Distributed Multiplayer Gaming

SnakeD

by ROAM

Rama Bhupathiraju (rbhupath)
Osei Poku (opoku)
Aditi Pandya (apandya)
Manan Patel (mdpatel)

Table of Contents

1. Introduction

This document describes the functionality and design of the implementation of **Snake** as a distributed game system. Snake is a popular game first release in the mid 1970s in arcades. After it became the standard preloaded game on Nokia phones in 1998, Snake found a massive audience. The player controls a long, thin creature, resembling a snake, which roams around on a bordered pane, picking up food (or some other special items), while trying to avoid hitting its own body or the obstacles in the playing area. Any such collisions would spell the end of the game. The snake's tail also grows longer as more food or special items are consumed.

The implementation described in this document is of a multiplayer version of the game that is to played across the network. Typically, multiplayer games involve a centralized server node

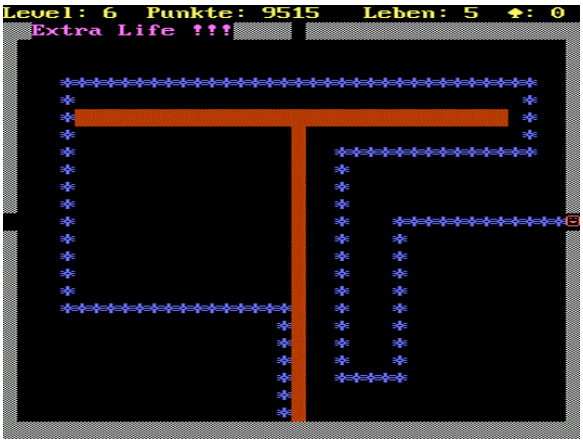


Figure 1: *Snake on an arcade screen*
http:/en.wikipedia.org/wiki/Snake_(video_game)

that other player nodes or *clients* connect to. The purpose of this server is to coordinate the events and decide on the shared game state that is then communicated to the clients playing the game. The distributed Snake game, referred to as **SnakeD**, avoids this client-server architecture and adopts an architecture in which the game state and player-generated events are shared in a completely distributed manner. Despite the additional communication and coordination complexity, this distributed architecture is more robust and reliable due to the lack of a single point of failure.

2. IMPLEMENTATION CONSIDERATIONS:

ERLANG: Erlang is a functional programming language. It is designed to support distributed, fault-tolerant, soft-real-time applications. Erlang is a programming language designed at the Ericsson Computer Science Laboratory and is now open source. In Erlang, processes communicate using message passing instead of shared variables, so there are lesser concurrency issues/bugs.

TCP: We initially started with using UDP for our project because of the real-time requirement of our application. But when testing it, we found that it was too unreliable on wireless. So, we switched over to TCP.

CHALLENGES:
We faced the following challenges during the design of the Distributed Snake game:

- Achieving real time responsiveness
- Synchronizing the clocks/event updates at all nodes
- When using "leader" to synchronize nodes, achieving continuity when leader dies or other nodes in the game die
- The game should be consistent in spite of the network delays

3. Project/Game Requirements

The primary requirements of the project will be to implement a playable, multiplayer, distributed version of Snake. Specifically, the following rules will be enforced. During game play, each player will independently control a snake according to the traditional rules of acquiring food and avoiding collisions with other players and obstacles. The game is inherently fast paced and so to maintain that pace, we will try to achieve a high enough frame rate (approx. 200ms) that creates the perception of a smooth snake motion. A simple point system will be decided later as this project is focused more on the implementation of the distributed system than on the game play. We will try to achieve 8 simultaneous players in one game.

To achieve a truly distributed architecture, we will avoid having a central game server which makes decisions about events occurring in the game. Each node will independently participate in the decision-making according to the rules that we define.

4. Design

This section describes the architecture and other implementation details of **SnakeD**. There are six modules that make up the node. The module is designed to handle a single specific task and they combine to form the system by sending messages between themselves.

