p2p.io: the more the merrier

Edward Chen, Godwin Pang, Frank Wang*
[ejc042],[gypang],[wew168]@ucsd.edu
University of California - San Diego

ABSTRACT

Massive multiplayer online games, especially with the introduction of IO-style games, there exists an increase desire to scale and reduce server side costs. Typically, these multiplayer games run on a traditional client-server architecture where all of the players connect to a single server. This architecture is enticing for game developers because of the model's simplicity and ease in development, but poses some obvious bottlenecks when all clients are connected to a single server. A peer-to-peer(p2p) architecture would distribute the computational load of the game to the client and lower the costs to run servers. Therefore, in this paper, we explore the efficiency benefits as well as a cost analysis of a p2p architecture compared to the traditional client-server architecture. For our project, we successfully recreated an Agar.io style game and test both architectures on 10, 30, and 60 t2.micro Amazon EC2 nodes. Our results indicate that the performance of the p2p system scales better than the client-server architecture. To our surprise, even on small scale systems with less than 20 nodes, the p2p architecture is still as performant. Although p2p model has a lower performance cost, the software development cost in creating p2p systems poses a significantly larger overhead compared to the client-server system.

1 INTRODUCTION

Multiplayer video games have become an integral aspect of the gaming community and a profitable industry. Online games typically depend on traditional client-server architectures, in which there is a single, dedicated machine that runs the game server and all players connect to this server as clients. In many cases, one of the client machine could even act as the server.

Within the past 5 years, IO-style games grew in popularity among the online gaming community. An IO-style game is a free to play, casual, multiplayer game with few mechanisms and minimalist graphics. Two examples that spearheaded the IO game community include Agar.io and Slither.io. "Both [of these games] used .io as their domain of choice, so, it became the de facto standard for similar games and eventually morphed into the name of the genre."[10] An IO-style game does not require an io domain as long as it meets the requirements stated above.

The simplicity of IO-style game play allows them to be played by a massive number of players simultaneously, which puts challenges on the traditional client-server architecture as the server can easily become a bottleneck. For this project, our goal is investigate a more scalable approach for these IO games. We would like to evaluate the performance and cost trade-offs between a traditional client-server implementation versus a peer-to-peer (P2P) implementation of the game, Agar.io[16]. As mentioned, a client-server model for games presents an obvious bottleneck on the side of the server. Thus from a scalability perspective it is desirable to offload computation onto clients, moving the server out of the critical path. Such a design also means that game developer will no longer have to pay as much for running servers, lowering the entry bar for indie game developers to access large-scale game development.

Our target game is Agar.io. We completed both a client-server model implementation of the game as well as a peer-to-peer counterpart with replication to support players joining and leaving the game. To evaluate the performance of each the architectures, we utilized a large cluster on Amazon EC2 and tested with a test bed of 60 machines (players). In order to investigate the point at which paying the development overhead of a P2P game provides sufficient benefits over a client server implementation, we will measure both client request RTTs and server costs. As an aside, our current testing framework allows us to easily increase the capacity of our cluster. Unfortunately due to restrictions on Amazon EC2, we have reached the maximum vCPU limit of 64 and cannot add additional nodes to this single AWS account.

2 BACKGROUND AND RELATED WORK

Peer-to-peer online gaming is far from a new concept. One of the earliest P2P gaming system, Amaze [1], could date back to 1985. Yahyavi and Kemme composed a survey [18] about the different architectures proposed for P2P online gaming, which will be used as the primary source as our discussion in this section.

*All authors contributed equally to this work.

223B '20

Structure. Pure P2P game systems typically form an overlay network on top of the Internet and can be categorized into two types: structured and unstructured. Structured P2P architectures including [2], [8] and [17] generally use distributed hash table (DHT) based mechanisms to store and propagate game states. Communications between nodes can either be routed through the DHT or by multicasting. Unstructured ones like [7], [1] and [12], on the other hand, lack a global mechanism like a DHT and instead rely on direct communication to exchange the game state, with the all-to-all mesh being an extreme example of this architecture. Alternative designs like [3] and [19] use a hybrid of P2P and client-server architecture to offload part of the computation to the clients while keeping the centralized server.

