Wolfpack: You Don't Need an Authoritative Server for an Online Multiplayer Game!

(But You Probably Should Just Have One)

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

Abstract

In order to avoid fate-sharing and the costs associated with a dedicated server, which is the norm in the majority of modern online multiplayer games[1], we built a "Wolfpack" game which communicates game movements completely over a peer-to-peer network. While developing "Wolfpack" we encountered numerous difficulties; these include troubles related to testing real-time systems which rely on player input, issues related to balancing speed vs. consistency and security, and problems related to our lack of experience developing real-time systems and games. In the end, we have produced a slow-but-consistent version of Wolfpack, and have experienced first-hand why so many online games employ a client-server model.

Problem

In the vast majority of online multiplayer games, a client-server model is used, with individual players communicating solely through a centralized server to send information about their gameplay and receive others' gameplay information. In this model, the server is the authority on the current game state[2]. However, this topology causes every player to fate-share with the server – that is, when the server dies, so will the game.

Additionally, servers can be costly – they may require an administrative entity to support and operate. Typically, indie game developers are reliant on third parties to host the central server such as Amazon Web Services or Microsoft Azure; the costs for these services can reach $12,000 monthly[3]. These issues make it prohibitively expensive for many smaller game developers to provide multiplayer aspects in their games. Small-time computer and mobile game developers that may not be expecting large returns on their games may not want to commit to the upkeep of servers over long periods of time.

Solution

A solution to running a central server is to use a peer-to-peer network to coordinate gameplay. Because we were constrained by the fact a part of the system had to run on Microsoft Azure, we decided to run the logical components of the player nodes on Azure, with the graphical component running locally (more on this later).

Although the system uses a server for the players to find and connect to other players, this server is lightweight and does not do any centralized processing. This will be discussed in detail later.

The game our system is based off of is a version of the 2-D "Wolfpack" game described by Google's *DeepMind*[4].

---

[1] https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/
[2] https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
[3] https://insights.dice.com/2016/02/23/indie-game-devs-face-the-multiplayer-challenge/
[4] https://deepmind.com/blog/understanding-agent-cooperation/

Design

Assumptions

- The server will never fail-stop; it will eventually recover
- The prey node will never fail and is trustworthy
- The network is reliable
- No more than n / 2 nodes will fail at any given time, where n is the number of players in the system, to allow us to use a majority to progress the game state

The Game

The game involves human-controlled players - the *wolves*, and an AI “player”, the *prey*. At any one time, there is only one *prey* player in the game. The single object of the game for each *wolf* is to catch the *prey*. When a player captures the prey by occupying the same space as it, that player is given a fixed reward in the form of points. Players use the four directional keys to move their sprite.

Figure 1: Screenshot of our “Wolfpack” game, showing the current player (wolf sprite) converging on the prey (rabbit sprite) along with other players (red sprites). The scoreboard is shown on the right.

Topology

Initially, all game nodes were intended to reside on Azure. However, we also intended to use Pixel[5], a 2D Golang game library, which provides a graphical display. Early on during the development process, we tried running Pixel on Azure with graphical forwarding via the `ssh -X` option. Unsurprisingly, we found the gameplay much too slow to be acceptable; this resulted in us segregating the game’s logic and its rendering completely in order to run them on separate nodes.

The resulting topology is illustrated below, with each player “node” being separated into two separate nodes: the Pixel (GUI) node and the logic node. The player’s logic node runs on Azure, and connects directly to all other logic nodes in a fully-connected network. The Pixel node connects to a logic node and forwards player moves to the logic node. This is discussed in further detail below.

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

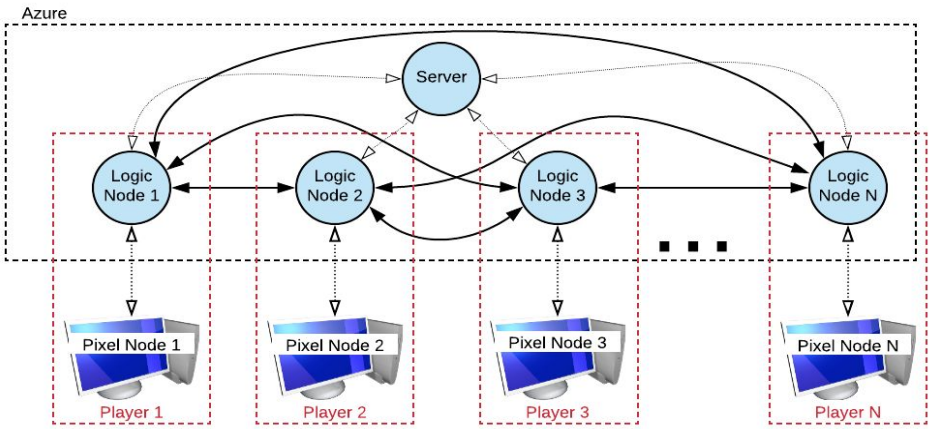

Figure 2: Topology of the network. Dotted lines indicate TCP connections between nodes, solid lines denote UDP connections.