4.1. Overview

Figure 7 illustrates the design of the system. The user is presented with an interface controlled by the **UI** module. The only objective of UI is to present the current state of the game to the user. **Keyboard** module persistently listens for user activity on the keyboard or mouse depending on the mode of control. Upon the occurrence of any event, Keyboard sends the events to **GameLogic** module. This is the "brain" of the game. Its objectives are to initiate broadcasting of events, process collected events from the network and update the interface. GameLogic achieves this by sending the appropriate messages to UI, MessagePasser. Specifically, the game communication protocol described earlier is implemented in this module. The final primary module is **GameManagement**. In this module, the bootstrap protocol will be implemented. This means that Game Management is responsible for managing adding of new players and removing of dead players.

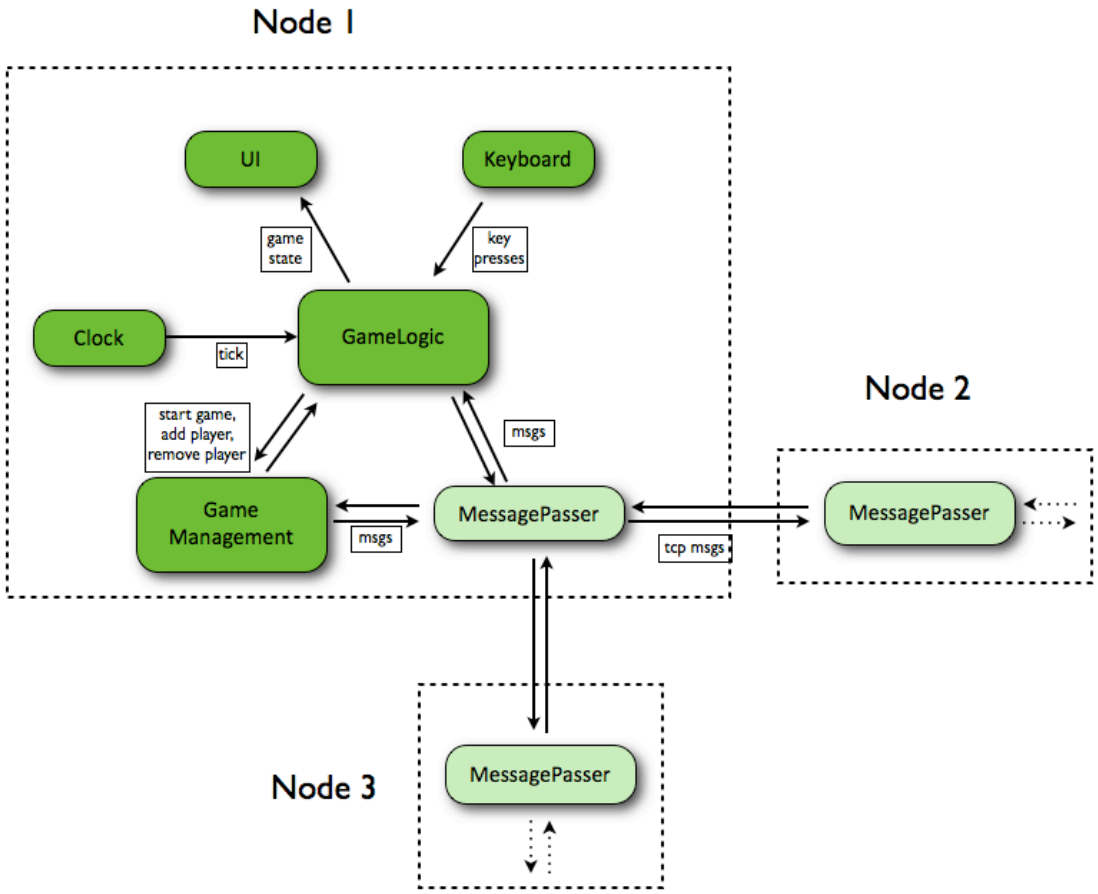
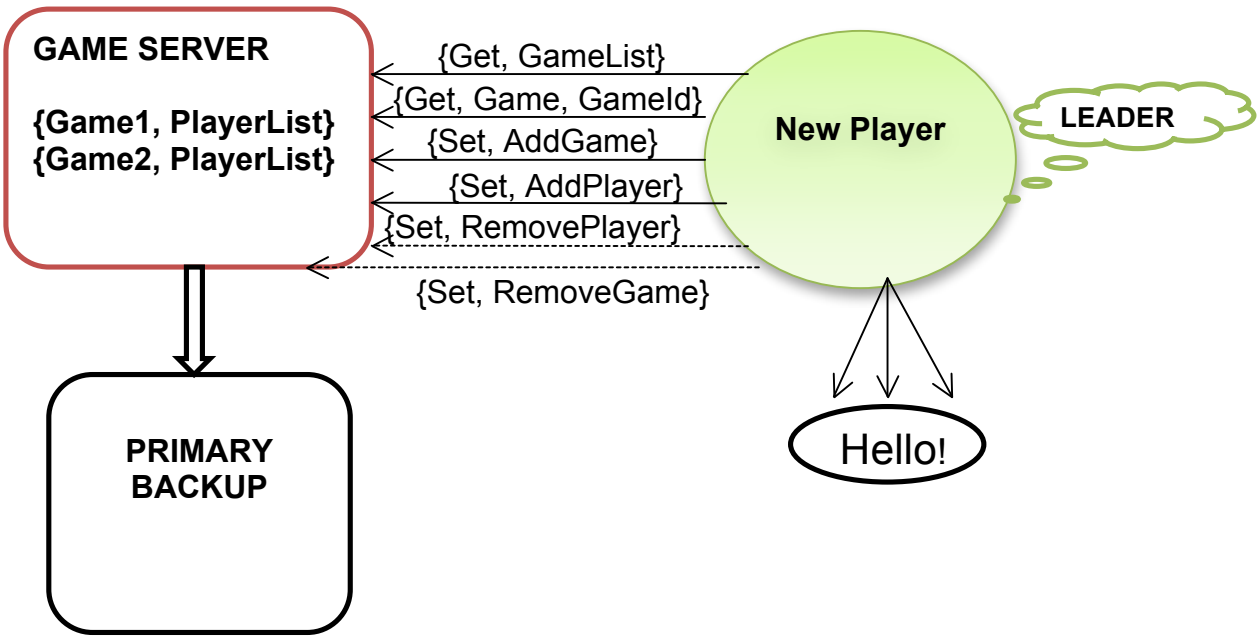


