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

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.

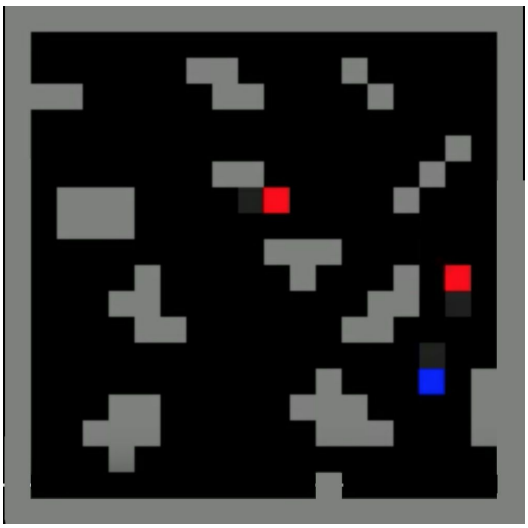

Figure 1*: Screenshot showing blue “prey” player being hunted by two red “wolf” players*

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

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

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.

Validation of Moves

Since communication of game state will be sent from peer-to-peer without a central server to validate them, 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)

We will attempt to prevent nodes from succeeding with dishonest practices in all 3 cases.

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 remaining nodes would be waiting indefinitely (if no timeout was implemented). 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.

Assumptions

- No more than (N/2) - 1 player nodes will fail at any one time
- The server will send an identical/consistent settings to all players.

Design

Nodes

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.
- For a game that has reached the winning state, it will receive information from one of the nodes which will then prompt a broadcast of the initial game state information to all registered players for a game restart.

Player Node (Local Server + Player Interface)

Local Server (Stateless)

The local server is associated with at most one player at any given time. It will run on the same node as the Player App/Interface, and exists mostly to separate the networking code from the game code. It provides the following services:

- The local server 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 / Interface

The application will access the initial game state and the following game state information from the local server to render the game on screen. It will also transmit the player's input to the local 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:
    - A S x S grid with pre-determined "walls" - spaces that are filled in,
    - A player cannot move their dot outside of the grid,
    - A player cannot move into a space on the grid if it is filled in (a wall),
    - A player cannot move into a space on the grid occupied by another player,
    - A player can move into a space on the grid occupied by the prey dot
- The reward:
    - 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 local servers as the game progresses. Based on the

ARM Game with Distributed States by Glen Berseth and Ravjot Singh[2], 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 local server has received a message from the player,
- The highest score of the player.

