DAD-OGP - Fault-Tolerant, distributed online gaming platform

Gonalo Garcia
Instituto Superior Tcnico
Desenvolvimento de Aplicaes Distribuidas
goncalotgarcia@tecnico.ulisboa.pt

Abstract

Multi-player video-games are one of the cornerstones of the gaming industry. Nowadays developers have to build systems capable of withstanding a massive influx of players around the world, while still providing the same quality of experience as their single-player counterparts. We introduce DAD-OGP, an online gaming platform that focuses on multi-player Pacman, with additional social features, to provide an integrated gaming and communication experience.

1. Introduction

In recent years, the scale of online gaming platforms has grown to immeasurable numbers. The increasing number of players combined with an ever expanding geographical distribution, caused by the enormous popularity of online gaming, has brought very strict performance and availability requirements to these systems.

Players expect extremely low latencies, to a point where it's difficult to tell a difference between single and multi-player experiences. This requires extremely efficient code to hide the overhead caused by the networking operations.

Equally important is the capability to offer close-to-zero downtime, by hiding and working around faults in the game's servers as well as in the client counterparts.

In this paper we describe DAD-OGP, an online gaming platform that provides a fault-tolerant, distributed gaming sessions and chat capabilities while maintaining consistent state across all connected clients.

DAD-OGP uses a passive-replication protocol which allows the system to hide omission faults, caused by errors in the servers. This protocol relies on a leader that connects to clients and shares the game's state with a replica set. The messaging functionality relies solely on the clients themselves and provides causal ordering of messages in a peer-to-peer setting.

Section 2 presents the system's architecture and the fault-tolerance protocols are explained in Section 3. An evaluation of the system's performance is further described in section 4.

2. Architecture

DAD-OGP is comprised of four distinct modules: A server, a client, a testing and deployment console called PuppetMaster, and a deployment server named Process Creation Server.

Given that PuppetMaster and Process Creation Server are both well described in the system's proposal, the focus of this section will be on describing the architecture of the Server and Client components.

All components have been written in C# and make use of the .NET Remoting framework's Remote Method Invocation capabilities to communicate among each other.

2.1. Client

2.1.1 Game Client

In the gaming aspect of DAD-OGP, the client component is merely a presentation layer for the result of the computation preformed by the server. The Client has the responsibility of collecting the user input and sending it to the server in order to be processed.

In order for this communication to be possible, the Client holds a proxy Server object that is the same across all instances. This assures that all clients will be communicating with the same server.

When connecting to the Server, the Client announces it's services and provides its RMI URL so that the Server is able to build a proxy object and use the Client's public API to share the game's state for each round, and perform operations required for fault tolerance.

When receiving a message from the server, it is placed in a queue from which a single message is dequed at each iteration of the game. The message (shown in Fig.1) contains

the round number and a list of every game-element's position, which is used to update the position of the on-screen elements.

```
public struct GameStateMsg
    {
        IList<PlayerPosition> Position;
        long roundTimestamp;
    }
```

Figure 1. Message containing the game's state

The processing of each of these messages is done asynchronously to improve the performance of the presentation.

2.1.2 Chat Client

DAD-OGP's Client module also includes a chat-room that ensure causal-ordering of messages. This chat uses a fully peer-to-peer protocol, and as such, it's necessary to be able to communicate with the peer Clients. To do this, once again, we use proxy objects. Whenever a new Client connects to the Server, its RMI information is sent to all of the previously connected Clients.

Causal ordering is ensured by the usage of vector timestamps. Each message contains the sender's vector timestamp at the time of sending. The algorithm[1] for ensuring causal order works as follows:

(i) Each vector initializes all positions with the value 0.

(ii) Before broadcasting a message, the sender increments its corresponding position in the vector by one.

(iii) When a message is received the system checks if there is a position of the received vector that is larger than its own vector by one unit, while all others are equal. This means that the received message is correctly ordered and can be delivered. Otherwise the message is buffered.