Figure 7: *Game Architecture*

Two other modules serve only as utility modules and perform critical actions but have no decision making capability. The **Clock** module keeps track of the ticks in the leader node and allows other nodes to perform timeouts. The **MessagePasser** module transmits messages sent to it from GameLogic and GameManagement to other nodes in the system. MessagePasser keeps track of the socket connections between the other nodes.

4.2 GAME SERVER:

The game server keeps tracks of all the ongoing games and the list of their players. When a new player wants to join a game, it gets a game from the game server and attempts to join that game. When all existing games are full, a node can start a game of its own.



4.3. Bootstrap protocol

When a node wants to start a new game, it contacts the central server which contains the list of all ongoing games and their players. Specifically, the server contains the Game ID, list of players that are a part of the game and their IP and ports for all the ongoing games. The server sends out this information for the requesting node.

Starting a new game

If the node that wants to join a game discovers that there are no ongoing games or that all the ongoing games are at full capacity (maximum 8 players), it starts a new game of its own. The new node will make use of the {set, AddGame} and {set, AddPlayer} messages to communicate with the server to start a new game.

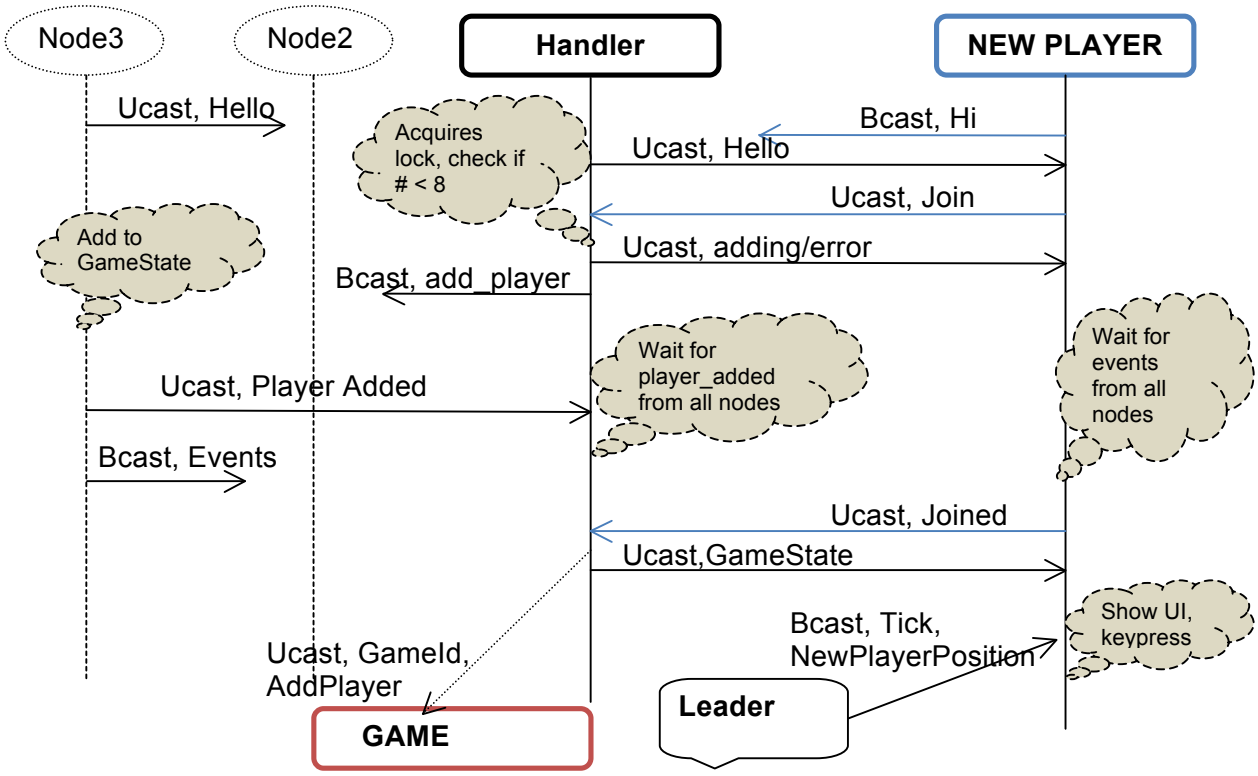
Adding a new player to an ongoing game

When a new player wants to join the game, the new player broadcasts a “Hi” message to all the nodes playing the game (Using the list of players from the game server). All the existing nodes in that game reply to the new node. The new node now chooses to communicate with the node that it got the fastest response from (i.e. its nearest neighbor). It sends the nearest neighbor a “Hello” message. The nearest neighbor is called the “handler”.

The “handler” acquires a distributed mutex first. This is done to ensure that only one current player can add a new player to the ongoing game. “Handler” then decides to add the new player, if the existing number of players is not already 8. It sends a “adding/error”unicast message to the new node. The handler sends out a broadcast message “add player” to all other nodes and waits for them to respond with “player added” which means that they have added

the new node to their game state (message passer's list of nodes, list of snakes, lock's list, leader protocol list etc). The new node meanwhile, waits for events from all the nodes, and once this is done, it sends out a "joined" message to its handler. The handler sends the game state to the new node and asks the game server to add the new player to the list of nodes in that game.

From now on, the new node shall receive all the events and the ticks form the leader. When the New player presses an arrow key; the leader sends out its starting position with the next clock tick. After this the new player's snake is seen on the UI and it can start playing the game.