Game entities and consistency. Generally, entities in a game can be divided to players (which are either controlled by a human or an AI, and will be referred to as just "players" in the paper from here on out) and objects. Depending on the game's semantics, different entities may require different levels of consistency when running on a distributed system. For instance, immutable objects like the game scenes and map boundaries need to be consistent across all players, and since such information is static, these entities can be instantiated easily. Renewable in-game resources that can be acquired through reward or random generation, on the other hand, do not need to be strongly consistent across players. The decision on the consistency model to be implemented should aim to minimize the latency experienced by players while preventing unexpected artifacts due to race conditions from the game play.

Interest Management. Due to the limitation of a player character's movement speed and sight range inside a game, only a small subset of the entire game world is needed in order to compute and render the next game state for an individual player. The visible and sensible space in a game for a player at a particular time is defined as the area of interest (AOI) of this player. In a P2P design, it is possible to restrict the player to only retrieve the game state inside their AOI to minimize bandwidth requirement and observed latency. This filtering process is call Interest Management (IM) proposed by Morse et. al. in [9]. AOI also has certain implications for the consistency model required for the game. For instance, Santos et. al. introduced the notion of Vector-Field Consistency in [11], which takes into the practical consideration that players usually don't need to have a full view of the game world, so the system only needs to keep the information up to date for each area within a player's AOI.

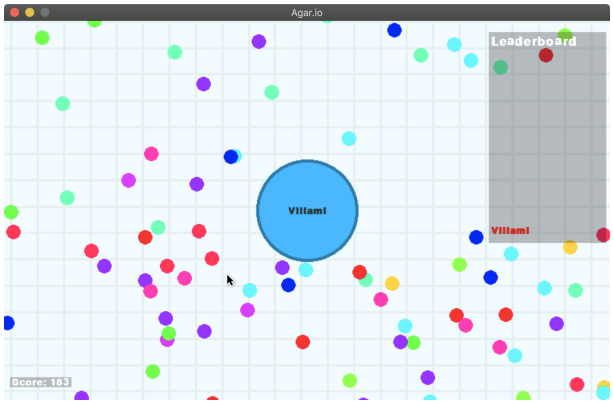

Figure 1: An Agar.io game run. The big blue circle at the center of the screen is the player, and the small circles are foods. The player is moving toward the direction where the cursor is located. Once a player engulfs a food, it will eat the food, have its score increased and grow in size.

3 DESIGN

In this section, we describe the game, Agar.io, specifications in detail and discuss the high-level design of both the client-server model and the peer-to-peer model of this game. Front-end clients (referred to as simply "clients") are strictly dumb terminals to the servers in both cases. The role of the client is to feed user inputs to the server, receive updated game state from the server, and render the current game state. Servers are considered as the source of truth and all game computation is handled within the servers.

3.1 Agar.io Game Features

Agar.io is a massive online multiplayer game where each player is represented as a circle. The size of the circle is determined by the amount of mass the player currently owns. The circle will always be centered on the player's screen, and the circle will move across the game's map towards the direction of the player's mouse position. The speed of the circle reduces as the circle accumulates more mass. The objective of the game is to accumulate the most amount of mass by eating randomly spawned food or by eating other players that have less mass than you. Figure 1 shows a screenshot of an Agar.io game play.

A few tweaks were made to our implementation of Agar.io in order to simplify the overall game model. In the original game, players after acquiring a certain mass limit, could split their circle in order to increase speed. Our implementation instead incorporates a "poison" food that will half the mass of a player. Secondly, our implementation of the game does not include "viruses", which are environment objects that could

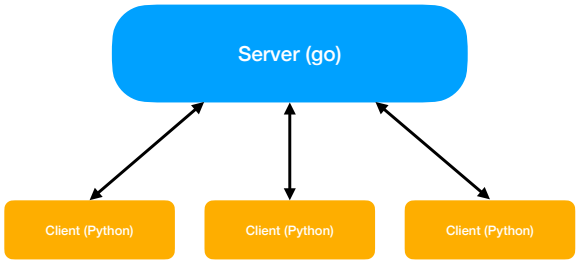

Figure 2: An illustration of the client-server architecture. A single server handles all the client requests.

reduce a player's mass. Therefore, our game incorporates 3 main features: player movement, players eating other players, and players eating food & poison.