Nodes

As noted above, the network has several node types. Each has a specific role in the system.

Server

The server, which resides on Azure, is responsible for connecting joining nodes to nodes already in gameplay. It also stores and communicates the game configuration to joining nodes, such as the location of walls within the game, the size of the board, and the number of points gained by capturing the prey.

Player Node

The player “node”, is actually comprised of two nodes: the logic node, which resides on Azure, and the Pixel node, which runs on the player’s local machine.

Logic Node

The logic node is responsible for consistency and security checks, as well as communication between itself and all other logic nodes. Additionally, it receives player moves from its associated Pixel node, which it validates and forwards to other logic nodes, as well as regularly forwards game states to the Pixel node for rendering.

Pixel Node

The Pixel node has two responsibilities: sending player keystrokes to its associated logic node and rendering the most recent game state received from the logic node using Pixel game library.

Prey Node

The prey node is a special logic node that does not have an associated Pixel node. That is, it does not have an associated human player; it moves to a random adjacent position that is valid every 250 ms without human input and does not require rendering abilities. Originally, we intended the prey node role to be assigned to one of the human players in play; however, we decided that it put the game at risk for both cheating, by having the prey node cooperating with a player node, as well as for game disruption by a malicious prey node that continually exits the game.

Communication

Logic Node to Server

The logic node uses two RPC methods communicate with the server over TCP. The first initiates the connection to the server. The server responds with the game settings (dimensions, location of walls, points

per prey capture), a heartbeat interval, and the IP addresses and public keys of all logic nodes already in-game. The server also assigns a new unique string identifier to the joining logic node. This id is included in all of the outgoing messages sent by the logic node and is how other players will identify this player. The second RPC method is run in a goroutine and sends a heartbeat to the server at a predetermined interval set in the game config; which ensures the server knows this node is alive so will continue to connect joining players to it.

Logic Node to Logic Node (Including Prey Node)

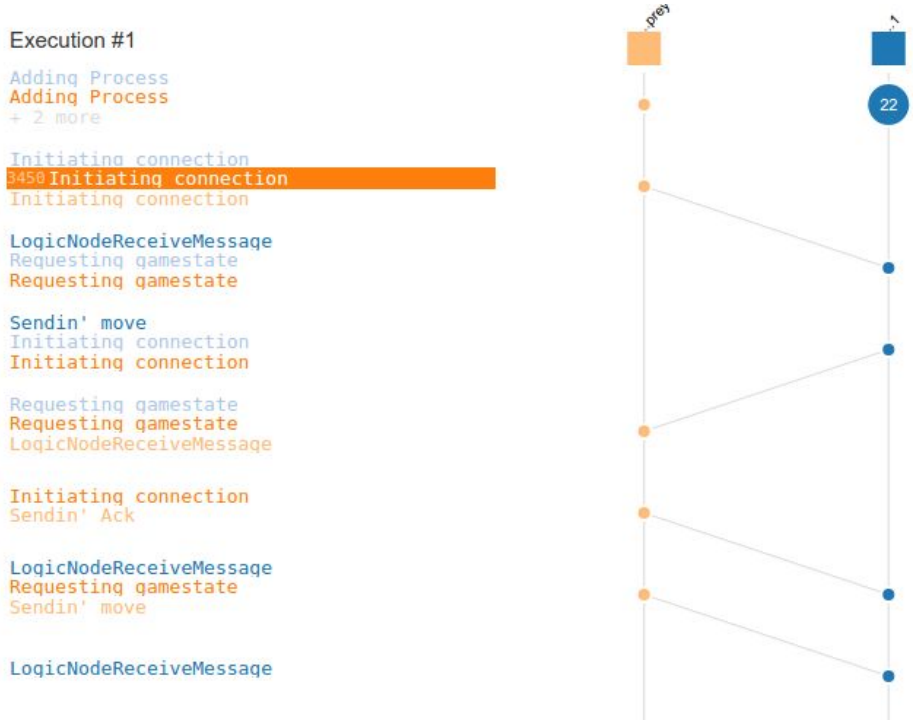

Figure 3: Player node 1 and prey node joining the game

The logic node sends messages to other logic nodes using a UDP connection. We created a message struct which accommodates different message types:

- connect (request for receiving node to connect to sending node),
- gamestateReq (request for a game state),
- gameState (message containing a gamestate),
- move (message containing a move),
- captured (notification of prey capture),
- ack (move acknowledgement).