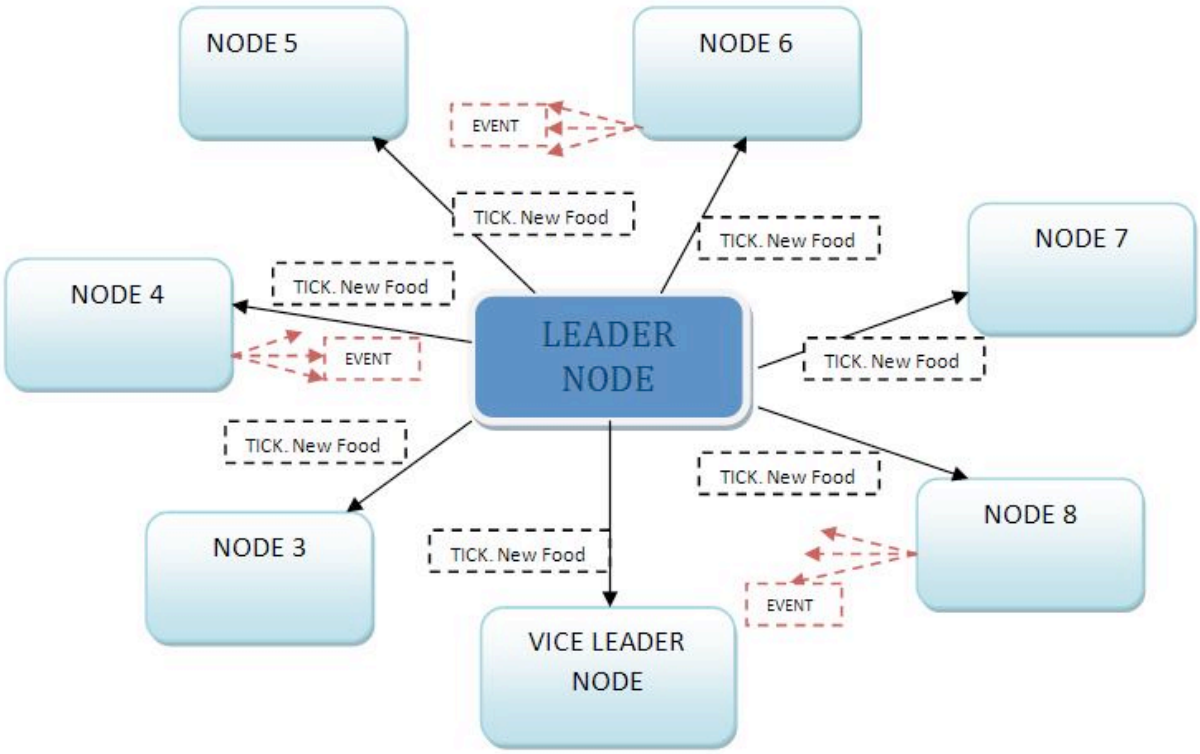
4.4. Leader Protocol

For the distributed snake game, all the players are required to send moves (key presses) to each other. All the nodes independently calculate every snake's position and draw them on the UI. But when should all the nodes process the key presses from the other nodes? When should all the nodes send out their events/key presses to the other nodes? Could they use their own physical clocks for this? The answer is "No", because this has to be synchronized. For achieving this, we designed the "Leader protocol".

We have a local clock ticking at each node, which needs to be synchronized with the other player nodes in order to ensure real-time game play experience. We thus introduce a Leader

node in the group. This Leader node will broadcast a clock TICK after '1 logical timestamp' to all the other nodes. When each node receives this 'master clock' TICK, they will broadcast the local event updates to the other nodes in the group. Each node then processes the events that occurred at the last clock TICK, and further update their local game state.

Our snake game has food for the snakes appearing randomly at random intervals. The food generation is also done by the leader. (This was done because the nodes cannot independently calculate the food positions using random number generators. If they do this, all nodes will have foods at different positions, because random number generator depends on the initial seed and the number of times it has been called).



