ECE419 DISTRIBUTED SYSTEMS: LAB 3 DESIGN DOCUMENT

University of Toronto

Faculty of Applied Science and Engineering

Programming Team Members and Student IDs

Divya Dadlani: 999181772
Geetika Saksena: 998672191

Design Team Members and Student IDs

Team 1- Divya Dadlani: 999181772, Geetika Saksena: 998672191
Team 2- Saminah Amin: 998714654, Nayantara Prem: 998698351
Team 3- Richa Rajgolikar: 999326956, Rupangi Mehta: 999270970

Date: Monday, March 24th, 2014

We have implemented a multiplayer Mazewar game which allows dynamic joins. This document describes the functionality of the algorithms used in this game and analyses the strengths and weaknesses of this distributed system.

Distributed Algorithm Overview

We have used the Total Order Multicast (TOM) algorithm to implement the distributed version of the Mazewar game. It is completely decentralized as no node is given any special responsibility. The only centralized component is the naming service which is only used to store and fetch client connection information upon joining and leaving the game.

There are two main threads running simultaneously in each client. The listener thread consists of a server socket that continually listens for incoming packets on the known port and spawns a handler thread to queue each request, sorted by the Lamport clock value. The execution thread is responsible for handling the moves that are dequeued. When a user presses a key, the node detects the move and multicasts it to every player.

Communication Protocol

The clients in the game will communicate using a MazePacket. Communication between clients is done over TCP/IP, by using ObjectStreams to send and receive the MazePacket. It consists of the following fields: Message Type, Client Info, Client ID, Event Type, Lamport Clock Value, Error Code, and Number of Acks. The MazePacket is used both for communicating with the naming server for initial setup information, as well as to exchange move requests and acknowledgments between clients during the game.

Locating, Joining, Quitting the Game

The game supports dynamic joins with the assistance of a central naming service. Any player joining the game will register itself with the naming service and request the location(ie. port number and hostname) of all the other existing players. The joining client will then send a connection request to all the players. Once the clients receive the connection request, they will reply to the client with their current positions and orientations so that the new client can add them. Then, the new client will add itself and broadcast its new position and orientation to the other players for them to add.

When a player quits, it first deregisters itself from the Naming Server so that no new player sends it a connection request. Then, it multicasts a DISCONNECT message to every player so that they update their own address books. After receiving all the ACKS, this player will close all its socket connections and exit. When the other players dequeue this message, they will remove that particular remote client from the maze.

Timings of Protocol events: Lamport Clocks
To implement total order and ensure game sequential consistency, our design makes use of Lamport clocks to queue requests. The Lamport clock value is a double, with the part after the decimal representing the client ID and the part before the decimal acting as the Lamport clock value. In case of a queueing conflict with the same Lamport clock, the decimal client ID value is used to resolve the conflict.

The local Lamport clock values for each client are synchronized upon each send and receive action. The Lamport clock is incremented just prior to sending and also inserted in the Maze Packet. Upon receiving a packet, the receiver synchronizes its own Lamport clock with the one received, by updating its own local Lamport clock value to the greatest of the packet and its own clock values plus one.

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

Strengths:

- The game implements dynamic joining which provides flexibility to the users playing
- It is a completely decentralized system and so no node is a bottleneck
- The move packets are ordered according to the lamport clock which makes the game fair and allows for consistency
- When a player quits the game, the rest of the players are not affected and the game functions normally

Weaknesses:

- When a node goes down or a packet is lost, the game hangs as every other player cannot proceed without the ack of any particular node
- When a player vaporizes, there is an inconsistent state untill the vaporized player moves again
- The number of packets (move packets and acks) that are multicasted can lead to a slow game if the number of players are high

Evaluate your design with respect to its performance on the current platform (i.e. ug machines in a small LAN).

- The design is optimal in the UG labs as there is no visible delay as the connection is fast.

How does your current design scale for an increased number of players? What if it is played across a higher-latency, lower-bandwidth wireless

network - high packet loss rates? What if played on a mix of mobile devices, laptops, computers, wired/wireless?

- The game assumes a maximum of 10 connected players, but that can be adjusted by setting the decimal part of the Lamport clock values to be more than one decimal place.
- As stated previously, the design is optimized for a Local Area Network. It relies on the reliable network assumption of Total Order Multicast, i.e. it assumes no packets are lost in transfer. However, we have implemented queueing functionality to allow for out-of-order packets. Hence our implementation would also work on a small LAN using wireless connection.
- The wireless handling feature is limited, as it cannot handle a mix of networks across a larger geographical area. Moreover it does not implement fault tolerance therefore it cannot be used on several wireless devices with disconnections and reconnections, where packets and Acks will be lost frequently
- As the application is implemented in Java, any system running Java with a small LAN can run this game without noticeable delays.

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

- The design is consistent throughout when the players join, move and quit
- The scores are synchronized correctly in every node
- The only temporary inconsistency is when a player is vaporized. The consistent state of the game is restored when the vaporized player is moved once again.
- As stated above, even when the FIFO assumption does not hold, our distributed algorithm deals with packets that are out of order - Sequential consistency is thus ensured throughout