Distributed Systems

K. Rojer, P. Wernke, M. Dortmond

Abstract— **With the increase in popularity of mobile gaming, game engines must support concurrent users. To support a vast number of users interacting with the system simultaneously, the system needs to have protocols that do not require to broadcast updates to all the clients for every attempted state change. Therefore, companies need to incorporate distributed systems to support these features. A distributed system is a collection of autonomous computation elements that appears to its user a single coherent system. WantGame BV intends to run a service of online warfare between computer-controlled dragons and hundreds of virtual knights. Hence, the architecture of the suggested system follows the traditional server-client model. This architecture complies with the requirements of a distributed system, where the client perceives the system as a single coherent working unit. The recommended system incorporates a load-balancer to distribute the clients among the servers. Moreover, the architecture of each server contains three components: a client component, a game engine, and a peer-to-peer component. Computer-controlled commands emulate each user-controlled player. For connecting to and disconnecting from the system, emulations are based on data from the Game Trace Archive (GTA)**

I. Introduction

According to recent market research published by market intelligence firm Newzoo, the value of the global video game market was 121.7 billion U.S. dollars in 2017 (Newzoo, 2018). The trend is positive as the market report also forecasts that the gaming industry would grow to 180.1 billion U.S. dollars in 2021. Furthermore, the US Entertainment Software Association reports that over two-thirds of American households play video games regularly. The increase in popularity of mobile games gives authoritative companies and gaming startups the opportunity to claim a share on this lucrative market.

Traditional video games are developed and deployed using a non-distributed system. However, with the increase in popularity of mobile gaming, a game engine must support massive amounts of concurrent users. Therefore, companies should set up their infrastructure by designing and implementing a distributed system. The computational complexity of these game engines is high, and the game engine should support simultaneous users. Therefore, companies should implement a distributed game engine.

The company, WantGame BV, intends to run a service of online warfare, between computer-controlled dragons and hundreds of virtual knights. The company plans to test their concept in the market by starting simple. To illustrate, they use a small virtual battlefield designed as a 25x25 grid.

The game rules are reasonably straightforward. Each player embodied by either a user or a dragon occupies a unique square. Each player has two attributes: a health status (HP) and attacking points (AP). Any user-controlled player can move in any relative direction by one step. In contrast, dragons cannot move. Moreover, warriors can attack dragons which are at most two squares further and vice versa. Nevertheless, warriors are not allowed to attack each other.

Furthermore, an attacker inflicts damage against the victim by reducing the victim's remaining HP with the attacker's AP. Consequently, when a player's HP decreases to zero or below, the player is eliminated from the battlefield. Of course, players can heal other nearby players. This action increases the remaining HP of the injured player with the healer's AP up to its maximum HP. More specifically, players can only heal other players within a distance of at most five squares.

A distributed system is a collection of autonomous computation elements that appears to its user a single coherent system. In a distributed system, each element referred to as a node may vary in size and function. The architecture of the systems follows the traditional server-client model. The system incorporates a load-balancer to distribute the clients among the servers. Furthermore, the server architecture contains three components:

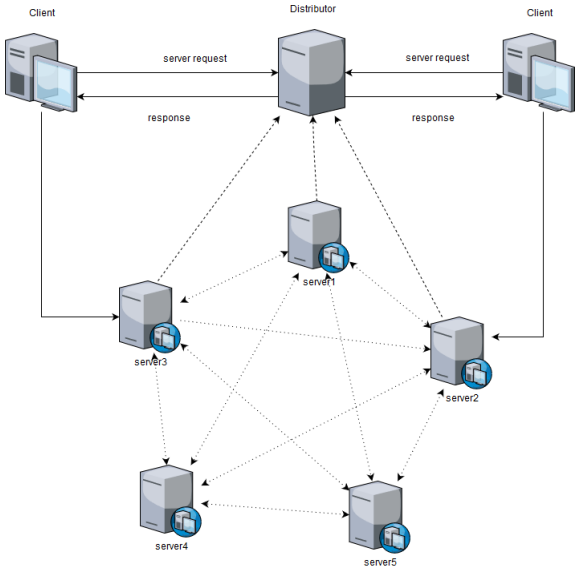

Fig. 1. Flow of messages when a client connecting to the server through a distributor.

a client component, a game engine, and a peer-to-peer component.

