Design Document for Lab3
Group: Erwin Magnaye, Seungmoon Rieh
Date: March 17, 2015
Title: Minimal message exchanges with delays and fault tolerance with a timeout mechanism

The basic mechanism is the following,

- A node that wants to enter critical section send REQUEST message to all other nodes.
- All other nodes that receive the REQUEST message compares:
    - If the receiver does not want to enter the critical section, send REPLY
    - If the sender of REQUEST has higher priority, then send back REPLY
    - If the receiver of REQUEST has higher priority, then the REPLY is queued until the receiver exits the critical section.
- Once the node acquires the lock, it enters the critical section and will update it's own state with its requested MOVE and broadcast a message with its MOVE to all other nodes
- When the node receives the MOVE_ACKs from all nodes, it leaves the critical section and dequeues and sends all REPLYs that have been accumulated

In each node, two different types of threads are running

- One thread to handle keyboard interrupts and send REQUEST messages to other nodes
- N-1 threads to handle receive REQUEST messages, send and receive REPLY messages, send and receive MOVE messages, and send and receive MOVE_ACK messages to and from the other N-1 nodes

Each node must have its own local data structure and local state. A lock can be used to protect data structure from different threads that are running on each individual node.

CONSTANT
Me: number that represents this node
N: number of nodes in the network
INTEGER
Seq_num: sequence number for this node's REQUEST
Highest_seq_num: highest sequence number this node has seen
Reply_count: number of replies this node has received
Move_ack: number of move acknowledgments this node received
BOOLEAN
Request_CS: TRUE if this node wants to enter critical section
Reply_deferred [1:N]: Reply_deferred [j] = TRUE if this node is delaying reply to node j
MUTEX_LOCK
Lock_acquire
Lock_release
BITMAP
Awaiting_REPLY [1…N]

Strength of this algorithm

- This algorithm guarantees mutual exclusion, no deadlocks, no starvation, and fairness
- Message traffic is 2*(N-1) to acquire a lock and enter critical section

Weakness of this algorithm

- Additional 2*(N-1) messages during the critical section to make a move
- N points of failure where one node can fail the whole system

Achieving Fairness

- This design achieves fairness because it is using Lamport clocks to achieve total order of all REQUEST messages when requesting the critical section.
- The use of monotonically increasing Lamport clocks will ensure the node with the lower timestamp will always obtain the lock first.
- During tie breaks, the node with the lower node ID wins.

Time bounds:

- Before any node can enter critical section, it must make sure no other nodes are entering. This checking process takes at least one round trip. Therefore, no request can be serviced in less than one round trip time.
- During the critical section, it must send it's move to all other nodes and receive an acknowledgement before it finishes its critical section. Therefore, the critical section is processed in no less than one RTT
- Throughput: when a node leave critical section, no other node can enter in less than one way trip time because the node leaving critical section has to notify all other nodes that it has left.

Communication protocol:

- All nodes communicate through TCP/IP protocol to ensure reliability and FIFO for each channel
- The packets are sent using the ObjectOutputStream and received using the ObjectInputStream provided by the Java library
- The object sent is of type MazewarPacket which contains all information in the message which include the type, sender name, lamport clock timestamp, and other useful information

Node addition

- Obtain a list of all other node ID, IP address, and TCP port from the Naming Server and register its own node ID, IP address, and TCP port
- Request the location of all nodes and register those nodes to local board
- Register its own location to all other nodes

Node deletion

- Delete this node from NAMES array
- Destroy data structure at this node
- Decrement N by one

Fault Tolerance:

- Fault tolerance is not supported by this algorithm
- If one node fails, it fails to respond to requests and a node may never receive N-1 REPLY messages in order to enter the critical section

- The probability of failure is N*P(1)
- The fault tolerance of this design is a huge bottleneck since it only requires one of the N nodes to fail the system (N points of failure)

Analysis

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

	Strengths	Weakness
Starting	- Only requires the location of the naming server to register nodes and start the game	- If the naming server fails, the locations of all nodes cannot be found and game cannot start - Nodes must not move until all players have joined or it will lead to inconsistent state - Dynamic joins not supported
Maintaining	- It does not require any centralized component, i.e. once game starts naming server is not needed	- N points of failure
Exiting		- Nodes exiting game are not supported - When one node leaves, system fails

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 sent for various time intervals and number of players. Analyze your results.

- Total moves per second =15 moves per second. Messages per move = 4*(N-1) = 12 messages per move. Total messages of moves = 15*12 = 180 messages per second over the network.
- Total ticks per second= 5 per ticks per second. Messages per tick = N-1 = 3 messages per tick. Total messages for ticks = 5*3 = 15 messages per second. To select next tick leader is negligible.
- TOTAL MESSAGES OVER THE NETWORK approx 200 messages per second.
- Each message is approx 200 bytes
- Total throughput of the system is 200*200 = 40 KBps = 320 kbps over 6 total channels
- This means each channel requires about 53 kbps, which is minimal for the ug machines, which operate over several megabits per second. Therefore the game requires minimal bandwidth but latency is still an issue since our design requires a minimum of 2 round trip times for each move.

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 design does not scale well for large amounts of players since the number of messages sent for entering a CS is 2*(N-1) and during the critical section is 2*(N-1). Also, the speed of the game has a bottleneck on the latency and bandwidth of each node's communication channel to each other. If any one node has a slow communication channel to another node, it limits the speed of the design.
- If it is played across higher- latency, lower-bandwidth wireless network the design will do well. It is limited more by the RTT while acquiring the lock and the RTT during the critical section than it is by its bandwidth throughput. The packets of the messages are small in size.
- During high packet loss rates, this will lead to huge issues in our design because our design is limited by its RTT. In packet loss, it may take up to one retransmission-timeout (RTO) to resend the lost packet. The RTO can be a one-second delay at minimum, which will clearly show in the game.
- If it is played on a mix of mobile devices, laptops, computers, wired/wireless the design's speed will be limited by an one of these communication channels and its latency/ bandwidth between any of these nodes.

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

- This design ensures that no inconsistencies can occur during normal gameplay
- The following irregular gameplay occurrences will cause inconsistency:
- Failure of one node if the node fails while sending a MOVE message. This case is not dealt with because we assume no node failures
- Failure of a communication channel. This is not dealt with because we assume communication channels are reliable and FIFO.
- Moving a node/firing while other nodes are still joining the game. The scores are not updated and will cause score inconsistency.

Reference

Ricart, G and Agrawala, A. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Comm. ACM* 24, 1 (Jan. 1981), 9-17