A Real-Time Multiplayer ‘Wolfpack’ Peer-to-Peer Game with Distributed State

Ying Ying Choi (j1l0b), Lisa Gagno (b1l0b), Ryan Lim (t9i0b), Kirbee Parsons (r8h0b)

Introduction

A popular topology for real-time multiplayer games involves all players communicating to a single server. However, this centralized communication is often the performance bottleneck in the network. Additionally, when the game server fails, all clients within the game fail. To circumvent this, we propose a distributed system in which players exchange game state information between each other. With this information being communicated between players instead of relying solely on the server, a server failure will not impact the current players of the game. Furthermore, this forgoes the need for a high-performance centralized server, which significantly saves on cost.

Wolfpack Game

The real-time multiplayer game that we will be creating is based on the 2-D “Wolfpack” game described by Google’s *DeepMind*[1]. The game involves two classes of players - the *wolves*, and the *prey*. At any one time, there is only one *prey* player in the game - and the single object of the game for each *wolf* is to catch the *prey*. Upon catching the *prey* – by occupying the same space as it – the *wolf* is given a fixed reward in the form of points. Both the game server and the player nodes will be hosted and run on Microsoft Azure.

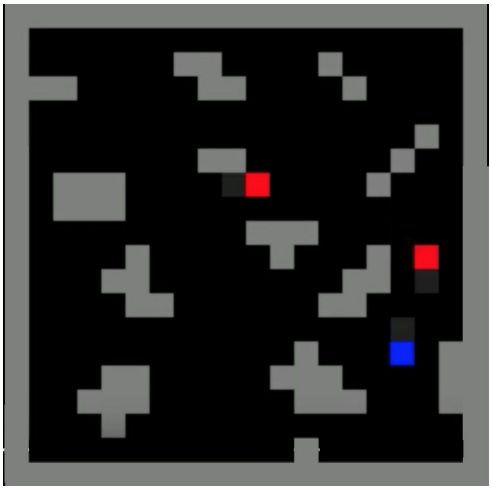

***Figure 1**: Screenshot of DeepMind gameplay showing blue “prey” player being hunted by two red “wolf” players*

The “Wolfpack” game was chosen for its:

1) Ability to support an arbitrary number of players
2) Simplicity of movement, game states and user states, to allow us to focus on the distributed systems aspects of the game

[1] https://deepmind.com/blog/understanding-agent-cooperation/

Considerations

Using a peer-to-peer architecture for a real-time multiplayer game, we eliminate the server-player fate-sharing if the server fails in a centralized system. However, this introduces problems like:

Synchronization

This issue is not isolated to a peer-to-peer game – as with any real-time multiplayer game, the synchronization of game state between clients is necessary for a good user experience. Having other players jump wildly around the map is undesirable, and so is “playing” in a game state that is far behind the global state - in fact, these issues will likely cause gameplay to break down rapidly for at least one, and possibly all, users. Thus, the main issue this project will face is ensuring a consistent game state for all nodes, and dealing with issues which may cause an inconsistent game state (see “Dealing With Slow/Failed Clients”). We will attempt to maintain a synchronized game state across all clients who are currently connected to the network, and gracefully deal with those nodes that are unable to maintain a synchronized state. This will be done using logical clocks. If there is a discrepancy between this node’s logical clock versus multiple incoming nodes’ logical clocks, then this node will realize that its game state might not be the most current and try to remedy this situation.

Validation of Moves

Since communication of game state will be sent from peer-to-peer without a central server to validate moves and disseminate only valid moves, player nodes cannot blindly accept the moves sent by other players. Though there is the potential for cheating (See “Preventing Dishonest Nodes”), there is also the potential to attempt to perform invalid moves in earnest, possibly due to performance or communication issues. Thus, validation of moves will be performed both before they are broadcast to other nodes, and upon receiving nodes from other players, and invalid moves will not be allowed.