3.2 Client-Server

The client-server architecture consists of many clients sending and receiving updates from a single server, as shown in figure 2. At a high level, the game loop begins with clients receiving inputs from the users, forwarding these inputs to the server, and blocking until a reply is received from the server. The server will then handle all game computation based on the received inputs from the clients, and reply to the clients the updated game state. The clients after receiving the updated game state will render the current state of the game and unblock to receive more user inputs.

Compared to a P2P model, the client server model has a far lower programming effort cost and is far easier to understand. There are, however, obvious bottlenecks to this architecture when the number of players increases. For example, if the server was multi-threaded, the current game state is likely to be represented as a shared data structure. In order to update the game state, the server will then need to pay the price for lock-contention.

From a cost perspective, the client server architecture will incur more costs for the game developers compared to running a P2P architecture. This is because the game developers would need to either rent or host their own game servers where as a P2P architecture would run the game on client side.

3.3 Peer-to-Peer

3.3.1 *Overview.*

In our peer-to-peer architecture, a monolithic server that handles the game logic no longer exists; instead, the computation is offloaded to each of the players' machine. Considering scalability and fairness, it is important to offer a best-effort load distribution amongst the players. We do this by dividing up the game map into regions using an overlay grid, and have each player be responsible for a subset of regions. Agar.io, along with many other IO-style games, exhibit a high degree of physical locality where in most cases only a finite number of blobs compete for resources. We designed our game implementation following the idea that each piece of information should have a single source of truth. Each player runs an implementation of the client similar to the one in our client-server model. Each player also runs a monolithic server, logically divided into the following components:

(1) Player Server
(2) Region Server
(3) Routing Service

Player servers are responsible for *storing* position and mass information of the local player and calculating a player's movement. Region servers are responsible for *storing* and *spawning* food positions within the regions they manage, and *updating* a player's mass upon collisions within these regions. The routing service is responsible for routing the request from player to the set of regions it's interested in, as well as for regions to contact each other for replication and fault tolerance.

We now discuss each component separately in detail.

3.3.2 *Client.*

The client in the P2P model is largely identical to the one in the client-server model. The difference is that instead of contacting the single game server, it contacts a local player server that processes the client's position update and sends the client about the player's new status. In other words, the P2P structure is transparent to the client.

3.3.3 *Player (Local) Server.*

The client interacts with a local player server, which is responsible for handling this node's player data. A player's data contain its IP address, its x, y position, an incarnation number, and its mass. Player's data are not replicated, since once a player leaves the game, its data should not persist and other players should no longer see it. The incarnation number is used to differentiate player instances across respawning.

Section 4.3 describes the APIs that are exposed to the client, which includes `Move`and `Region`. Clients send player movement updates through `Move`, by taking in the current mouse position. Upon receive `Move`, the server computes the new position of the player as well as the regions that the

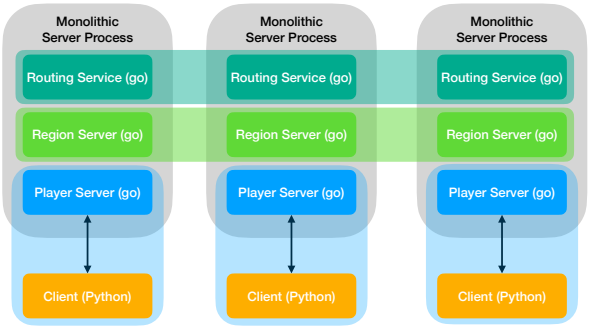

Figure 3: An illustration of the P2P architecture of the game. Each node runs a monolithic server process that contains a player server, a region server and a routing service. The routing services and the region servers each forms a logical layer. The player server serves the data about the player, and connects the python graphical client to the region servers for pushing update and fetching map information.

player would be able to see given its mass (i.e. the player's AOI). It then pushes the player's data to the primary and backup region servers (described in 3.3.4) that handle the regions it overlaps with, and fetches the updates from the primary region server to obtain information about change of mass (eating food and other players) or liveliness (getting eaten by another player). All information contains a timestamp so that out-of-order information sent to a region server can be rejected. Finally, the player server returns the updated player position to the client. When a player dies, the player server will notify all regions in the player's AOI with a special tombstone update.

