CS677: Programming Assignment 2
Renaldo Part Deux: A Server-less Game

Distributed Operating Systems Spring 2003
Department of Computer Science
University of Massachusetts, Amherst, MA 01003 USA
shenoy@cs.umass.edu

1 The problem

This project has tree purposes, namely to gain experience with...

- issues concerning distributed consistency
- transactional techniques
- leader election algorithms
- clock synchronization

You are encouraged to re-use your work from your first programming assignment as much as possible. In this project, you need to design a server-less version of your centralized multi-player game from programming assignment 1.

2 Introduction

After learning of the wonderful work of CS677 students on their Renaldo game concept, Fundingo game corporation has decided to pick up the Renaldo game project. A meeting with Dr. Jane Dasani, Fundingo's Chief Scientist and proud UMass-CompSci Alumna, has resulted in the edict, "Hey, make it scale." After hours of technical discussion, the descision was made. "Dude, go server-less."

3 The concept

In programming assignment 1, you implemented a centralized client-server approach. In the client-server game system, clients issue requests to a centralized

server which implements game dynamics and computes global game state. In a server-less environment, each client maintains its own game state. In this assignment, we no longer employ a centralized game server. A server-less gaming environment can be thought of as a peer network. A peer environment has many advantages over a centralized approach. These include robustness and scalability. A peer environment is more robust because a single failure only affects the failing node. A peer environment is more scalable because it induces a natural partition of load (computation of state) among participants in the game. As is the case with many distributed systems issues, these advantages come at a cost.

Because each client computes its own game state, there is a real possiblility for inconsistency across each client participating in the game. That is, two participants may come to a different "understanding" of the current state of the game. For example, two Renaldo clients can incorrectly think that s/he has consumed the same food item in the same grid square.

Because of this, each participant must implement a means of coping with computation of distributed state. Simply put, the participants (or peers) in the game must reach agreement or consensus on a number of things. Consensus is a recurring theme you will see in distribted systems in areas such as leader election, distributed synchronization, and transactions [3]. Generally, in a distributed system, consensus is reached among participants by the exchange of messages. The format, meaning, and response to these messages are our protocol. When you think of what is a client and what is a server, think about what role each plays. A client initiates a request. A server receives a request, performs some computation and returns the reply. A peer can assume both a client and server role. It acts as a client when it initiates requests. It also acts as a server when it receives requests, performs a computation and returns a response.

4 Game Details

To simplify things a little, your game will be limited to 4 players. Each player maintains a copy of the grid world including the grid squares, and walls(figure 1). When a participant makes a move, he must inform his peers in the game. This is to be accomplished using application-level multicast. By this approach, a list of peers on the network is maintained and a unicast message is sent to each (figure 2). Thus, each peer implements an application-level multicast client to perform application-level multicast (ALMC).

One of the peers is to be designated as the chairperson (chair). The chair is responsible for managing the placement of food and publication items in the department. Thus, the chair implements a food/publication client while each peer implements a food/publication server. Renaldo's advisor, Prof. Rhea Sarch, has gone on sabattical for a semester. While this means that Renaldo's advisor will not be present (at least for programming assignment 2), Renaldo must still "read" publication items in addition to eating food and drinking coffee. When the chair places a food item or publication, his "move" is ALMC'ed to peers on the network. The game begins with a single peer. As new peers join