(iv) If the received message is not buffered, the system will attempt to deliver messages from the buffer by repeating step for each old message(iii).

(v) The new vector timestamp of the Client is the timestamp of the last message delivered.

```
public struct ChatMsg
    {
        string msg;
        int[] vector;
    }
```

Figure 2. Message shared between chat peers

To simplify these operations we built a custom CausalOrderQueue that provides non-blocking methods to dequeue messages in causal order.

2.2. Server

The Server is the main component of the DAD-OGP platform. It's tasked with receiving client updates, and updating the game's state accordingly. This includes calculating new positions for all clients and in-game objects, as well as updating scores and player deaths.

When deployed, the server immediately advertises it's RMI functionality to the .NET Remoting framework. This allows Clients and other Servers to connect to it by generating proxy objects. In order to ensure that the state is consistent across all proxies, a specific instance of the server is shared, as opposed to using the framework's proxy alternatives.

When started, the server will then wait for the specified number of Clients to connect. This makes sure that no group of players will start the game before any other player. Once all players are connected, the Server will then begin its execution loop, described below:

(i) Wait a specified amount of time for Client inputs. All inputs received outside this window will be ignored.

(ii) Compute the new positions for all players and ghosts.

(iii) Check if any player has died or scored a point and update the state accordingly.

(iv) Perform fault-tolerance related operations (described in section 3).

(v) Send the updated state to each client.

This process will be repeated until the game finishes. The algorithm ensures that the game's state is consistent across all clients, since it's calculated in a centralized manner, and only then propagated.

This means that even if the client is slow, and refreshes the game at a lower rate, it's local state will only be late when compared to the global state, as opposed to transforming into some other arbitrary state.

The game's state is stored in the form of a Dictionary that maps the object's ID (which can be a Client's PID or simply a ghost's name) to a PlayerPosition struct. Once the round is over, each player's struct is compiled into a list which is sent to the clients, as seen in section 2.1.1.

```
public struct PlayerPosition
    {
        int X, Y;
        string Name;
        Moves Move;
       public int points;
    }
```

Figure 3. The struct containing the information about each object

The server attempts to perform all game-state related operations asynchronously, in order to reduce the amount of time lost between rounds. This is necessary as the server does not start listening for input on a clock's tick, instead waiting until every following operation is completed.

3. Fault Tolerance

The real-time nature of DAD-OGP requires very high availability and survivability against crashes. Without any form of fault tolerance, a crash of the Server, or even one of the Clients would be a disaster, as the game would immediately become unreachable to all other Clients.

More than that, since Clients can connect to all parts of the world, it's important that the fault-tolerance protocol can distinguish between an omission and a high-latency connection between two peers.

We propose a passive-replication fault-tolerance algorithm that is based on a leader and several replicas that interact with each other to detect faults. This protocol is proposed in order to abstract the fault tolerance process from the Client which will only know when a new Server connects.

3.1 Boostrap

The bootstrapping process is commandeered by PuppetMaster. The first Server to be deployed immediately becomes the leader of the set and it's information will be shared with all peers, including replica Servers and Clients.

The following Servers will be considered replicas and will not contact Clients. When deployed, the replicas will announce themselves to the leader so that the game's state can be shared with them.

3.2. Leader

The leader is the most important Server of the set, and it is tasked with running the game-updating algorithm seen above.

The fault-tolerance operations mentioned in the previous sections are simply replicating the game's state with the all the connected replicas.

Once a new replica is connected, the leader is tasked with sending it's RMI information to all previously connected replicas. This is necessary as all replicas must interact in case the leader is presumed dead.

This replication is performed before the propagation of the state to the Clients in an attempt to ensure that the replicas will always have the latest state and that no Client will be ahead of the replica set.

3.3. Replicas

While the system works correctly, the replica Servers are mostly idle. The work is put upon the the leader to share new replica's added to the set as well as state information.

Every time the leader sends a state replication message, a timer is started on each replica. This timer has a duration of 5 x MSEC_PER_ROUND, which is the maximum expected latency of a state replication message.