On a `Regioncall`, the server computes the player's AOI (the visible regions of the player) based on its current state to determines which region it needs to contact to retrieve the information. It then scatters fetching requests to the regions within the AOI, aggregates the information as a set of visible food and players, and send it back to the client.

When the player server tries to fetch information from a primary region server that has gone down (due to active exit, failure, or not ready), the player server will fallback to the designated backup server.

3.3.4 *Region Server.*

Region servers are responsible for handling food spawning, player eating foods, and player eating other players. Each region maintains a data structure that stores the (x, y) coordinates of food in the region.

Regular operation. When a player server pushes the player's information to the region, the region server would compute the collision and decide which foods were eaten. After the computation finishes, the region server will push these results to the player server so the player can increase its mass properly. The region server also periodically computes the number of food in this region; if it's below a certain threshold, the region will spawn new food.

As mentioned, the player servers are responsible for pushing each player's information to the region server. We say a player is *in* a region if its center coordinate lives in this region. We also define that a player is *seen* in a region, if the player's circle overlaps with the region but its center is not necessarily in it. Region servers serve as the source of truth for player-player interaction (i.e. player eating or being eaten by each other). For each player that is *in* a particular region, that region would compute whether the player is being eaten by another player by iterating through all the players that are *seen* and calculating if another player overlaps with this player. If a player was eaten, then the region server will notify the eaten's player server that the player has died, and will notify the eater's player server to increase mass the eater's mass. Notice that the player data stored on the region server essentially acts as a local cache, and two adjacent region servers could cache inconsistent information when a player crosses from one region to another. In this case, two region servers could potentially think that the player belongs to it, and give the player server inconsistent information. We admit the policy that if any region server tells the player server that the player was killed, then the player will assume the player has died, ignore any further updates, and start the clean up routine. Any inconsistent information given to a player server can be resolved based on the timestamp of the sent information.

Region servers will store a timestamp for each player, and a clean-up routine runs in the background to periodically remove the players who have not contacted this region server for a timeout period (currently set to one second).

Replication. Food is a generic type of object that needs to stay persistent throughout the game, independent of nodes joining or leaving the game. Therefore, information about food is replicated from the primary region server to the backup region servers. In our experiments, we only have 1 backup region server for each primary region server and utilize the successor of the primary region server as the backup server. For higher fault tolerance, these number of backup servers can be easily extended. A primary-backup scheme with sequential consistency is implemented for food replication: when a region server decides to spawn food in

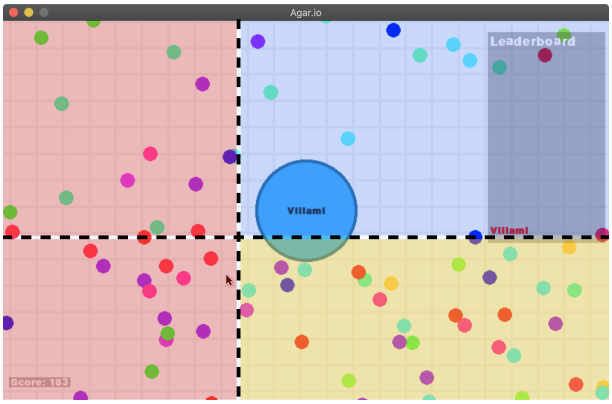

Figure 4: A grid overlaid on Figure 1. The player is *in* the blue region; the orange region has *seen* the player, and the red regions don't know about the player.

a particular region, it will add the food into the region and then contact the backup server to replicate this action; the added food cannot be observed by the player server until the backup has finished replicating the spawned food. This procedure is the same when a food is eaten by a player. We chose to use a sequential consistency model mainly because of the language limitation of Golang (that one has to hold a lock on a map for concurrent read-writes), but we argue that for this particular game, it is acceptable to implement a weaker consistency model since it's unlikely to affect player experience.