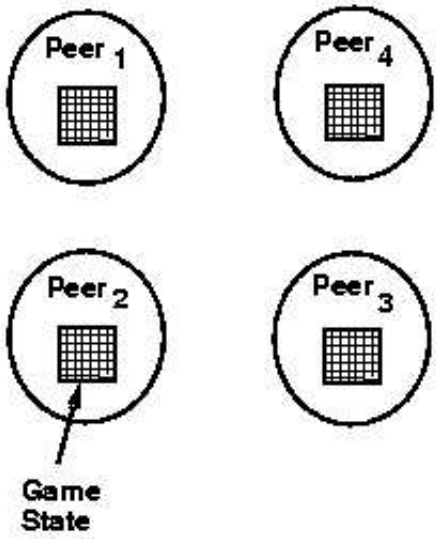

Figure 1: Server-less game

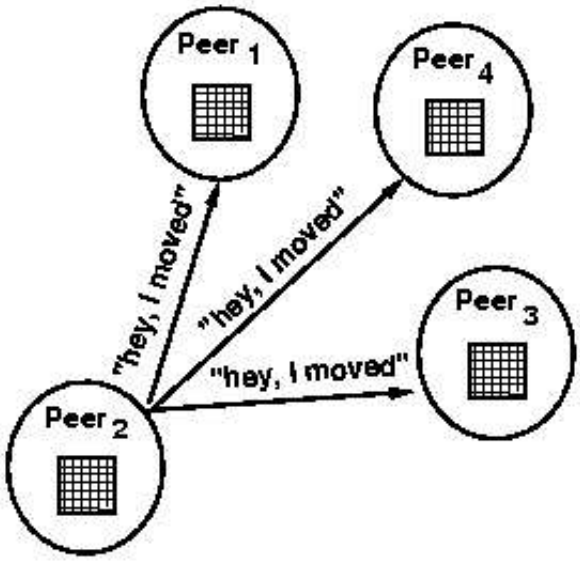

Figure 2: Application-level multicasting of moves

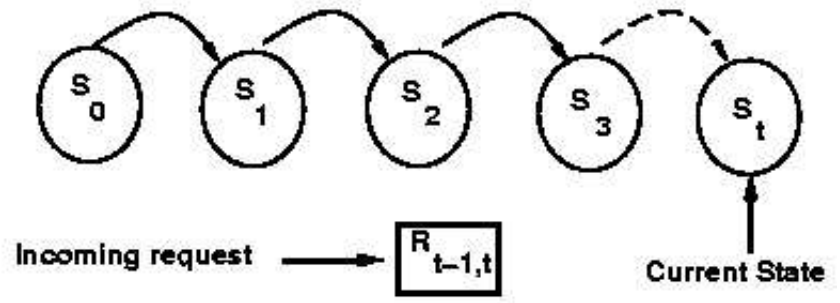

Figure 3: State changes due to request messages

the game, they must be informed of the current game state. It is also the job of the chair to bring the new graduate student up to date. The department chair also implements a bootstrapping service used to bring newly joining peers up to date on game state. Thus, each peer implements a bootstrap client while the department chair implements a bootstrap server. It is assumed that peers perform graceful joins and leaves for the game. When a new peer joins the network, it discover's the department chair and engages in a bootstrapping protocol to acquire state from the department chair. As update requests are serviced by a peer, its local state moves through a sequence $s_0, s_1, s_2, \ldots$ of states. Define s_t to be the state of a peer at timestep t. Given a request r_{ij}, a state transition (s_i, s_j) is effected (figure 3).

If the the current state (at timestep t) is such that $s_t \neq s_i$, then the request is a conflicting message. When this happens, the peer has reached an inconsistent state. Assuming a total ordering on request messages, system state must be reverted to the last known consistent state s_i. In transaction processing, this is known as *rolling back* system state (figure 4). To implement rollback, a log of state history must be kept. Because we are not concerned about durability of system state, this log does not have to be made persistent. An in-memory log of up to 100 messages is adequate. Exactly how logging is implemented depends on the type of actions performed.