When a new state replication message arrives, the timer is restarted and the process repeats. Only if no new message is received in the above interval is the leader presumed dead and the election process is started.

3.4. Leader Election

The leader election algorithm was designed to work as fast as possible and with the least amount of messages exchanged. These constraints were devised due to the real-time nature of the game, which should endure the smallest possible number of election related pauses.

Once a new election is triggered, each replica will send a message to all its peers containing its PID, the last round number received, and a random number between 0 and 999999999 that works as a challenge. To make this challenge as random as possible, the RNG's seed is the Environment's Tick Count multiplied by the Thread ID.

Once a replica has received all of its peer's messages, it will compare all of them and the new leader will be the one who has the best combination of parameters in the following order of importance:

(i) Round ID

(ii) Highest challenge

(iii) Largest PID

Once this algorithm is completed, the leader will be the same across all replicas. Allowing the whole process to be repeated correctly when needed, at a later time.

```
public struct VoteMsg
    {
        string name;
        long roundTimestamp;
        long challenge;
    }
```

Figure 4. Message shared between Server replicas

The new leader will then start the promotion process. This means it will try connecting to the old leader again, and if it succeeds, the old leader will be demoted and added to the replica set. The new leader will then reset the state of all the other replicas and broadcast a message with the RMI information of all replicas in the set, which will reset the fault-tolerance protocol.

The new leader will then connect to the client, send its latest state, and begin receiving new inputs.

In order to tolerate replica faults, once a replica broadcasts its vote, it will start a timer with a duration of 2 x MSEC_PER_ROUND. This is the maximum expected time for all replica votes to be received. If all votes are received in this time frame, the timer is stopped, otherwise the election process will be started with the messages received at that time.

This election process guarantees that there will only be one leader at any given time, and that the leader will be the same for all replicas.

3.5. Evaluation

This project was tested with a maximum of 6 clients, refreshing at 20ms, connected to a set with one leader and two replicas. At this level, there can be seen some ”lag” in the refreshing of the presentation, which is noticeably reduced when players begin dying. While this is noticeable, the game remains consistent across all nodes, and there is no type of desynchronization happening. Reducing the refresh rate will result in a reduction of the ”lag”, but the overall game will be a slightly slower. The best tradeoff found during the evaluation of the system was to set the refresh rate in between 30 and 40ms, resulting in a smooth game which isn't too slow.

On the server side, the fault-discovery and leader election process is very efficient, and a new leader can be elected and connected to the client in under half a second and without any noticeable lag, in the case of a false fault-detection. If a true fault happens there may be a pause in the game, as the new leader will try to connect to the previous leader before connecting to the clients.

This step is necessary to make sure that there aren't two leaders, but as a further improvement it could be altered to execute asynchronously, and this way make the election process less noticeable.

Something that may be noticed is that there can be a lot of elections in the Server set given the short timeout. While this is not ideal, the efficiency of the process allows the game to run quite smoothly, even if a new leader is elected every 5 seconds.

As mentioned in a previous section, there is a possibility for a Client to trail behind it's peers when displaying the game. This is more noticeable when a user moves the game window on the screen, which triggers a slight pause in the Client's loop. This is only noticeable when comparing two Clients side by side, and should not be considered a desynchronization of the game, as all events that happen in a single client will happen in all clients, even if slightly late. To counter this, an optimization would be to remove the event queue and update the game immediatly upon receiving a message from the Server.

4. Conclusion

The field of online-gaming protocol design is ever increasing in the number and scope of new algorithms. This is due to the importance of multi-player in today's gaming landscape.

New protocols try to reduce the latency of the game, to provide player's with a true sense of real-time control, all while tackling fault-tolerance for (sometimes) millions of players scattered throughout the world.

DAD-OGP tried to provide the features mentioned above at a smaller scale, with pleasant results. As a proof-of-concept it works well, and could be extended to provide much more functionality and scalability.

5. References

[1] Cloud Computing Concepts, Part 1 - University of Illinois - Coursera