Dragons Arena Backbone

A distributed serving system for real-time MMOs

Authors
Chris Froussios
4322754 - c.froussios@student.tudelft.nl
-
Ivanis Kouamé
4342122 - kouame.i@student.tudelft.nl

Many thanks to
Alex Iosup
Yong Guo

Abstract
The idea behind this project comes from the fictive Want Game BV. They have requested distributed designs for serving their real-time MMO, Dragon Arena. Here, we propose a design that minimises latency, has parameterised fault tolerance and scalable message complexity.

Introduction

The popularity and ambition of modern MMOGs (*massively multiplayer online game)*, with thousands of players from across the globe playing in ever more complicated virtual worlds, has made it necessary that such games are powered by distributed systems.

The fictitious game-developing company WantGame has designed an MMO game, named Dragon Arena, and is currently exploring the possibility of having it backed by a distributed system, interconnected via a privately owned high-performance network. The goal of this project is to design the system that will allow players to play this game over a distributed service.

We will begin by describing in more detail the system and its requirements. Then, we will detail our proposed solution to Dragon Arena's requirements. Next, we will present the testing of our design. Finally, we will close with a discussion of our design and our recommendation to WantGame, with regard to the feasibility of serving Dragon Arena though a distributed system.

Table of contents

I. Background on application

1. DAS (Dragon Arena System)

The project is an action game that involves humans who are the players against dragons who are controlled by an AI (Artificial Intelligence). The goal and winning condition is for all participants in these battles is to eliminate all of the units of the other side.

2. System requirements

a. Scalability

The system must be able to perform on a game with 100 players (i.e. clients connected to a server), 20 dragons and 5 server nodes. Players and dragons have a maximum action frequency.

b. Fault-tolerance

A client may fail either by crashing or by network failure. The game must continue for the rest of the players while a particular player is experiencing difficulties. Servers may fail by crashing or losing connection to any of the other system components. The system must provide guarantees of consistency for any such single failure, and up to given number of simultaneous failures.

c. Consistency

Clients must have access to the entire state of the game and the state of the game must be consistent through all of the working network entities. That means that the state is to be kept in all entities and be refreshed in case an event arrives. The system must tolerate varying network delays.

d. Performance

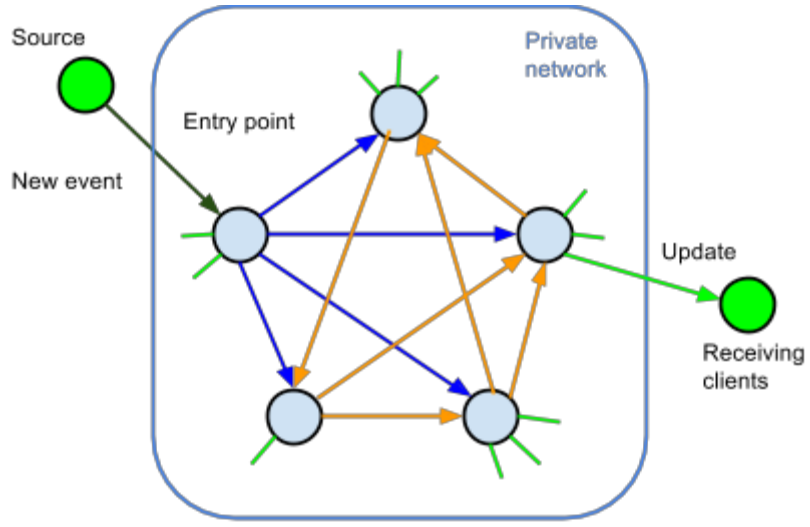
We understand latency to be the first priority. The system must make updates available at the earliest moment allowed by standard network delays.

II. System design

Our design separates participating applications into two levels: outer level (clients) and inner level (servers). Clients "play" the game (by being either players or dragons) and use the inner level as a service, submitting their actions and receiving others' actions. Servers cooperate to guarantee fast delivery of all the events to all the clients. Servers and clients alike maintain a local running instance of the game.

All programs involved on the inner level are equal in role and functionality. Each server uniquely serves a number of clients. Clients inform their server on actions (or "events") they have performed, the server being the entry point for that action into the inner level. Following a basic validation of the action by the entry point, the event is the broadcasted to each other server. The first time a server is made aware of an event by a peer (i.e. the event is already in the inner level), it relays to every client it serves, and every target within their window, which is a redundancy measure that we will discuss now.