When a message is received by a logic node, we can use its type to direct it to the right handler to process the request. If the request requires a response, the logic node can use the id of the sending node included in the message to issue a response back.

Figure 3 shows a time space diagram of the initial nodes joining the game. Node 1 registers with the server and gets a game state. The prey then joins and requests the game state from node 1. Node 1 then can start sending moves to the prey and vice versa.

If another node is in the game, then the game is “in progress”. A node joining an in-progress game will need to request the current game state. The joining node requests the game state from each of the other nodes it is connecting to until it receives a game state.

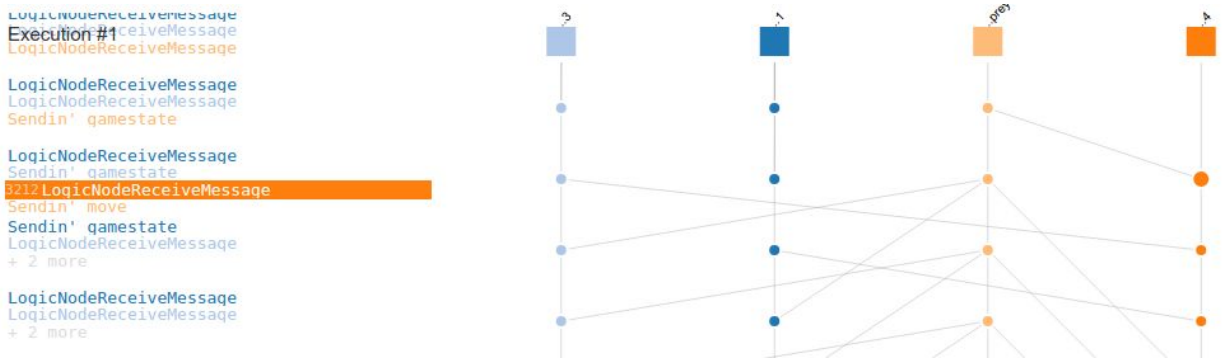

Figure 4: Joining Nodes Game In Progress - Node 4 joins the in-progress game, and requests a game state from prey, node 1 and 3. The node will accept the first game state it receives.

Logic Node to Pixel Node

On startup, the Pixel node establishes a TCP connection to an already-running logic node. On connection, the Pixel node waits to be sent the game configuration from the server; it waits until it receives this to render the GUI. The Pixel node uses this TCP connection to send keystroke information to the logic node. The logic node also uses this connection to send a game state for rendering to the Pixel node.

Handling Failures

Server Failure

Because the server is not involved in the communication of game state, the server can fail without interrupting the current game. Additionally, logic nodes intermittently attempt to reconnect to the server while it is down, so the server can be restarted at the same address and allow players to join the game again. The only aspect of the game that will be affected on server failure is player discovery. Because of this, our system operates on the assumption that the server will eventually be restarted after a failure.

Logic Node Failure

The server and other logic nodes individually manage their own list of logic nodes in the game. When a logic node fails, the server will remove it from its list of active game nodes within the heartbeat interval, configured in the game settings, as it will cease to send regular heartbeats. To other player nodes, the node that failed will remain stationary until they “strike out” – this occurs when a logic node unsuccessfully tries to send a message to the failed node three times. At this point, the node is considered to have “struck out” and will be removed from the list of other nodes and have its sprite removed from the gameplay area.

Pixel Node Failure

Since a player’s Pixel node is their interface for interaction with the game, we designed the logic node to fate-share with the Pixel node; that is, when a player closes their Pixel GUI or it shuts down unexpectedly, the logic node will also terminate.

Prey Node Failure

Our system makes the assumption that the prey node will never fail. The reasons for assuming this are, firstly, the game would be purposeless without a prey to capture; and secondly, we originally did not plan to have the prey node to not be controlled by a player, so this complexity was not factored into our development plans. Future methods of ensuring game survival may include enabling the server to spin up another prey node in the event of a failure.