Node Join and Leave. We allow players to join and leave the game with a finite interval between each event. The underlying routing service will notify the region server when its successor or predecessor changes. On such notifications, the region server will perform a data migration. There are four different cases:

node-leave = node that leaves
node-join = node that joins
current node = node that observed the change

(1) *Predecessor leaves the ring*: In this case the key space of the node-leave and the current node merges. The current node becomes the primary for all regions that originally had the node-leave (Regions) as their primary. By how the replication was done, the current node will have the backup data for Regions, so it simply promotes these Regions to primary region servers and replicates this backup data over to the Regions' successors.
(2) *Successor leaves the ring*: In this case the current node loses its backup, so the current node needs to push its data to its the new successor.
(3) *New predecessor joins the ring*: Here the node's key space should split, so part of the regions now have the new node as their primary. The current node computes which set of regions should migrate their data to the node-join and sets the node-join to be the backup for the regions. The node-join would likewise acquire these corresponding regions as the primary. Finally, the current node would remove these regions on its successor.
(4) *New successor joins the ring*: In this case the current node migrates its backup from it's previous backup (its successor's successor) to the newly joined successor.

Each region has a ready flag that is set to true once data migration is over. Therefore, a newly joined node will not have ready regions until most of the data migration is complete. When player servers contact regions that are not ready, the routing service will redirect the message to the backup server which has the complete information.

3.3.5 *Routing Service.*

We use consistent hashing to hash the nodes (with their IP address) and regions (with their normalized location) into the same hash-space, and place each region at its successor server, just like in the Chord DHT [14]. When any entity needs to retrieve or update information inside a region, it first goes to the routing service to perform a lookup. The lookup will return the successor server of this region, as well as a backup server that handles replication and fault tolerance. We planned to use a DHT service to build the structured P2P gaming system, but many of the online DHT libraries we found were unreliability. Currently we're running the system with a fixed set of IP addresses on initialization and use a full mesh ping to determine which server is up. However, a Chord DHT service can be easily substituted in place of our current routing service.

The routing service is also used for contacting player servers (i.e. for increasing mass), but since the player's information contains its IP address, the routing service merely converts this information to an open RPC connection.

The last responsibility of the routing service is to notify the region server about node join and leave. Structured P2P system typically includes this functionality in the DHT layer. In our case, we cache the current predecessor and successor node and update them on each ping, and when this information changes, the routing service will notify the region to perform data migration.

In a DHT based routing service, for a new node to join the running game, the node needs to know the address of at least one node who's already in the DHT, or if none exists, the new node needs to create one. To facilitate such discovery, an entry server was built to serve as a "lobby". This is similar to the bootstrapping node that is commonly found in overlay

networks. When a node tries to join the game, it will first contact this entry server and obtain information about nodes in the DHT. In our current implementation with an all-to-all connection, an entry server still exists to inform the nodes whether they are creating a new game or joining an existing game (see section 4.2). In either case, the entry server has a very light workload as it's only contacted when nodes try to join the game, and thus has a centralized design. Currently the entry server is not replicated; we can use Paxos to make it fault-tolerant.

4 IMPLEMENTATION

We use Python with the Pygame[4] framework to implement our graphical client that interacts with the human player, and Golang (Go) for our server(s) in both the client-server model and the P2P model.

4.1 Game Implementation

Pygame is a Python module designed for building video games. We build our work on top of the work of Viliami Tuanaki, who has a single player implementation of Agar.io at [15]. On the server side, we use the go language to implement the game logic. gRPC [6] is utilized for communication between clients and the servers for the client-server model. For the P2P model, we also used gRPC for communication between the Python client and the Go player server despite they run on the same machine, to avoid the complexity of cross language compilation.

4.2 Game Start

When a node first starts, it contacts the entry server to find out whether it's joining an existing game or should start a new game.

In the first case, the entry server tells the new node that it should join an existing game. The node then simply start its region server and wait for its successor and predecessor to push the region information to it.

In the later case, the node then enters a hold state and keeps pulling from the entry server. The entry server is configured with a minimum number of players before a game starts, and when the number of players in the hold state reaches this minimum number, it responses to the pulling nodes with a "can start" message. Upon receiving this message, each node would ping the nodes in the list of IPs, determine the set of regions that it possesses as primary and backup, and allocate these regions. The nodes finally start their player servers to allow the Python client to join.