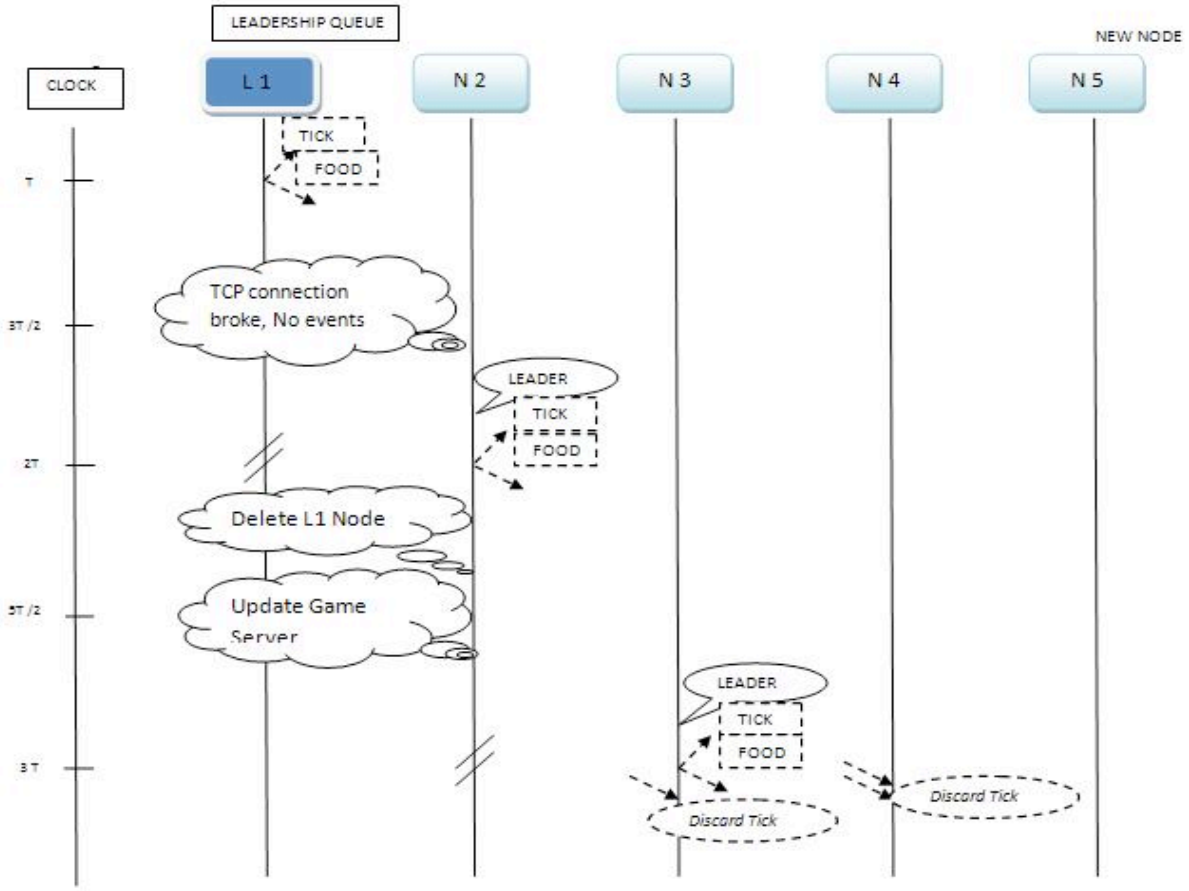
4.5. Game Communication Protocol

As new players are added to the game, each player needs to update its local game state according to the event updates received from every other player involved in the game. These event updates are sent to the other player nodes after a predetermined time quantum (logical timestamp). At each timestamp, there could be multiple events occurring at one node. We define an Event Queue to handle multiple events at a node.

We have a local clock ticking at each node, which needs to be synchronized with the other player nodes in order to ensure real-time game play experience. We thus introduce a Leader node in the group. This Leader node will broadcast a clock TICK after '1 logical timestamp' to all the other nodes. When each node receives this 'master clock' TICK, they will broadcast the local

event updates to the other nodes in the group. Each node then processes the events that occurred at the last clock TICK, and further update their local game state.

4.6. Fault Tolerance Protocol



The ticks are broadcast by a leader to all the other nodes every time quantum. Whenever the tick (or any indication of a new time quantum) is received by any node, the node advances its local clock.