This article continues forward by first describing the background of the application and its requirements. Then, the article follows with a system overview explaining the design and its features. Afterward, the paper reports the experimental design, setup, and results. Finally, the article summarizes the findings and discusses the flaws, trade offs, and opportunities for implementing this particular distributed system.

II. Method

Systems Design

In essence, the architecture of the application appears to follow the traditional client-server model. In this centralized design, a client connects to some server that runs the DAS game through a network. Specifically, a client gets the address of a server via a distributor as shown at the top of figure 1.

The benefit of a client-server design is that most of the computations occur in the back-end of the application. Thus, reducing the number of messages required to preserve a consistent state across a wide-area network (Funkhouser, 1995). This architecture complies with the requirements of a distributed system, where the client perceives the system as a single coherent working unit.

However, to support a vast number of users interacting with the system simultaneously, the system needs to have protocols that do not require to broadcast updates to all the clients for every attempted state change. Therefore, the model of the servers should be a mirrored and reliable distributed peer-to-peer (P2P) system. In this decentralized structure, each server can make requests for data and respond to such requests. The central value of this P2P system lies in the fact that any component can be down, yet the system works fine. Using this P2P network, the servers share all incoming commands and render the game logic in a synchronized way. It needs to select valid client actions and broadcast these changes to the connected clients. The distributed system is designed such that any client should connect to the closest server, capitalizing on low latency. This will reduce the number of invalid moves, ensuring a better client experience.

Moreover, a client manages its local entities as well as copies of remote entities. Each time a client receives an update message for a remote entity, it updates the local copy of the remote entity. Most noteworthy is the fact that clients do not communicate directly with other clients. Instead, the servers forward messages to other connected clients using the aforementioned P2P network.

Each server module contains three main components, each performing isolated tasks. The server consists of a Game Engine component, a client communication component and a P2P communication component that other servers can use to establish a connection. The server provides the Game Engine with commands by caching the incoming commands using an IPC queue. Additionally, the Game Engine executes all game commands in an isolated process. Ultimately, the server broadcasts the responses to other servers utilizing a response queue.

Furthermore, the client communication component accepts connections from clients and acquires their game commands. These commands are validated by the server, by applying the game rules. The server can dismiss a command that was valid for the user due to previous commands by other players to other servers. The servers have to agree on a message order and broadcast the response

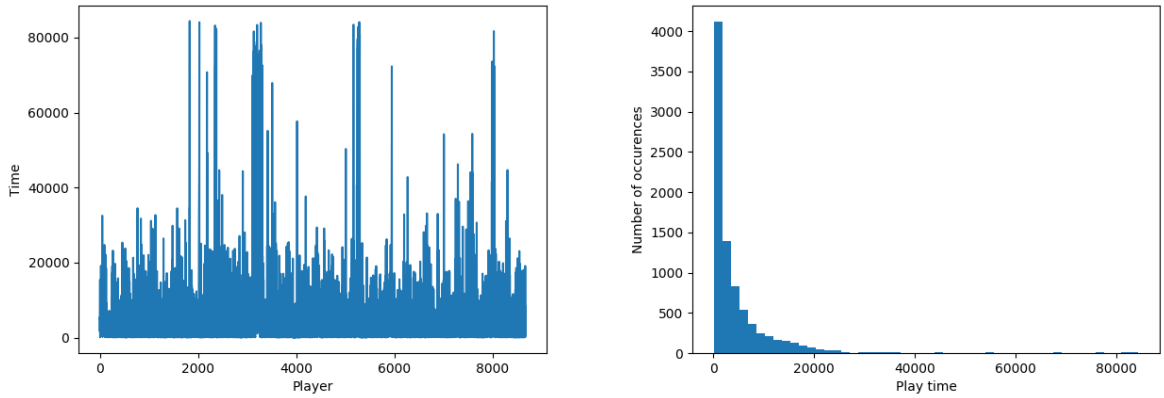

Fig. 2. Distribution of play time in seconds for a World of Warcraft game trace.

to all of its clients. More details of this process are described in the synchronization section. After the broadcast, the clients can update their replica of the game state. Both the client and P2P server communication use TCP to ensure full and in order message delivery.