Preventing Dishonest Nodes

Furthermore, player nodes can attempt to cheat the game by any of:

1) Reporting to move to an invalid space in order to give themselves an advantage by
    a) Making a move that is valid in “step” but through a barrier
    b) Making a move that is invalid, so “teleporting”
2) Reporting a score that is greater than the score they have actually received by capturing prey
3) Reporting they have caught the prey when no such capture has occurred (they have not occupied the same space)
4) Misreporting other nodes information
5) “Lookahead” cheating; delaying transmitting their moves until they know all other nodes’ moves to give them an advantage

We will attempt to prevent nodes from succeeding with dishonest practices in all 5 cases.

To prevent threats 1, 2, and 3, player nodes will independently verify the information they receive from other nodes. In order to minimize the chance that moves are misinterpreted as illegal due to a dropped

packet, all moves will be sent in duplicate. Depending on the observed incidence of packet issues, further remedies will be taken to minimize the impact of dropped packets. Possible solutions include requesting game state from other nodes, or counting “strikes” against a player for invalid moves over a certain period of time.

To prevent threat 4, player nodes will provide their public key when connecting to other nodes to enter the game. Furthermore, player nodes will encrypt movement information with their private key before sending it to other nodes. These nodes will now be able to check that the movement information was sent by the correct player, by decrypting it with the appropriate public key.

To prevent threat 5, a *lockstep protocol*[2] will be implemented. Nodes will need to broadcast a hash of their move as a “commitment” to all other nodes before they receive information on any other nodes’ movement. When movement is received, its hash is checked against the commitment hash to ascertain the node has made the move they committed to before it knew about other nodes’ movements. If the movement matches the commitment, the move will be accepted (provided it is legal); if not, it will be ignored.

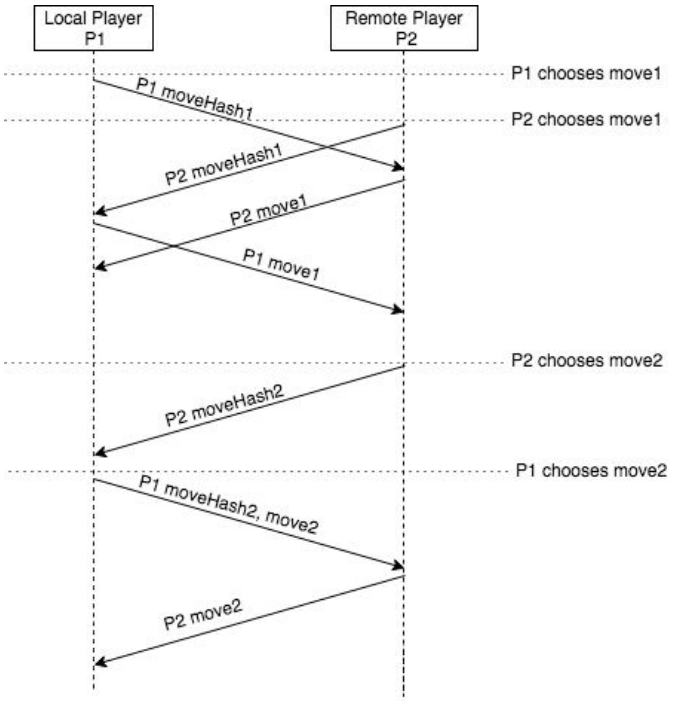

***Figure 2**: Simplified lockstep protocol time-space diagram showing two nodes withholding move details until commitments received from the other node*

Dealing With Slow/Failed Clients

A real-time game must proceed at a roughly constant frame rate; that is, the game should not exhibit regular lag. To do so would be to provide a poor player experience and thus a bad game. However, in a peer-to-peer network, waiting for all other player nodes to broadcast their next state would certainly lead to significant lag if one or more players had a bad connection. Even worse, if the player failed, the

[2] https://en.wikipedia.org/wiki/Lockstep_protocol

remaining nodes would be waiting indefinitely (if no timeout was implemented). Similarly, we will have to handle the case where a node momentarily fails and rejoins. If players are kicked off for a single missed heartbeat or dropped packet, it will not be a good experience.