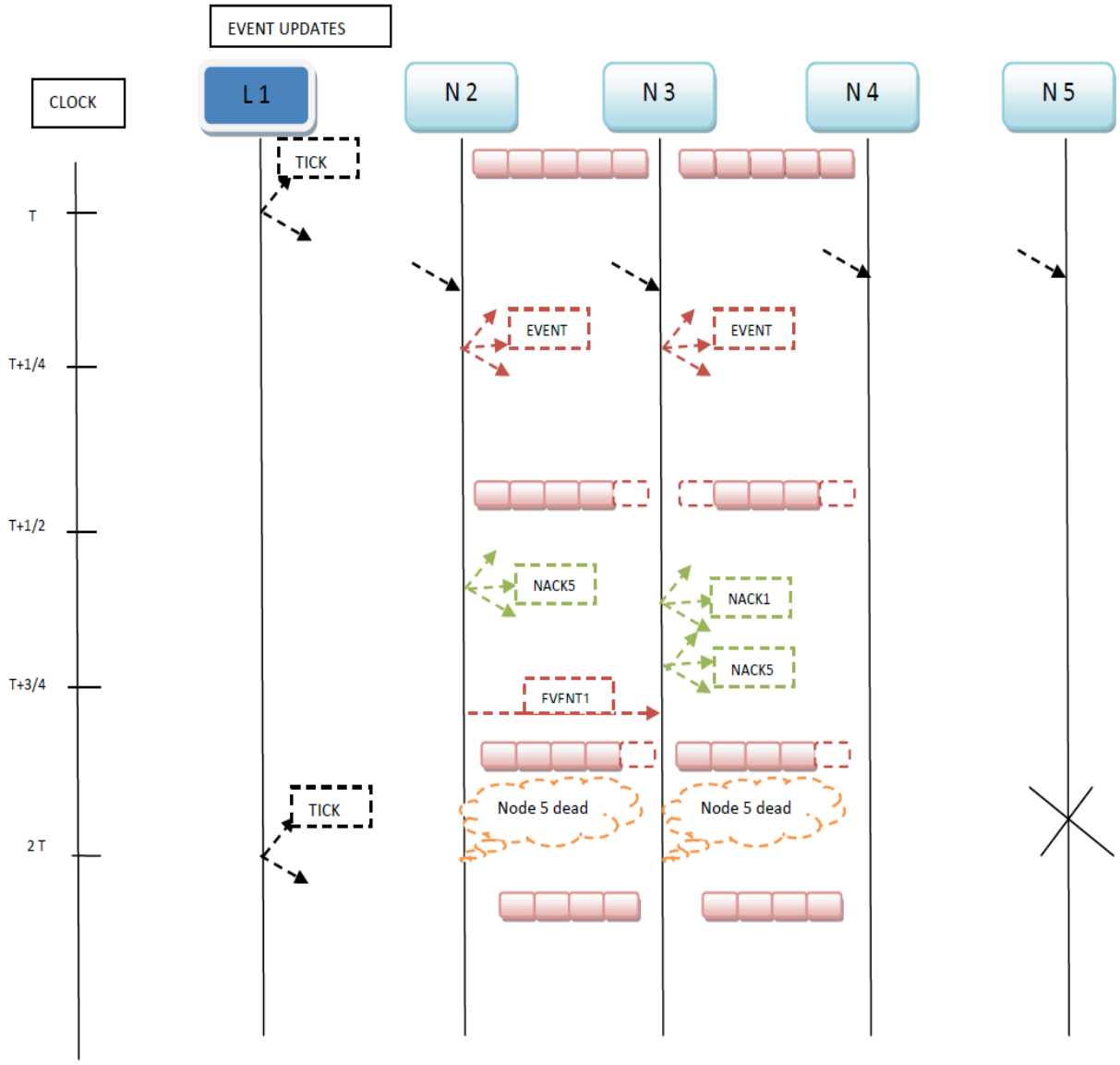
To handle the case where the leader node goes down or the messages from leader node are dropped or delayed due to network issues, we keep a leadership queue which contains the list of nodes in the game. A copy of the leadership queue is maintained at every local node. The first node in the queue is given highest priority followed by the other nodes in the leadership queue for assuming leadership in case of any issues with the current leader node. A leader node will join at the end of the leadership queue if a node in the queue assumes leadership in case of the leader node not performing properly.

Exceptional cases

Each receipt of a tick triggers a local timer on each node for a period T which is little more than the time quantum plus the network delay. This protocol has the capability of handling following exceptional cases:

- If ticks are not received or events from the leader are not received, the first node N in the leadership queue will assume leadership and start broadcasting ticks at the end of time period T plus time out. The new leader shall delete the old leader from the game by requesting the game server to delete the old leader from the list of nodes and also informs all other nodes to remove the leader from their data structures.

- Receipt of duplicate ticks at a node:
  Any node will advance its local clock upon any indication of the new time quantum. Specifically, the node will advance its local time when it receives a new tick or a new ALIVE message (from current leader or a new leader) or any events message (Ek) from any other node for the new time period. The node will ignore any other indications of the new time quantum thereafter. This ensures that even if there are multiple ticks received at a node from two different leaders (e.g. old leader and new leader), duplicate ticks will be ignored and the node will keep on functioning properly.