Systems operation

The system operation requirements consist of multiple actions. Computer-controlled commands emulate each user-controlled player. Each emulated player performs one of the three following actions in order. The player first checks whether another player within heal range of at most 5 squares has lower than half of its maximum HP. If so, this player gets healed. Secondly, the player attacks a dragon whenever it is within attack range of at most 2 squares. Lastly, if none of these activities is taken, the player moves towards the nearest dragon.

For connecting to and disconnecting from the system, emulations are based on data from the Game Trace Archive (GTA) (Guo and Iosup, 2012). Specifically, DAS is similar to World of Warcraft (WoW) (Suznjevic, 2013). This Game Trace Archive has been investigated and the distribution of connection times can is reported figure 2. It tells us that most users are in-game for a minimal period ($<$ 1 min) and that a tiny portion of users is online for more than 5 hours.

Synchronization

Synchronization of the servers occurs utilizing a time interval approach. After a specific interval the server's synchronization the data among the servers in the P2P network. Specifically, each server receives commands from their clients during the interval. All servers collect these commands and requests but do not execute them immediately. However, at the specific intervals, the servers send their client commands to other servers over the peer network, whereby it is guaranteed that the server receives data from all other servers. The servers then order and execute these command all at once in a synchronized manner, removing any faulty inputs. Afterward, the servers broadcast the response to their connected clients. A time interval approach is chosen because it provides the best guarantee for data consistency. In a real-time approach, data inconsistency occur due latency in the command propagation among the network. Solving the data inconsistencies requires time and lead to rollbacks on both the server and client side. Frequent rollbacks provide a negative experience for the user and should, therefore, be prevented.

Load balancing

As mentioned before, connecting to the DAS system happens through a distributor. The distributor acts as an intermediary between the clients and the system. Whenever a client connects to the system, the distributor assigns the client to a specific server. The benefit of having a distributor is load balancing. The proposed DAS distributed

system implements two distributors. The primary reason for this choice is to have a back-up for reliability. The game servers send a non-blocking update message of their player total to distributors when their communication with clients is low, thus making sure that the distributors can make a fair assessment when assigning a new player. It assumes homogeneous game servers that run optimally when each has an equal amount of players. The distributor is transparent to the user.

Scaling

The game itself gets set up when the first server contacts the distributor. This server creates the initial game state and his assigned number of dragons. Clients can now have a playable game by contacting the distributor and receiving a server location. More servers can be added to the P2P network, by messaging the distributor, which gives a list of all servers in the network. The last added server in the network gives the current game state to the new server. This way of dynamically adding a server to the network could be used by the distributor to autoscale the number of servers when the player workload gets too large.

Fault-tolerance

The fault-tolerance for crashes or sudden disconnections have to be considered for all 3 systems.

The infrastructure contains 2 distributors with identical tasks, meaning that a single failure is handled. However, both can go down by malicious activities such as DDoS. If such an attack is carried out, it would only crash the distributors. This means that the game servers are safe and continue running the game, leading to excellent reliability for the players and showing high resilience of the infrastructure.

When a game server crashes, it does not take down the entire P2P network. Its peers remove the unresponsive server. This does mean that all clients connected to the missing server have to be reconnected using the distributor, which will assign them to one of the still operational game servers. System-wide failure will cause all clients to contact the distributor, which in turn can add more servers due to high demand.

A client failure is simply handled by removing the player.

Experimental setup

In order to do experiments on the proposed setup we implemented the DAS game using Python 3.5 with the sockets and multiprocessing libraries (Dortmond, Wernke, and Rojer, n.d.). There are some small deviations from the proposed initial setup, which should not influence the experiments: single distributor instead of multiple, dynamically adding servers and the game tick-rate can be changed for reduction of the simulation time.