Protocols

Since communication of game state is sent peer-to-peer between logic nodes without a central server to validate and disseminate only valid moves, player nodes cannot blindly accept the moves and information sent by other players. Our logic nodes assume that every other logic node (except for the prey node, as it is under our jurisdiction) has the possibility to be malicious. Thus, move commits, moves and scores must be verified before sending and upon receipt.

Move Verification/Dissemination Protocol

A player's own logic node verifies moves received from the Pixel node before sending them. It will not send a message if the player is, for instance, requesting to move into the space occupied by a wall, or off the gamespace. This prevents extra useless messages from being sent over already busy communication channels.

We only allow players to be responsible for their own moves. This is to reduce the size of messages being sent over the network and ensure that nodes do not start misreporting the position of other nodes. When a logic node receives a move from another logic node, it first checks its authenticity by verifying it with the included signature and the stored public key of the node that is associated with the move using the `crypto/ecdsa` package. As a reminder, the server provides joining nodes with the public keys of existing nodes, and connecting nodes will also pass their public key to the node that they want to connect to. Once it passes this check, it updates its game state with the new coordinates for this player and sends them back an ACK.

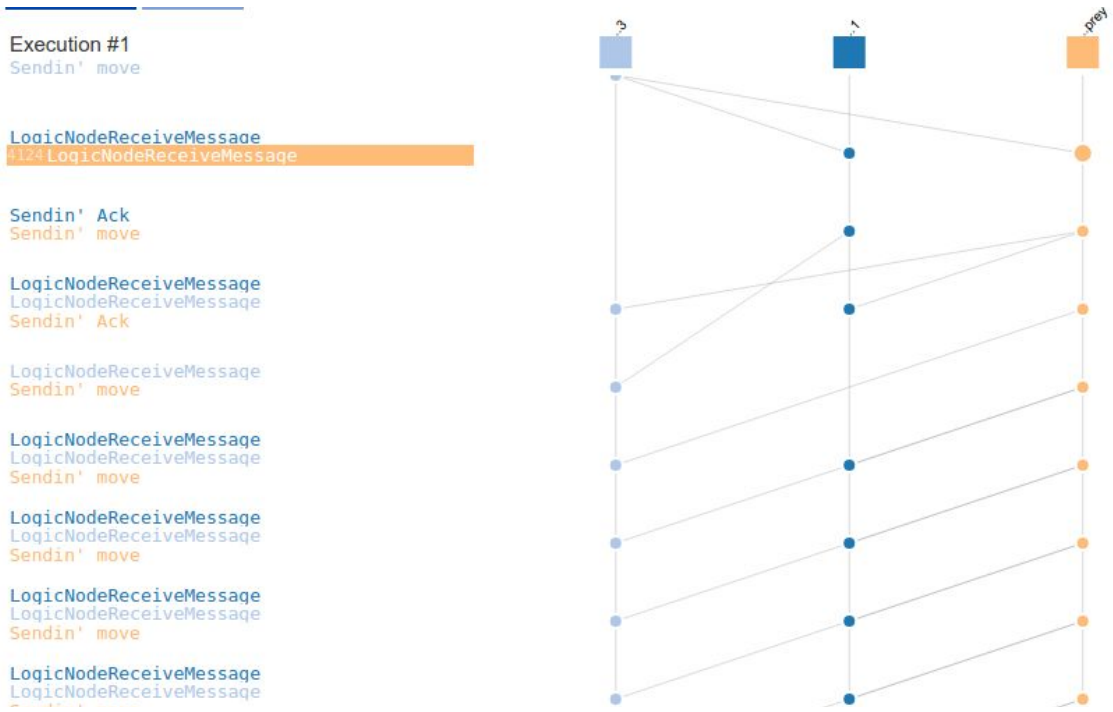

Figure 5: Movement verification protocol: Node 3 sends a move to the prey and Node 1, which then sends messages acknowledging this. The prey sends movement messages but does not need acknowledgements from the other nodes as it is a trusted entity in the system

When a logic node sends a move message to all of the other nodes, n, it stores this move as pending. Only when it receives n / 2 + 1 ACKs back for that move will it then update its game state with the move coordinates. If it doesn't get a majority consensus from the other nodes after a few rounds, then it will drop that move and get the next pending move from a channel.

The prey node's moves are not a part of this protocol. As mentioned previously, the prey node is a trustworthy node that makes moves based on random calculations.