In the case of lost event updates

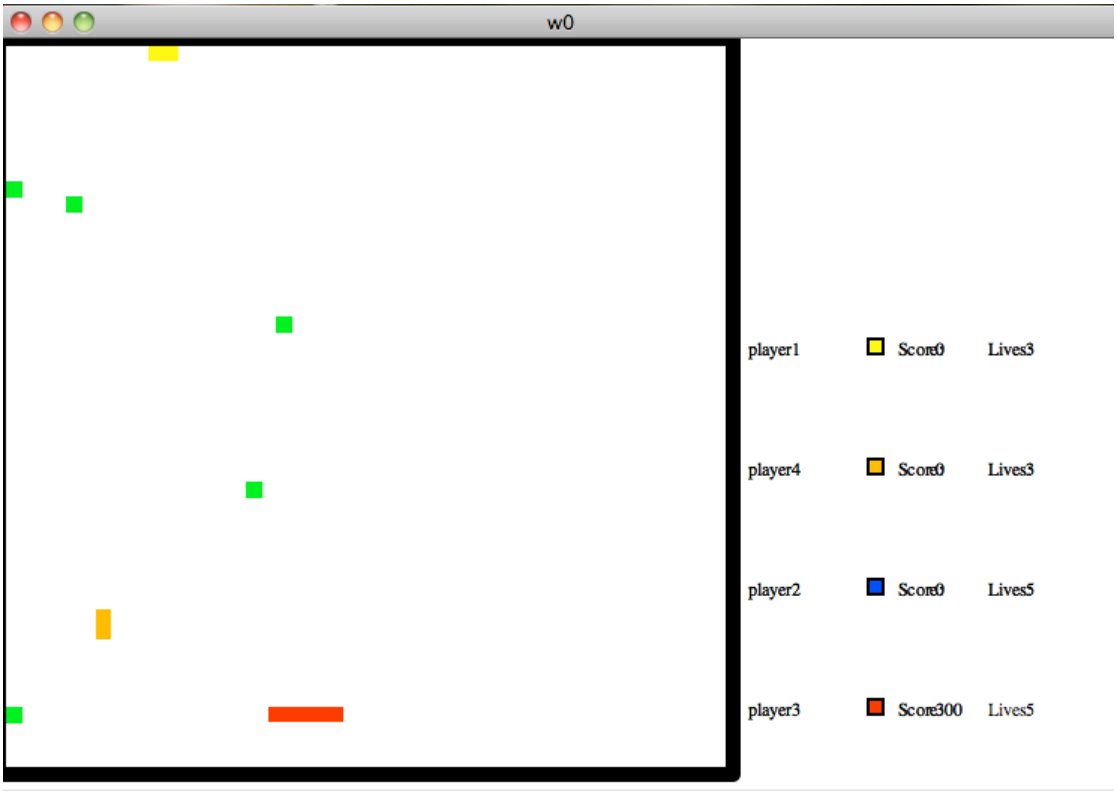
Every node sends the updates about the events occurring during a time quantum at the local node (e.g. key-press events at the local node) at the end of the time quantum to every other node.

When a node doesn't receive updates from some node, but its tcp sockets to that particular node are not broken, we know that the events will come after some time and the reason that event has not reached this node is the network delay. So, this node will stop sending its events to other nodes and ignores the ticks from the leader. When the leader doesn't receive event updates from any of the nodes, it keeps the clock tick count the same without incrementing it. Eventually, the node receives the event and starts sending events to all other nodes. This way the game is consistent and made tolerant to the delays in the network.

If a node's tcp socket is broken, it knows that the other node has died. So, it delivers this information to all the other nodes, which remove them from their data structures and also asks the game server to delete that node form the game list.

It should be noted here that this entire information discovery procedure should be over within a period which is less than the game time quantum T. This ensures that the nodes will discover information about failure of any node within one time quantum. It also ensures that the game state data structures at each node remain consistent.

4.7. User Interface



Contribution

Description	Assigned
Project Selection	Team
Project Requirements	Team
Project Design - bootstrap - protocol - game logic	Team
Design Document	Team
Ramp up on Erlang	Team
Setting up GIT code repository	Osei
Snake GUI	Rama
Message Passer	Team
Game Logic for single player	Osei & Aditi
Boot strap	Manan, Osei
Testing for initial demo	Team
Initial Demo - Single player/leader	Team
Finishing Game Management, Bootstrap	Osei, Manan
Game Server	Osei, Manan
Leader Fault Tolerance	Rama, Aditi
Missing events Fault Tolerance	Rama, Aditi
Integrating & Testing for final demo	Team
Final Presentation	Team
Final demo	Team
Project Report	Team