Servers acting as entry points broadcast events, making the inner level network a complete graph. When receiving events from other peers, they assume a role in a virtual ring. If the event is new to them, they forward it to the next *W* peers, where *W* is a predefined number called the window size. In this way, they ensure the next *W* peers are consistent.



A graphical example of an event being propagated through the system. Green nodes are clients, while cyan nodes are servers. Green arrows are updates delivered to clients. Blue arrows are the original broadcast by the entry point. Orange arrows are messages forwarded in the virtual ring (here the window is 2)

1. System overview

a. Scalability

Each unique event that enters the inner level results in *n*(w+1)* message exchanges among servers, where *n* is the number of servers and *w* is the window size. Each unique event also results in *p* messages from the inner the outer level, where p is the number of clients in the game.

If *r* is the rate at which clients (including players and bots) perform actions (i.e. produce events), the system's message complexity is *O(r*p*(p+n*w))*, out of which, *O(r*p*n*w)* messages are transmitted within the private network that connects the inner level. As we can see, the system scales exponentially with the number of clients served, and linearly with regard to the number of servers and the standard of fault tolerance.

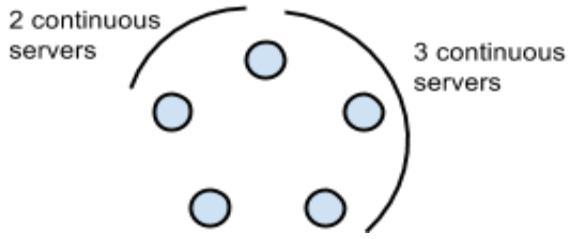
b. Fault tolerance

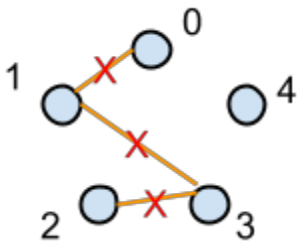
In the event of a server crash, the game continues and only clients connected to the crashed server suffer. They may resume their role by reconnecting to another server. If the server crashes while an event was being broadcasted, as long as one running server has received it, the broadcast will be completed, as the event will be forwarded through the virtual ring. This is always true if the window is greater than *2*. With a window of less than *2*, the chain will break at the crashed server and the system may lose consistency. More generally, the system can tolerate simultaneous crashes of up to *W-1* continuous* servers, *W* being the window size. If broadcasts are not interrupted, the system can operate with down to just one server running. Note that our system does not guarantee the atomicity of broadcasts, favouring speed instead.

In the event of a client failure, their character in the game remains idle while the game continues.

In the event of a point-to-point network failure, broadcasts are completed by having messages forwarded by the preceding nodes in the virtual ring. The system can tolerate network failures from one node to up to W-1 continuous nodes.

** with continuous servers, we mean a range of servers in the virtual ring. See following graphic*





#0, #1 and #2 have point-to-point failures to one continuous server.

#3 has point-to-point failures to two continuous servers.

c. Consistency

Our design employs eventual consistency. Every machine maintains its own running instance of the game and all events are replicated in every machine and accepted locally as soon as they become available. Game instances tolerate late and out-of-order events by using the Trailing State Synchronisation (TSS) algorithm [1]. TSS is an optimistic synchronisation algorithm that maintains older game states and uses them to retrospectively recover, if the current game state is invalidated by a late event. Events are ordered by a scalar clock. Actions that were issued under bad information are ignored by the game instance, if they are invalid. If they are valid, even after the context has changed, they are still accepted. This practice is acceptable because, within the rules of Dragon-Arena, there are no self-harming actions.

d. Performance

With regard to latency, the immediate broadcast of an event by the entry point and the immediate relay of events to clients ensures that clients are made aware of events at the earliest time allowed by network speeds.
With regard to data transmitted, in our design, complete game states are only exchanged in the initialization phase. After that, only updates are transmitted.

e. Inserting nodes into the inner level (extra)

The inner network is assumed to be established at the start of a session. However, we have added a provision that allows new or recovered nodes to enter the network. The recovered node (manually selects) another node to be synchronised from. The synchronisation is performed by transmitting the complete current game state from a running server to the recovered one (all servers maintain a running instance of the game). To ensure that the recovered node does not miss events that were in the process of being relayed through the inner network, the running server also pushes all subsequent new events to the recovering node for a fixed amount of time. For an absolute guarantee that no events are missed, the fixed amount of time mentioned should be equal to the maximum delay tolerance of the TSS implementation.