4.3 Client-facing API

The server in the client server model, and the player server in the peer to peer model expose two APIs that the python client can utilize: `Move`and `Region`. When the client has a new mouse position update, it calls `Move`to send this update to the server. The server then computes the next game state and returns to the client. The client then calls `Region`which fetches the information about food and other players that are within the visible regions on the client's graphical interface and does the rendering.

4.4 Agar.io Bot

In order to test our architectures at scale, a simple Agar.io bot was created to simulate players. The bot prioritizes eating other players with less mass than itself as well as running away from players with greater mass. If no players are within the bot's AOI, the bot will prioritize eating the closest food. Movement is calculated by moving the bot towards or away from its next objective (player or food). If no objective is within range, the bot will repeatedly decide a random direction to move in for 1000 steps until a new objective enters the player's AOI.

5 EVALUATION

We evaluated the performance of the client-server and P2P versions of our game at different scales, based on the RTTs of gRPC requests from client to our player server. Our metric for evaluating the effectiveness of our peer to peer model by comparing the median RTTs and 99th percentile latency of our experimental runs. Running our Python game on the EC2 instances required some modifications due to the lack of a video driver on the EC2 instances. As a result, we ran our game clients in Pygame's headless mode, which omits the actual drawing calls. This has no effect on our measurement of gRPC RTTs.

5.1 gRPC Wrapper

To time our gRPC requests, we wrote a small python wrapper that times the region and move calls separately. We keep all RTTs in a data structure in memory, then added a SIGINT handler to the Python client that flushes it to disk in a JSON format.

5.2 CLI Tool

To orchestrate the test bed, we wrote a custom CLI tool in Python that handled spawning and despawning nodes, starting and stopping server/clients, and gathering logs from all nodes. Our CLI tool allows us to run experiments on testbeds spanning hundreds of nodes, but AWS imposes a limit of 64 vCPUs per account and thus we were only able to spawn testbeds within those limits. We utilize AWS's boto3 [13] library to handle EC2 instances, and Paramiko [5] to run commands over ssh connections.

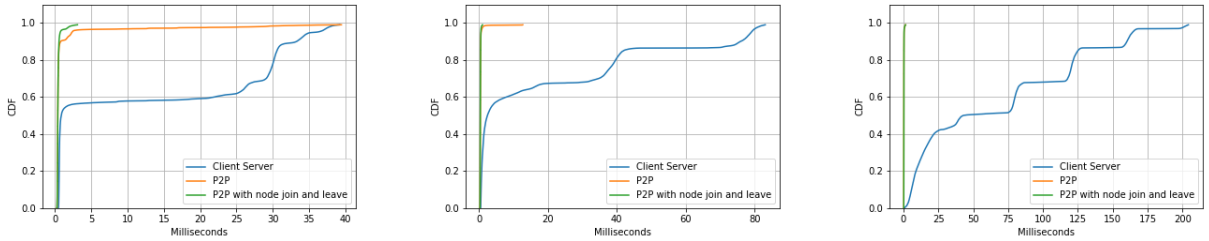

Figure 5: CDF of Move gRPC RTTs. Left to Right: 10 Nodes, 30 Nodes, 60 Nodes

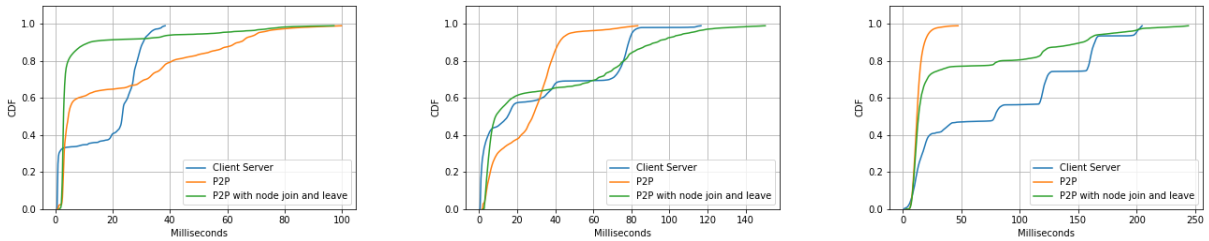

Figure 6: CDF of Region gRPC RTTs. Left to Right: 10 Nodes, 30 Nodes, 60 Nodes

Since we utilize an all-to-all ping for routing between nodes, each server requires global knowledge of all ips of nodes within the games - we use EC2 private ips to do such routing. To propagate the information, our CLI tool updates a configuration file and fans out the update to all nodes within the experiment via git. To start the P2P experiments, we follow the order of starting the entry server, then the servers, and finally the clients. To start the client-server experiments, we follow the order of starting the server, then the clients. We do the above by running commands over ssh connections. This admittedly is not the best solution. Since servers/clients are long running processes, we have to open multiple ssh connections and keep them open. There is also no guarantee that a server has actually started before we run the client; we run an abundance of sleep commands as a best effort guarantee.