The experiments are carried out by simulation on a single machine, by using *localhost* with a port number as the IP-address. Each server, client or distributor instance is spawned as a separate thread, with no inter process communication except using sockets on the aforementioned port numbers. This means that the latency has to be simulated. This is achieved in the client process by waiting a fixed time before sending to and after receiving from a game server. The latency is set as the Euclidean distance between the server and the client. Both of which are initialized with an x and y position on a 1 by 1 grid, mimicking an actual geographical distribution of systems. The servers are placed in evenly spaced locations on this grid, in order to maximize coverage. The distributor uses this information to select the best server for a client, based the smallest communication overhead. Communication overhead (CO) is defined as:

$$CO = latency \times \#players$$

Where players is the total number of clients already being handled by that server.

The influence of latency on a user his experience can be quantified by looking at the number of invalid moves. Every move that is invalid, which includes a role-back on the clients end, means lost time and added frustration for the user. We will use the fraction of all moves that are accepted as the user-friendliness (UF).

$$UF = \frac{\#valid}{\#invalid + \#valid}$$

Here, UF = 1 means optimal service and UF = 0 means that no user was able to move.

III. Results

In figure 3 we see the influence of adding servers on latency. This shows that an increase in servers

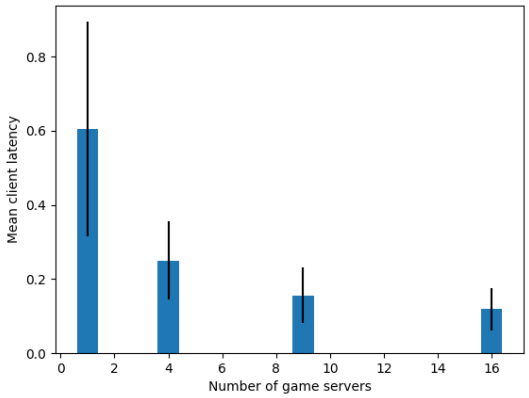

Fig. 3. Client-server latency as a function of the number of servers.

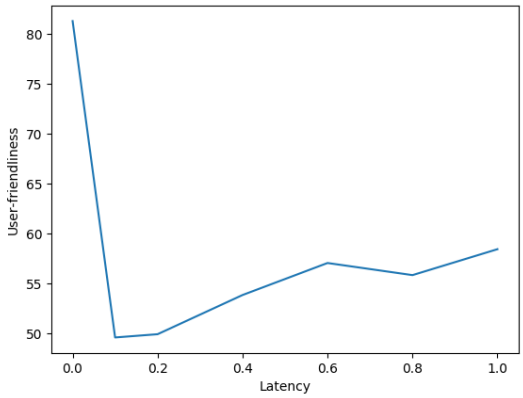

Fig. 4. User-friendliness as a function of client-server latency.

not only increases the compute power for game mechanics, but also significantly reduces the delay between sending and receiving a game update.

The impact of latency on user-friendliness for the DAS system is shown in figure 4. It was obtained by spawning 4 servers with 4 dragons and 12 users randomly placed on the grid. For each latency, the game was played 8 times for 60 seconds, where the latencies where fixed for each client. It reveals an interesting dynamic, which goes against the intuition of saying that lower latency is better for users. In the DAS setup, the clients wait for a response from the server on their previous action, which means that the number of actions per client is larger for faster connections, because the same game length is used. This number of actions as a function of latency is shown in figure 5. It shows a clear peak at a latency 0.15 seconds, which is the exact position of the unexpected minimum of user-friendliness in figure 4. This creates an interesting user-optimization problem. The best experience can either be had by ensuring extremely low latency, with a very small margin, or high latency, to give the user a less acceptable, yet constant game experience.