Network Topology

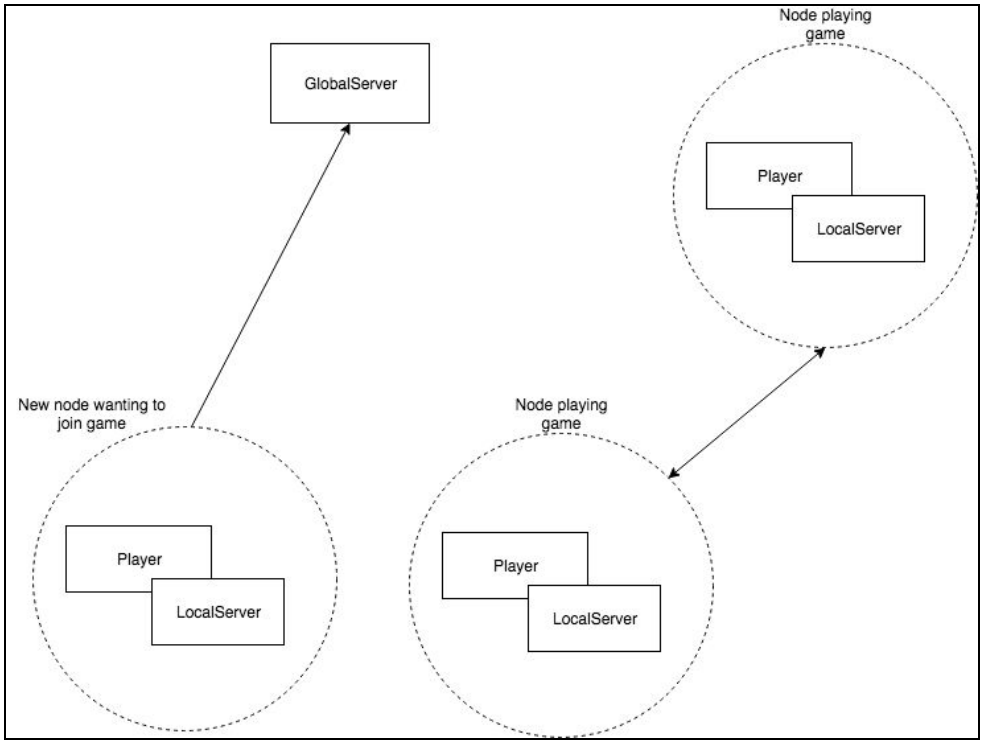

***Figure 2**: A prospective player node "Nz" makes a request to the server in order to join the game.*

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

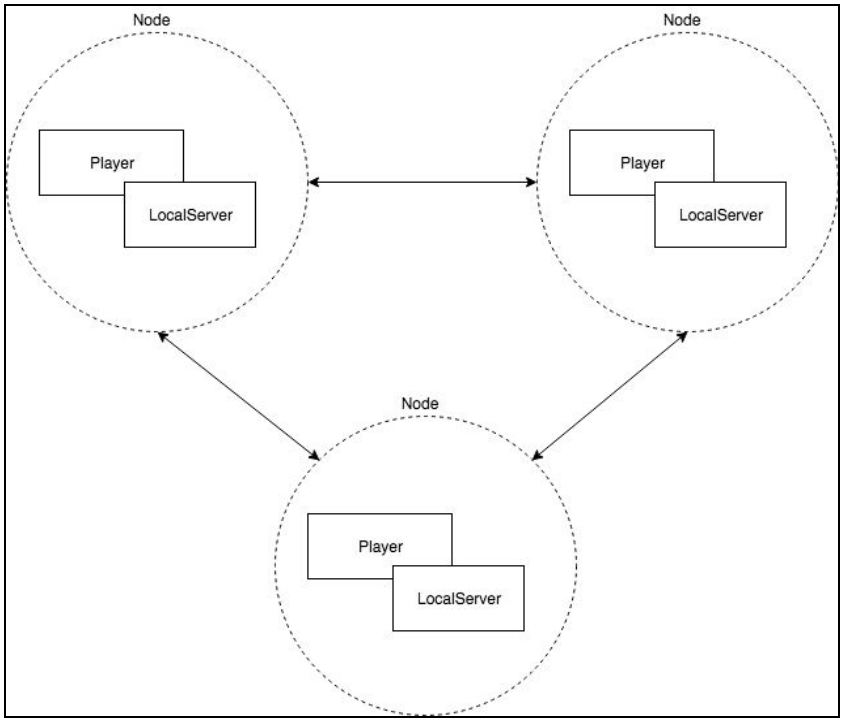

***Figure 3**: "Nz" has successfully joined the game by connecting to the two nodes already in gameplay.*

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
    - https://arxiv.org/pdf/1102.0720.pdf
    - https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/donnybrook.pdf

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 amount of data sent increases by O(n^2) for each node added to the system. This will very quickly overwhelm the upload bandwidth of most systems. 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.

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 important deliverables in other coursework for project team
Opportunities - Ivan and course team for help - Game development libraries with good documentation exist for Go - 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 on finding a game development library that is easy to use - Many Golang game libraries are 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
March 12	- Integrate game library into project - Build test framework - Develop single-player game version - Setup server registration protocol - Develop a simple UI

March 19	- Set up meeting with project TA - Add multiplayer capabilities - Set up client heartbeat - Integrate state transfer/sharing into players
March 26	- Begin final report - Handle client failures - Write and run tests for synchronized player states - Integrate anti-cheating checks
April 2	- Additional performance and stress testing - Bug fixes - Review and document code - Complete final report - Prepare demo