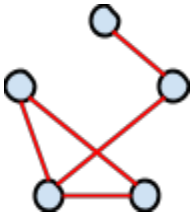
III. Experimental results

We were unable to establish or acquire access to an environment with multiple machines. Our first option, the DAS-4 system, was unavailable to us. Our second option, the Amazon EC2 service, was a very problematic environment for us. Our applications, implemented with Java RMI, suffered from a very bad quality of service and even minimal test cases were non-operational. After spending a lot of time trying to overcome our problems with the environment, we decided that we lacked the experience to make use of the EC2 environment within the frame of this project.

Our experiments were performed on two connected machines, each running multiple instances of servers and clients. This setup did not allow us to experiment with natural latencies, bandwidths and network failures, so studying those aspects of performance was deemed pointless.

We were only able to confirm that our implementation met the theoretically expected behaviour and fault tolerance, by manually simulating server crashes and point-to-point connection failures. We experimented with 5 servers, each serving a small number of clients.

In one experiment, the window was $W=2$. We confirmed that the game continues normally if one server fails, or if network failures happen, where one server loses connection to one one other server. For a window of size $W=3$, we again confirmed that broadcasting of events are completed if servers each lose connection to up to $W-1=2$ continuous servers. To the right is an example of the setups that were confirmed to work (red denotes point-to-point network failures).



We suspect that with such dense failures, that rely on forwarding through the virtual ring to complete broadcasts, the performance of the system with regard to latency will suffer significantly. However, all latencies were insignificant in our test environment.

We intended to run tests on scalability with regards to players. However, bot programs are non-negligibly taxing on both memory and computing power. Our machines would typically become overwhelmed by more than 30 bots each, which is below the benchmark goal of 120 bots. We were unable to determine the source of the strain, but we suspect it comes from the cost of running too many instances of the game world on each machine.

IV. Discussion

Our design has several theoretical advantages. With regard to performance, our design minimises latency: events take the shortest route through the private network, without the need for the event's propagation to be blocked at any intermediate stop. This trait is very important for real-time games, such as Dragons-Arena, since latency has a direct impact on the player experience. We are satisfied with the properties of our design in this regard.

We consider this design to be scalable. Message complexity scales linearly with the number of servers. This is not the optimal complexity for a broadcast. However, we considered this an acceptable trade-off, in the interest of minimising latency. Latency is constant. A foreseeable bottleneck for the system is scalability with regard to number of players served. We have already seen that message complexity scales exponentially with regard to the number of clients served. This would have been the case in any implementation we can imagine, where all players share the same view of the game. The only way to alleviate this problem is partitioning the game. Our current design does not support such a practice.

Our design makes good use of resources and can adapt to demand. With the addition of a mechanism that can fairly distribute clients between servers by directing them where to connect, the load on different machines could approach full balance. All machines involved in the inner level share the same functions and responsibilities. Topical imbalances in demand can be addressed by deploying more servers in the same region. Temporal imbalances can also be addressed: machines can enter the network while a game is in progress and accept new players. Under this design, however, it is not possible to have machines exit the graph without disrupting the experience of the clients they serve, which have to be disconnected and reconnected to another server. Still, the effect of this process can be alleviated by increasing the complexity of the client: a client can connect to two servers and receive updates from both, pushing events only to one, while preparing to drop the other server.

Overall, our design meets the goal of minimising latency, while keeping consistency. The biggest disadvantage of this design is the inability to partition players within the same game. Dragons-Arena is of finite size, and if WantGame does not intent to increase the size of an individual game, this should not be an issue and our distributed design can be adopted. Otherwise, WantGame should explore alternatives that enable players to be partitioned, for example by location in the game world.

Conclusion

In this report we described a simple MMO game that was to be served by a distributed system. We proposed a design that fulfils the requirements and implemented it. We confirmed that our system can support games of Dragons-Arena in their current form. We considered the potential of our design and, in the end, recommended that WantGame considers our design for adoption, provided that Dragons-Arena maintains its finite size.

Appendix

Time sheets

	Christos Froussios	Ivanis Kouamé
Total time	98 hours	72 hours
Think time	11 hours	8 hours
Dev time	45 hours	40 hours
Xp time	7 hours	7 hours
Analysis time	2 hours	0 hours
Write time	10 hours	7 hours
Waste time	25 hours	10 hours

Source code

The code of our implementation is publicly available at GitHub at https://github.com/Froussios/DCS-Dragons-Arena

References

1. Cronin, Eric, et al. "An efficient synchronization mechanism for mirrored game architectures." *Proceedings of the 1st workshop on Network and system support for games*. ACM, 2002.