DAD-OGP

Tomás Cunha
81201
tomas.cunha@tecnico.ulisboa.pt

Guilherme Santos
81209
guilherme.j.santos@tecnico.ulisboa.pt

Abstract

In modern society, online games have become more and more common. In this paper, we introduced a solution for easily creating fault-tolerant distributed games in semi-synchronous environments (synchronous between servers, asynchronous between clients). Game clients employ a prediction mechanism to mask delays in the server. The server may be replicated across an arbitrary number of replicas that take its place when a failure is detected.

1. Introduction

With the advent of the internet, multiplayer games are becoming mainstream. With this in mind, it is important to design solutions that minimize latency and tolerate faults so that the player cannot perceive any downtime in the servers.

It is also desirable for the players to communicate with each other during the course of a game. This means that message order causality must also be taken into account.

2. Overview

Our system is based on an abstract server class that employs a round-based approach to the game cycle. This abstract server implements the communication logic; every time a new game needs to be added, the developer only needs to subclass this abstract server and implement the game-specific domain logic.

The same approach is taken on the client: we have an abstract class implementing the communication logic, and developers wanting to support a new game need only to implement the game-specific logic.

2.1. Main Game Loop

At the start of the game, the server waits for all the players to connect. When a player connects, the server contacts all currently connected players and announces the new player to them; this is because players need to know every other player in order to communicate with them (the chat is implemented in a peer-to-peer fashion).

When all players have connected, the server calculates the initial state and sends it to all players.

At this point, the server enters a loop until the game is over. In this loop, it gathers inputs from the clients, calculates states according to these inputs, and then sleeps for some amount of time (this time is configured when the server is started, and corresponds to the round time).

This loop may be represented in pseudocode as follows:

Algorithm 1: Main game loop

```
1 while running do
2 |   sleep(ROUND_TIME);
3 |   calculate_state();
4 |   round_time := round_time + 1;
5 |   broadcast(state, clients);
```

Out of the functions used in this loop, only **calculate_state** needs to be implemented by the specific game servers. This massively simplifies the creation of new games, since the load of managing the distributed side of the application is taken care of by our abstract class.

2.2. Client-side Overview

Similarly to the server, there is one abstract class representing a client. This class uses several abstractions to help make it more extensible. All the logic for processing received states and sending inputs to the server is done in this superclass; subclasses need only to implement the logic for drawing the game board on the screen, based on the states that they receive.

This means that, just like in the server's case, the creation of new types of games is simple: a developer meaning to create a client for a new type of game only needs to worry about what sprites to draw in the positions given to it by the server; anything else is already handled by the abstract client.

Additionally, the abstract client class takes of care of receiving and sending chat messages in causal order, meaning that if a message was sent after another message was received, these two messages will never be received in the wrong order by a third party who receives both of them (see 3.1).

3. Client side concerns

3.1. Chat message ordering

To implement causal ordering, we used an implementation of vector clocks as timestamps to order messages, as described in [1].

Since the number of clients in our system is known at the start of the game, each client can create and zero-initialize a vector of scalar timestamps that is used to order every message that is received. The implementation can be summarized in the following pseudocode:

Algorithm 2: When message m is received by client i

```
1 if is_successor(m.clock, i.clock) then
2 |   deliver(m);
3 else
4 |   queued_messages := queued_messages ∪ {m};
```

In the above pseudocode, a message's clock is considered to be the successor of the client's successor if, for every timestamp except for the one of the client that sent that message, the client's timestamp is greater than or equal to that timestamp, and if the message's timestamp for the client that sent it is the successor to the corresponding timestamp on the receiving client's clock.

Algorithm 3: When message m is delivered to client i

```
1 final_deliver(m);
2 to_send := ∅;
3 foreach m ∈ queued_messages do
4 |   if is_successor(m.clock, i.clock) then
5 |   |   to_send := to_send ∪ {m};
6 |   |   queued_messages := queued_messages \
  |   |     {m};
7 foreach m ∈ to_send do
8 |   deliver(m);
```

With these algorithms, we ensure that there will always be a causal ordering between messages; if a message m_2 depends on a message m_1, then all clients will receive m_1 before they receive m_2.

3.2. State Prediction

In order to mask away delays in the network between the server and the client, we employ a prediction mechanism in the client. After a certain amount of rounds, if the client hasn't received the next state from the server, it will use the current state it has to simulate the next state. When the server's state reaches the client, the calculated state will be corrected if need be.

This way, if the messages between the server and the client suffer some temporary delay, the user will not immediately perceive this, as the next state will be calculated on the client's side. Of course, the client can't guess what movements the remaining players made without the server telling it, so it might mispredict what the real state actually is. This misprediction is corrected when the state updates the client with the real state, but at least the client doesn't see a frozen screen during this delayed period.

This was done by moving the state calculation logic into the game state class that is shared between the client and the server; the server is usually the one calling that calculation method, but if the client perceives a delay in the connection, it will use that method to keep calculating states.

4. Server side concerns

4.1. Replication

In order to ensure that the game keeps going in case of a server failure, we needed to decide on a replication protocol.

We had several alternatives; one of those would be to use active replication: every round, the clients would gather states from a majority of replicas and then draw the state. This solution, however, was not the best for our purposes for a few reasons:

- the client would need to be aware of all the replicas, which would introduce some additional unnecessary complexity and overhead.
- the client would also need to wait for a majority of responses from the servers before drawing the new state, which might be too slow for our purposes (since latency is also a concern in this system)

For these reasons, active replication was not the correct option, so we opted for passive replication.

By assuming that the system is synchronous between servers, the problem is simplified. This assumption is not far-fetched, given the strict time constraints that game servers are supposed to satisfy. If we assume this, perfect fault detection is possible: the server maintains a list of replicas, and periodically send them a message informing them that it is still alive. Whenever there's a state update, it

sends it to all the replicas as well. When the server fails, the replicas elect a new primary server. Repeat this process until there are no replicas left. Using this protocol, if we have f replicas (including the primary server), we can tolerate $f - 1$ faults.

The only remaining problem, then, is electing a new primary server. This is trivial, in this case: we can just assign a unique id to each replica (for example, the IP address and port of that particular replica), and order the list by that id; then, when a failure is detected, the first element of that list is selected as the new primary server. If that replica also failed, it is okay, as this fault will also be quickly detected and the next one on the list will become the leader.

5. Conclusion

In a multiplayer game environment, it is of the utmost importance to reduce latency while preventing the game from being interrupted by a failure on the server's side. Our proposed solution takes care of the latter without compromising the former, by employing passive replication on the server's side along with prediction mechanisms on the client's side.

References

[1] C. J. Fidge. Timestamps in message-passing systems that preserve partial ordering. 10:56–66, 02 1988.