The two classes of actions with which we are concerned are known in transaction processing as *unprotected* and *real* actions [2]. This distinction is made based on wether or not the change in state brought about by an action can be reversed or *undone* by executing an operation or compensating action similar to the original action. In our server, actions are unprotected. For an unprotected action, the system requires knowledge of the compensating action c_i and a piece of data d_i. Associated with execution of each action a_i, a tuple $< o_i, d_i >$ is appended to a list. Given a current state and data item, o_i performs the functions necessary to return system state from some state s_j to s_{j-1} where s_{j-1} is the state just before a_i executed or $o_i(s_j, d_i) = s_{j-1}$. Consider a "move left" action performed on a move request server which caused Renaldo to move from position $(2, 2)$ to $(2, 3)$ and his energy level to change from 25 to 24 in a peer's local state. Associated

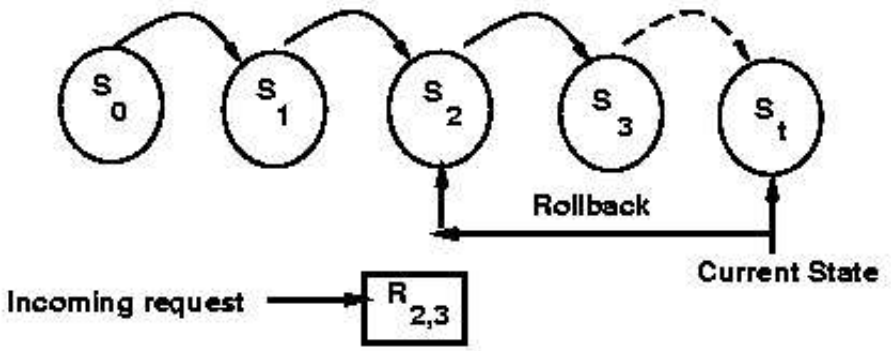

Figure 4: Roll back state upon conflicting message

with this action is an operation $undoLeft$ and a data item $< 1, < 2, 2 >>$ where the $< 2, 2 >$ is Renaldo's location before the move and the 1 is the number of units decremented from Renaldo's energy level. The $undoLeft$ operation will set Renaldo's position to $< 2, 2 >$ and add 1 to his energy level. Given a sequence of such tuples $< o_0, d_0 >, < o_1, d_1 >, \ldots, < o_n, d_n >$ and a current state s_t, a rollback can be performed by iterating through the list and computing $o_i(s_t, d_i) = s_{t-1}$ sucessively for each tuple in the list. This list is our undo log. Iterating through it as mentioned is rollback. An example of inconsistent state is a request to place vegetable#1 in position $(5, 5)$. If a peer's current state s_t is such that vegetable#1 is currently in position $(1, 1)$, then the request is a conflicting message.

5 Reaching Consensus

Consensus must be reached on the total ordering of request messages. Mechanisms covered in class include Lamporth and Vector clocks. You are to choose one of these mechanisms to ensure a total ordering on request messages exchanged among peers. The department chair is a single point of failure. In the event that the department chair crashes, a new chair must be elected. A number of leader election algorithms have been covered in class. You are to choose one of these mechanisms to ensure that all peers in the network agree on who is the department chair.

6 Protocols

The following protocols are to be designed and implemented...

- discovery of chair

- bootstrapping
- moves in grid world
- food/publication placement
- application-level multicasting

7 The assignment

You are to implement the mentioned consensus items, protocol items, and rollback mechanism. To reduce complexity, you may reduce the size of the grid to 50×50. Remember to start thinking about this early and to spend your time focussing on the distributed sytems issues.

8 Extra Credit

Application-level multicast does not scale with the number of peers. In a distributed system, we have a physical network which is modeled as a graph. In this graph, a node represents a host on which a peer runs and an edge represents the physical interconnect between nodes. This graph is sometimes referred to as a host graph. Among the peers we maintain "logical" connections through an RPC mechanism. This is modeled by a graph where nodes are peers and edges are logical connections. This is sometimes called a task graph. An overlay network is a task graph built upon or *embedded* in a host graph. For extra credit, construct and employ an overlay network for the distribution of request messages as an alternative to using application-level multicasting. This approach is more scaleable and efficient then having each peer send $n-1$ unicast messages.

References

[1] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet, 1999.

[2] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, San Fransisco, CA, 1993.

[3] N. Lynch. *Distributed Algorithms*. Morgan Kaufman, San Fransisco, CA, 1996.