At the end of each experiment, we kill the clients and servers in reverse order to how they were started. Once everything is killed, we run scp commands to fetch the logs files from the EC2 machines.

5.3 Experiments

For our game setup, the map size was 10,000 by 10,000 units with 100 regions, each region was 1,000 by 1,000 units. Each region has a maximum food limit of 60 food and has a random spawning rate once the region contains less than 30 food. The player's initial positions are randomized across the map and players respawn in intervals of 10 seconds upon death.

We chose to run our experiments with testbeds of size 10, 30, and 60 t2.micro nodes. The testbed size is representative of the number of players within the system, and does not include auxiliary servers that are required for the game to function. For example, our P2P experiments run the entry server on its own t2.micro machine, and runs all players separately; the 60 node testbed would be 60 nodes for the players plus 1 for the entry server. The same applies for our client server experiments.

We also ran experiments on our P2P setup with node joins and departures. We had 2 alternating node departures and join events separated by 36 seconds each resulting in the same total experiment lengths of 3 minutes. A detailed breakdown is as follows:

- t=0s: Start all nodes
- t=36s: Kill 1 node at random
- t=72s: Start killed node
- t=108s: Kill 1 node at random
- t=144s: Start killed node
- t=180s: Kill all nodes

Note that our CDFs are truncated to the 99th percentile. This is done to eliminate the presence of long running outstanding requests. In an ideal implementation, clients would add timeouts onto each requests to avoid outstanding requests from lowering the frame rate of the game; our current implementation does not include that. Our results show that the performance of our P2P implementation of the game

improves with the number of nodes in the game, whereas the client server implementation's performance decreases.

We observe the trend that P2P tail latencies decrease significantly as the number of nodes in the game increase. This suggests that given the same map size and the same number of regions, the game's performance will increase with the number of nodes in the game which is characteristic of a P2P system. We also observe that in the P2P experiments with node joins and departures exhibit lower median RTTs and similar tail latencies for the region calls, suggesting that the P2P model performs at least as good as the client server model in the presence of node failures.

All of the client server experiments exhibit an evident stepped CDF, especially for the region gRPC RTTs. This is due to lock contention. We see similar steps in the P2P CDFs but with less steps as the number of nodes increase, presumably due to the lower number of regions stored on each node.

For some experiments, we observe some anomalies where the P2P experiment without node joins and leaves exhibits a much higher tail latency than the P2P experiment with node joins and leaves. Unfortunately due to the highly random nature of our bots and food spawning, it is hard to ensure a perfect performance difference between systems. Even if we deterministically program the bots to all move in a certain way and have the food spawn with the same random seed, it is difficult to ensure that the client and servers start at exactly the same time. We defer deterministic evaluation of distributed systems to later work.

5.4 Issues

5.4.1 Map Partitioning.

5.4.2 Hashing.

In implementing consistent hashing for regions, we initially used the fnv library to implement hashing of regions to servers. Our results showed that the P2P model was performing increasingly worse than the client-server model, which is opposite of what was expected. Upon inspecting the distribution of our hash functions, we observed that the regions were extremely unevenly distributed across our nodes which reduced our peer-to-peer system to a pseudo client-server system with the additional overhead of replication. The fnv hash function exhibited high locality when hashing region ids under our id scheme of using the 16 bit x, y coordinates as a 32 bit key.

Swapping over to CRC 32 as our hash function resolved this problem; regions no longer exhibited high locality of hashes.

5.4.3 Locking.

We originally implemented the system using plain locks where we could have used read-write locks. This lead to the P2P model performing an order of magnitude worse than the client server model due to lock contention. Re-implementing the system using read-write locks eliminated the performance decrease on the side of the P2P model.

