Distributed Systems
ECE419
Assignment 3

Design Document

TO MCast with Lamport Clock

Submission date: March 14th, 2016

Student Names	Student Numbers
Joo Young Kang	995903330
Hae won Kwak	998339392

1. Overall Design

Our design for Assignment 3 implementation uses TO multicast with Lamport logical clock to achieve decentralized behaviour. Basically, each client will multicast event request to all other clients through information acquired from naming service and only deliver message to display (application) once n acknowledges (including itself) are received. Specifically, our design is optimized to work under the following environment.
-> low/high network bandwidth, very high parallelism, very low network latency, low/high end nodes

2. System Design and Components

● Naming Service

In our design, naming service is responsible for connection/disconnection setup(joining and exiting), clients list maintenance. It will store client name, host name, and unique port number each client will be listening to in concurrent hashmap data structure (key=client ID). When player requests to join the game, each client will specify preferred port number. Unless the same port number is requested, naming server will acknowledge the connection and store each client data in hashmap table and also assign specific processor ids to each client. Processor ids will be used to break the tie in Lamport clock.
Assumption: every clients know where naming service is running and it is always started (or running) before actual game is starting.

● Client Packet

Each client packet will contain

- Processor ID (unique for each client, used to break tie)
- Lamport Logical Clock Number
- Event Type (join, move, fire, exit, ACK)
- Event Information (direction, player name)

● Client Listener/Sender/Handler

Client listener is responsible for

- Receiving peers' MPacket messages
- Enqueuing incoming event queues into local client queue
- Reorder events (local queue) in monotonically increasing Lamport logical timestamp

Client sender is responsible for

- Attach Lamport clock of local processor
- Handling all outgoing event queues

Client handler is responsible for

- Dequeuing event from head of the queue and deliver the message to application (display)

- Update local user's screen based on packet information

● Total Order Multicasting Algorithm

Our protocol will be based on TO Multicast with lamport logical clock. In order for this algorithm to work, minimal reliable delivery will be implemented with a simple timeout/retransmit architecture to deal with packet loss. Just like Ricart-Agrawala algorithm, when a player has an event to be multicasted, he/she multicasts a request containing its lamport logical clock timestamp(ts) to the group. If a receiver also has an event to multicast, ts of received request and its ts are compared. If ts of received request is lower, it sends an OK reply. If ts is higher, receiver will queue it and sends nothing back. If the sender gets N number of OK replies including from itself, it will now ready to multicast the actual client packet. Once player gets a client packet, it sorts according to ascending order of ts and saves it into the queue.

3. Questions to be answered

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

Strength

- Client joining/exit requires only 2 messages exchange with naming server. (1 for request and 1 for acknowledgement) Scales up well with high number of clients
- Total ordering is guaranteed across all players
- Scalable with large number of systems under such assumptions: High parallelism, Low latency
- Packet drops are handled with retransmit

Weakness

- Single point of failure (naming server failure will cause entire system to malfunction)
- Lack of detecting node failure

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

Assuming packet latency over ug machines to be less than 0.5ms in LAN, our algorithm is not expected to run slow (unplayable) in such high-end nodes, considering our packet is relatively small in compared to maximum packet size over TCP (65K). For one message to deliver from one client to peers and then execute on local screen, N multicasts and N^2 ACKs are required for single post.

3. *How does your current design scale for an increased number of players for the type of environment you chose? (hypothetical scenario, Big Oh notation analysis)*

- For example with N clients, each event requires N * N^2 messages (including multicast to itself) to be delivered on local screen.
- If it scales up to **1000 players**, it will require 10^6 + 1000 message per entry/exit and also the same number of messages in delay for each entry per node.
- Big Oh Analysis: **O(N + N^2)**

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

- As our design ensures totally ordered multicast of events across clients, consistency in each client's local screen orientation is guaranteed. (under reliable multicasting availability)
- However, if original packet is lost and retransmit packet is also lost, it might appear to be lagging or even worse, packet drop won't be detected forever. In result, it will
- However, as the biggest drawback of this algorithm, it is definitely slower than centralized algorithm (Assignment 2) which might "appear" to be inconsistent across client if large number of messages are transferred through large number of nodes (>1000 nodes = 10^6 messages required per event).
- N-points of failure: If any client crashes during the game, it will block all subsequent events. Therefore, entire game will block since none of the events can be dequeued from local queue,