We will therefore aim to provide good player experience throughout gameplay for those nodes with acceptable connections by gracefully dealing with both slow and failing player nodes. A “good” experience is defined as gameplay that is no more than 10% slower than the expected “ideal” gamestate (which will a fixed time goal per ‘tick’ that is to be determined during implementation), and remove nodes from the game that jeopardize this goal and a descriptive message describing the reason for game removal.

Design

Global Server

The global server will be accessed by players during the registration protocol. The server will do the following things:

- During registration, the player wanting to join the game will send the global server its connection information. This information will then be stored on the server.
- In response, the global server will then provide the player with other players’ connection details as well as a unique number to be used to identify the player within the system.
- The global server will also provide an initial game state to players joining the game.

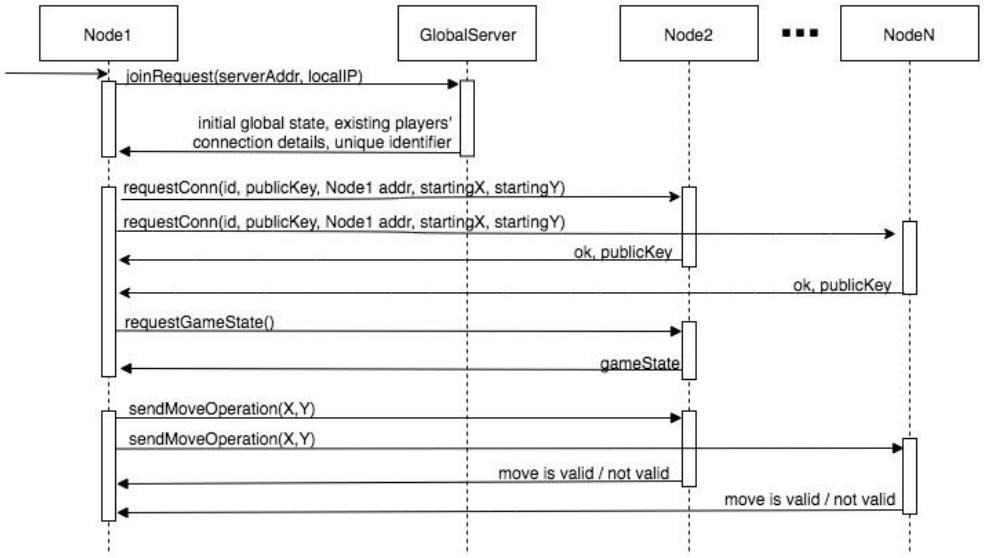

***Figure 3**: Time-space diagram showing the series of calls needed for Node1 to initiate gameplay*

Player Node (Communication Interface + Player App)

Communication Interface

The communication interface is associated with at most one player app at any given time. It will run on the same node as the player app, and exists mainly to separate the networking code from the game code. It provides the following services:

- The communication interface will facilitate communication to and from server and other players. For the server, this includes the authentication/registration protocol, heartbeat program, and a time synchronization protocol. For the other players, it will broadcast game states as the game progresses.
- It will also verify moves that its player makes, based on the initial game state information from the server, and ensures it does not make any invalid moves based on the most recent game state.

Player App

The application will access the initial game state and the following game state information from the interface to render the game on screen using Pixel, a 2D Golang game library[3]. It will also transmit the player's input to the app.

System Invariants

- A player's position must always be within the boundaries of the grid,
- A player's position cannot be the same as a filled in space on the grid (a wall),
- A player can share its position on the grid with the prey dot,
- A player's current position must be no more than one "step" (game environment unit) away from their previous position
- A given player's score is always equal to the sum of the number of times they captured the prey times the reward value stored in the server

Communication

Initial Game Configuration (Global Server to Player Node)

The initial game configuration will be sent by the global server to the players joining the game. It includes environment settings and logic. It will contain:

- The spatial dimensions and boundaries of the game
- The amount of points that a player gets when the prey dot is captured.