6 CONCLUSION

While we only ran our experiments at a limited scale and did not try more powerful servers in the client-server model, our preliminary results show that P2P systems for io games can both perform better and scale better than the traditional client server architecture. Supporting a system with similar game performance for a large number of players with the client-server model is likely to require a much more powerful machine at a higher cost. We have built a framework that supports statically partitioned regions, and with APIs that can be modified to support simple updates and information retrieval.

In developing the P2P version of our game, we realize that distributed systems programming introduces a prohibitively high cost of development compared to the client server model. For reference, development of the client server version of the game took a few days while development of the P2P version took the better half of 3 weeks. Due to this, we recognize that much of distributed programming would benefit from being abstracted into a higher-level framework that provides multiple levels of consistency guarantees.

REFERENCES

[1] E. J. Berglund and D. R. Cheriton. 1985. Amaze: A Multiplayer Computer Game. *IEEE Software* 2, 3 (1985), 30–39.

[2] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. 2006. Colyseus: A Distributed Architecture for Online Multiplayer Games. In *Proceedings of the 3rd Conference on Networked Systems Design Implementation - Volume 3* (San Jose, CA) *(NSDI'06)*. USENIX Association, USA, 12.

[3] Luther Chan, James Yong, Jiaqiang Bai, Ben Leong, and Raymond Tan. 2007. Hydra: A Massively-Multiplayer Peer-to-Peer Architecture for the Game Developer. In *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games* (Melbourne, Australia) *(NetGames '07)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/1326257.1326264

[4] Pygame Community. 2000. *Pygame.* https://www.pygame.org

[5] Jeff Forcier. 2003. *paramiko.* https://github.com/paramiko/paramiko

[6] Google Inc. 2016. *gRPC.* https://grpc.io

[7] J. Keller and G. Simon. 2002. Toward a peer-to-peer shared virtual reality. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*. 695–700.

[8] S. Kulkarni. 2009. Badumna Network Suite: A decentralized network engine for Massively Multiplayer Online applications. In *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. 178–183.

[9] Katherine L. Morse, Lubomir Bic, and Michael Dillencourt. 2000. Interest Management in Large-Scale Virtual Environments. *Presence: Teleoper. Virtual Environ.* 9, 1 (Feb. 2000), 52–68. https://doi.org/10.1162/105474600566619

[10] NightMayorn. 2019. *What are io games?* https://www.addictinggames.com/what-are-io-games

[11] Nuno Santos, Luís Veiga, and Paulo Ferreira. 2007. Vector-Field Consistency for Ad-Hoc Gaming. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware* (Newport Beach, CA, USA) *(MIDDLEWARE2007)*. Springer-Verlag, Berlin, Heidelberg, 80–100.

[12] A. Schmieg, M. Stieler, S. Jeckel, P. Kabus, B. Kemme, and A. Buchmann. 2008. pSense - Maintaining a Dynamic Localized Peer-to-Peer Structure for Position Based Multicast in Games. In *2008 Eighth International Conference on Peer-to-Peer Computing*. 247–256.

[13] Amazon Web Services. 2014. *boto3.* https://github.com/boto/boto3

[14] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Netw.* 11, 1 (Feb. 2003), 17–32. https://doi.org/10.1109/TNET.2002.808407

[15] Viliami Tuanaki. 2017. *Agar.io clone.* https://github.com/Viliami/agario

[16] Matheus Valadares. 2015. *Agar.io.* https://agar.io

[17] M. Varvello, C. Diot, and E. Biersack. 2009. P2P Second Life: Experimental Validation Using Kad. In *IEEE INFOCOM 2009*. 1161–1169.

[18] Amir Yahyavi and Bettina Kemme. 2013. Peer-to-Peer Architectures for Massively Multiplayer Online Games: A Survey. *ACM Comput. Surv.* 46, 1, Article 9 (July 2013), 51 pages. https://doi.org/10.1145/2522968.2522977

[19] Shinya Yamamoto, Yoshihiro Murata, Keiichi Yasumoto, and Minoru Ito. 2005. A Distributed Event Delivery Method with Load Balancing for MMORPG. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games* (Hawthorne, NY) *(NetGames '05)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1103599.1103610