Score Verification/Dissemination Protocol

Each time the logic node receives new coordinates from its Pixel node, it checks to see if the new coordinates matches the current coordinates of the prey. If they do, it updates its own score then sends out a "captured" message to all nodes in gameplay.

When a logic node receives a captured message from another player, it does three checks: first it checks to make sure the player's coordinates matches the prey's coordinates based on its own game state, second, it checks to make sure the move is valid, and lastly, it checks to make sure the score being sent is accurate based on its current record of the player's score. If all these checks pass, the logic node will then update its game state with the player's new score.

Lockstep Protocol

In order to prevent "look-ahead" cheating, when a player delays transmitting their moves until they know more information about the game, we implemented a lockstep protocol. A lockstep protocol involves waiting to receive signed hashes of moves, or move "commits", from other players before revealing your own move to them. When you do receive a move, you check it against the hash you received to ensure the move that was made is in fact the move that was committed to before your move was revealed to them. If it was not, you can discard the player's move. We also used the `crypto/ecdsa` package here to sign and verify the move commits.

Initially, we were planning to implement lockstep between all the nodes; however, upon further inspection, there is no advantage to knowing the future locations of other players, as physical interaction between non-prey players has no effect on the game. Thus, only the prey node needed to ensure it had move commits from all nodes before revealing its move to them.

Yet, we realized that this implementation would cause the prey to freeze in the game if the prey node doesn't receive move commits from all in-game nodes, which is a loophole malicious nodes could exploit. To resolve this, we did two things. First, if a player doesn't input a move within one second, the logic node will automatically send out a move message of its current position. That way, it will be constantly sending out move commits for the prey node to process. The second modification was made in the prey's logic node, where we've added a timeout so that after a certain period of time, the prey will send its move out to those who have submitted a move commit. For those who didn't get their move commits in on time, then they would see the prey teleport the next time they send through a move commit. However, since the prey is trustworthy (see assumptions), this isn't an issue.

Testing

Unit and Integration Tests

During the development of project 1, we encountered several instances where breaking changes were made, causing hours of debugging to locate the introduced problem. These changes continued to be integrated due to a lack of tests; and we only discovered the issues when we would attempt to run the whole system at a later time.

Learning from that experience, we required all pull requests adding or modifying functionality to Wolfpack to include at least one test. The result is a fairly comprehensive test suite of the expected behaviours in this

system. However, this test suite was, at times, completely unable to provide feedback on whether the game was still functional.

Testing Difficulties

Despite having all integration and unit tests passing, there were many times the game was played and noted to be completely non-functional. This is because it was difficult, if not impossible, to write good tests that capture the real-time, unpredictable nature of player inputs, and therefore also the order in which they would arrive. Unit tests were no substitute for gameplay.

In order to regularly assess our progress on the game, we would play the game over Azure every few evenings, using online messaging and in-game logging to document what we were observing. Though certainly not the most efficient method of testing, it did allow us to identify and fix several bugs that would have otherwise gone unnoticed. However, the need to do this to test our system certainly slowed our development process, and means our system has effectively only been tested with 4 actual player nodes in the system.

Evaluation

We used the DInv and ShiViz tools to help verify system correctness, and the Golang pprof tool to run a CPU profile.

DInv

The DInv logs were created using an older commit. This was because integrating DInv required changes to the structure of our code (e.g., removing relative paths) which would cause merge conflicts and decrease the organization of our code. The last merge before splitting off DInv would be merge #62. The logs will be in the final report, as well as branch titled “dinv”.

To ensure our system was running correctly, we took several steps to protect data from being modified concurrently, could be described linearly, and would not receive unexpected messages.

Invariant #1: impl_node-node-interface_numAccess == 0

Our program required multiple goroutines to run at the same time to handle communication between nodes. This meant that the structure that stored information about connections to other nodes needed to be controlled tightly. For example, if two goroutines were to access the connections, and one adds or deletes a connection at the same time that another is sending a message, we could send a message to a deleted node, fail to send a message to the added node, or cause our node to crash due to a concurrent map iteration and write. The solution in this case was to allow only a single goroutine to access the node connection data.

To demonstrate that there would be only a single routine accessing node connection data we added a variable `numAccess`, which was incremented each time that a process entered the critical section and decremented each time the process left. When a goroutine wanted to access these processes, they messaged the goroutine via a channel to request an action. The DInv invariants showed that outside of the critical section this value would always be zero, showing that no two processes would be accessing this section at the same time.