With the goal of optimizing human experience in mind, we searched for parameters resulting in an approximately 10 minutes lasting game, while supporting many players and dragons. This was found to be the case for 20 dragons on 5 servers and a final number of around 110 players. The clients were created with a sampled playtime length using the distribution as seen in figure 2 and with a

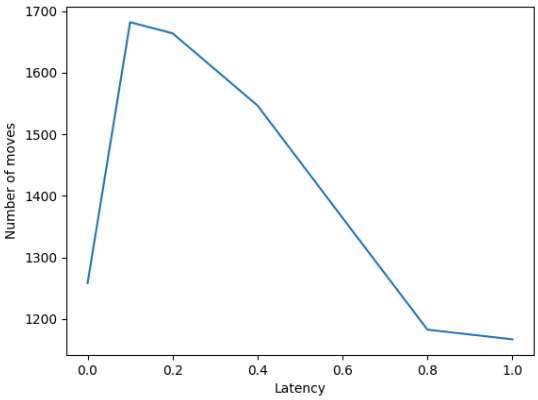

Fig. 5. Moves as a function of client-server latency.

constant join rate of 2 clients per second. This does require enough users to provide such a constant stream of clients.

IV. Discussion

Obtaining results using our experimental setup proved difficult, due to simulating on a single machine. The servers would unexpectedly write messages at the same time, creating a single faulty message. We did not take security into account for this project, meaning that unrecognized patterns could cause server crashes. We surmise that this will not happen if it is deployed as an actual distributed system, because the errors are created by incredibly high demand of the file-system. This demand reaches critical level at 100 players, which includes almost 250 threads. The erroneous nature of these issues meant that we could not benchmark

our infrastructure.

The experiments were made even more challenging by the influence of background processes on the speed of communication. We could define clear time frames in the results where the number of invalid moves suddenly spiked. We suggest that communication is the cause for this, because it is the most time dependant factor in the system. In order to reduce the influence of these bad simulations, we had to manually inspect each simulation result and run them many times over. This proved tedious and has a negative impact on the validity of the results, but we were fair and rigid in the removal of simulation results in unusable time frames.

The interesting dynamic found in decreasing latency could also suggest that the definition of user-friendliness needs to be adjusted. If we would take the number of moves per second as another requirement for the enjoyment of the user, it could show different results. In either case however, 0 second latency is best, as the results point out.

The design of the P2P synchronization might not have been ideal for this DAS problem, due to the large communication overhead. Other methods such as sharding can reduce this, by not having to share the moves for each player. The setup used here however, is a relatively simple implementation, which does well when computation overhead is large. This is not the case in DAS, a game with rules that are quickly applied.

This project leaves a lot of room for further development or research. First and foremost, the code should be supporting 2 distributors and the dynamical adding of servers. The latter could be further analyzed and optimized by receiving high quality updates from the servers and choosing a location for minimal latency.

Secondly other important research areas of distributed computing have been neglected, such as: secure communication, where messages get encrypted to ensure both parties are who they claim to be, or multi-tenancy, by running multiple games on the P2P network.

V. Conclusion

The implementation of the designed DAS peer-to-peer game servers with load balancing of clients by a distributor, shows great promise with respect to user-friendliness and scalability. The former seems to be optimal for small latencies, but quickly drops of to a steady experience. The latter proves difficult to benchmark on a single system, due to the high communication costs.

References

Dortmond, M., P. Wernke, and K. Rojer (n.d.). *distributed systems DAS game*. URL: https://github.com/neverbeam/distributed-systems.

Funkhouser, Thomas A. (1995). "RING: A Client-server System for Multi-user Virtual Environments". In: *Proceedings of the 1995 Symposium on Interactive 3D Graphics*. I3D '95. Monterey, California, USA: ACM, 85–ff. ISBN: 0-89791-736-7. DOI: 10.1145/199404.199418. URL: http://doi.acm.org/10.1145/199404.199418.

Guo, Y. and A. Iosup (2012). "The Game Trace Archive". In: *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pp. 1–6. DOI: 10.1109/NetGames.2012.6404027.

Newzoo (2018). "2018 Global Games Market Report". In: pp. 1–6.

Suznjevic, M. (2013). *WoWSession game trace archive*. URL: http://gta.st.ewi.tudelft.nl/datasets/gta-t3/.

Appendix

The hours spend working on the DAS for all members of the team.

	Martijn	Patrick	Kevin
think	4	8	3
dev	68	57	5
xp	10	18	0
analysis	3	6	2
write	4	6	12
wasted	4	3	4
total	93	98	25