result, all processes will playback packets in the same order and players will see consistent state in UI. Similarly, as discussed, this in no way guarantees fairness due to the lack of a synchrornized physical clock.

4.4 Other Factors for Performance

Scaling with players As discussed early, the naming service allows great scalability. The broadcast method however, requires each pair of players to maintain a socket connection. Since TCP layer is required to guarantee no packet loss and in order delivery, we could not use other method of packet broadcasting that does not require a large number of sockets. The number of socket connections grows exponentially as the number of players

grow, and could eventually lead to slow system performance. However, as in the context of this lab, this design should easily support up to 4 players

High latency network In high latency networks, packet could be lost and have high delivery time. However, since we employ TCP layer with our socket implementation, packet delivery is guaranteed. However, there could be latencies where packet from some machines are not delivered. In that case, the game will be stalled in waiting for packet deliveries. As a result this design will only work well on a system where all players are present on a consistent and responsive network.

Mixed Devices Same as the scenario described in the High latency network, a mixed group of devices such as laptop/mobile on a mixed network produces high latency due to overhead of the network protocol. These delays will be reflected on the UI since every acknowledgement of packet delivery needs to be received for a move to be played.