Game State (Player Node to Player Node)

After the initial game state sent by the server, the players will be responsible for sending the game states between themselves via their communication interfaces as the game progresses. Based on the

[3] https://github.com/faiface/pixel

ARM Game with Distributed States by Glen Berseth and Ravjot Singh[4], we will be transmitting this information between players:

- The player's unique number,
- The location of this player's dot on the grid,
- The timestamp of the clock for the player,
- The last updated time the communication interface has received a message from the player,
- The highest score of the player.

Network Topology

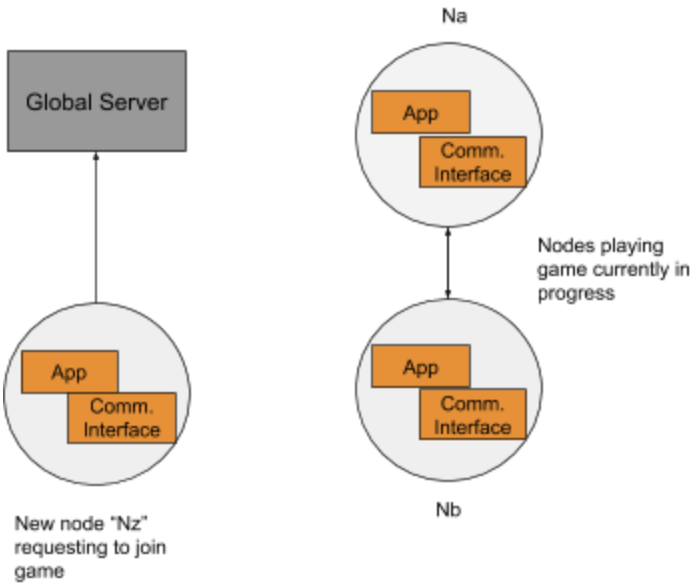

***Figure 4**: Left: A prospective player node "Nz" makes a request to the server in order to join the game. Right: Player nodes "Na" and "Nb" participate in gameplay by sharing their moves directly.*

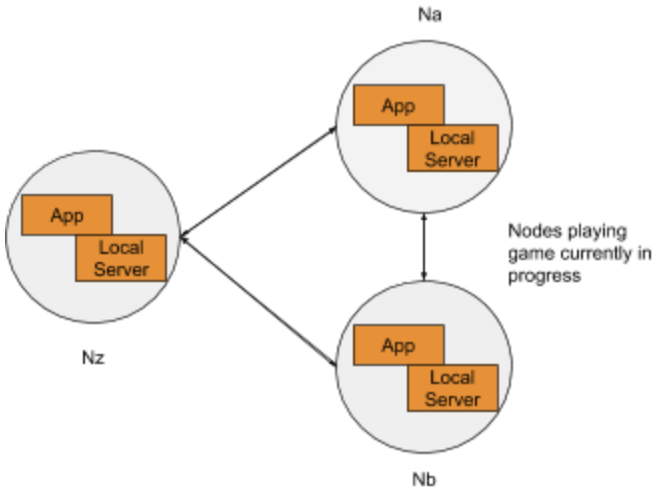

***Figure 5**: "Nz" has successfully joined the game by connecting to the two nodes, "Na" and "Nb", already in gameplay.*

[4] http://bit.ly/2oAKxtn

Initially, nodes connect to the global server in order to be connected with nodes already in gameplay. Upon successful request, the server will provide them with the game configuration information outlined above, as well as information required to connect to player nodes already in game.