Invariant #2: impl_node-node-interface_1_oldPreySeq < impl_node-node-interface_1_preySeq

In a peer-to-peer online game, one of the major hurdles is determining a global state efficiently. Because each node is separated from every other node, it is difficult to know when a particular message was sent or

received by another node. The solution for us to get at least some information we could corroborate with the other nodes was to include sequences in the messages. To prevent messages from being received out of order, we limited the rate at which messages could be sent from a given node. From empirical testing, we found that it was rare(<1%) for a message to be received more or less than 0.1 seconds from when it was expected to be sent. We limited the time to send 2 messages to 0.2 seconds so it would be very rare to receive one before the other. To test this, we added a sequence variable to the messages so we would know the order they were sent and tested that they would increment with DInv. The result showed that the new message would always have a sequence number greater than or equal to (they are initially equal) the previous message.

Invariant #3 - impl_node-node-interface_acks <= impl_node-node-interface_otherNodes

One of the major issues with creating a consistent game is implementing a verification procedure. If a receiving node knows that a game state is incorrect, they must be able to communicate that information to the sending node. To do this we used an ACKing system. However, when we receive ACKs, we do not know which moves are being ACKed. To solve this, we binned the ACKs using the message sequence number as a key to a map. To demonstrate that we would not receive more ACKs than expected, we ran DInv with the variables that count the number of ACKs as well as the number of nodes connected. The result shows that the number of ACKs will be equal to or less than the number of other nodes we are connected to.

Shiviz

Figures 3, 4, and 5 are diagrams derived from Govector and ShiViz. The logs to create these diagrams are in the report folder. We used a forked version of Govector that can be found here: https://github.com/rzlim08/GoVector. The forked version includes only minor changes related to bug fixes and smoothing out the process of loading the Log files into ShiViz.

CPU Profile

We ran a CPU profile on our code to determine the reason for the slow execution and communication. The processes that used the most CPU cycles were the ones related to signing and verification of move commits and moves. Because we signed and verified each of the move commits and the moves being sent, and each node could send multiple of these per second, we found that the cost of having this verification grew quickly over time. This shows our threat model significantly impacted the gameplay experience.

Discussion

Consistency vs. Speed

Naively, we had originally planned to make a game that was both consistent and fast; however, we were soon faced with the facts that there is a fundamental trade-off between consistency and speed. It may not be possible, especially with our current skill set, to create a game that has a high degree of consistency and speed using a peer-to-peer implementation. In fact, in writing this report, one of the sources we found cites the rate-limitation imposed by a lockstep protocol as part of the reason why this protocol, and peer-to-peer online gaming, has fallen out of style in favour of a centralized server model[6].

Before implementing the lockstep protocol, which rate-limits player movement to how fast the prey is able to collect and verify move commits from players, and check move commits against actual moves, a player could expect to see their inputs reflected in the game state between 180-200 ms. However, after

[6] https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/

successfully implementing this protocol, this number rose to 400-500 ms, on average. This is an increase in latency of more than 100%, and results in a significant negative effect on gameplay.

Players value quick response times to their inputs; our system does not deliver that. Upon further reflection, it would probably be better when designing a real-time online game to recognize the trade-off between consistency and speed, and make decisions which emphasize speed over consistency when designing the system. Such considerations were not made in the planning of our system due to our inexperience designing and implementing real-time, distributed systems.

Complexity

As we noted in our proposal's SWOT analysis, we felt we could be drastically underestimating the difficulty of implementing a real-time game due to inexperience. Indeed, this was the case. As we built the system, the number of messages sent between nodes became increasingly unwieldy, with short gameplay sessions generating vector clocks with individual logical clocks in the many thousands. Though this may be the norm in online gameplay, it is not something we are experienced in working with. Of course, the number of messages a node sends and receives would have been drastically reduced had the node only had to communicate to one other node, such as a centralized server.

The complexity was further exacerbated by the fact we originally expected a "player" to have only a single node, rather than the current logic/Pixel node setup. This further complicated our system; adding to the number of messages a logic node would need to process, complicating coordination of game state, as well as testing and further building on the system.

Peer-to-Peer or Centralized Server?

After building our system using a peer-to-peer model, we would not advise any novice developers to do the same. The lack of trust between peers creates the need for protocols and checks which are costly in terms of time and CPU cycles. Furthermore, the coordination of messages between such a large number of nodes is difficult without significantly throttling gameplay. Despite the fact that Wolfpack "works", its gameplay probably would have benefitted from some of the simplifications that arise from a client-server model.

THE END