Because speed is a factor in this game, we will need to determine the best way of achieving the maximum speed while allowing as many players as possible to join the game. We will try two approaches:

1. Limit the number of players that can connect to each other to a reasonable amount (<10) and create a fully connected network
2. Allow only a subset of players connect to each other and implement a gossip or communication protocol to send and receive events via an Adaptive Gossip Protocol[5], which, in brief:
    - Propagates events to a randomly selected subset of nodes
        - Randomly selected subset is based on a random factor, but also past communication protocols
    - Nodes which send updates when updates are expected are valued higher than nodes which do not
    - Might be possible solution for nodes which “cheat” by eventually excluding them from gossip

Both these approaches have drawbacks. The first severely limits the type of gameplay that can occur in a peer-to-peer system. In a fully connected system, the number of connections increases by N-1 for each node added to the system since it must connect to each of the other nodes, where N is the total number of nodes. This can very quickly overwhelm the bandwidth of most systems as N increases. However, the second approach would not guarantee that every other node would get an update immediately, and would take multiple hops for each update in the network. This would slow down the game. We will try both approaches and determine which will work best for our game.

Assumptions

- Fewer than half (N/2) player nodes will fail at any one time. If we lose more than half of the nodes we lose consensus which we will use for detecting and eliminating malicious nodes from the game and rectifying game state in the case of dropped communication.
- The global server will never fail permanently; it will eventually recover after any failures.

[5] https://arxiv.org/pdf/1102.0720.pdf

SWOT Analysis

Strengths - Team members have intensive Golang programming language in last 2 months - All team members are excited about the challenge of creating a game - Can apply what we learned from Assignment 2 / Project 1	**Weaknesses** - All 4 team members have only been programming in Go since January - No team members have any game development experience - Difficulty of real-time state sharing may be drastically underestimated due to inexperience - Project timeline is concurrent with deliverables in other courses
Opportunities - Ivan and course team for help - Pixel game library has current commit history and extensive documentation - Real-time multiplayer games are an interesting, ongoing distributed systems problem - Lots of ways to iterate on the game to make it more complex depending on time	**Threats** - Success of the project is highly dependent being able to integrate multiplayer capabilities easily into Pixel - Pixel is still under heavy development - Macbooks may not compatible with CPSC 416 development rigour (#RIP)

Timeline

Week of	Milestones	
March 9	- Fine-tune and submit final proposal	~ Completed by: Everyone

March 12	- Integrate game library into project
	- Develop a simple UI
	- Develop single-player game version
	  - (just a player, no prey)
	- Set up global server
	  - (registration, initial game state)
	- Set up communication interface
	  - (heartbeat, node connections, time synchro)
	- Build test framework
	- Set up scripts for Azure VMs
	- Handle client failures
	  - ongoing process
		~ Completed by: Ryan
		~ Completed by: Ryan
		~ Completed by: Kirbee
		~ Completed by: Lisa
		~ Completed by: Ying Ying
		~ Completed by: Ying Ying
		~ Completed by: Ryan
		~ Completed by: Everyone

March 19	- Set up meeting with project TA (March 23) - Add multiplayer capabilities - (other players, and prey) - Integrate state transfer/sharing into players - Move Validations	~ Completed by: Ying Ying ~ Completed by: Ryan / Kirbee ~ Completed by: Kirbee ~ Completed by: Ying Ying

	- Encryption of Movement - Lockstep Protocol - Develop “sybils” apps	~ Completed by: Lisa ~ Completed by: Ryan ~ Completed by: Lisa
March 26	- Begin final report - Write/run tests for synchronized player states - Integrate anti-cheating checks	~ Completed by: Everyone ~ Completed by: Ryan ~ Completed by: Ying Ying
April 2 - April 6 (deadline)	- Additional performance and stress testing - Bug fixes - Review and document code - Complete final report - Prepare demo	~ Completed by: Everyone ~ Completed by: Everyone ~ Completed by: Everyone ~ Completed by: Everyone ~ Completed by: Everyone