Asynchronous Real-time Multi-player Game with Distributed State

Glen Berseth, Ravjot Singh

April 21, 2015

Abstract

Multiplayer games are awesome but they have a single point of failure and are not very scalable.

1 Introduction

We propose to construct a distributed *client-server* model in order to gracefully handle fail-stop scenarios. We will use this system to support a simple computer game of a number of agents moving around in a 2D world. We assume no limits on bandwidth and have strong constraints on latency which significantly impacts online game experience [Claypool and Claypool, 2006].

2 Related Work

To build an online real-time multiplayer game, there are two popular approaches known. One approach is *peer-to-peer lockstep*, is based on *peer-to-peer* architecture. In this approach, all clients start in the same initial state and broadcast their actions. The overall performance of the system is dependent on slower clients in the system. Moreover, since the clients use broadcast methods to communicate, the approach has high message complexity.

To achieve better real-time simulation systems switched to a *client-server* model (see Figure 1(a)) [Coleman and Cotton, 1995]. With this model the state of the game is stored on a server and clients send updates to the server. This works well as latency is determined by the client to server connection. However, this was still too slow for real-time online games, which lead to the introduction of client-side prediction [Bernier, 2001]. Simply put client-side prediction allows the client to simulate its own version of the game (sending the results to the server) but the server can still step in and override the client's game state. This creates complexity in handling server overrides on clients smoothly (not just in code but also in animation and audio). Another reason for this model to gain traction was the ability to handle malicious clients by validating all actions on the server.

3 Methodology

In this section we outline our solution to our distributed state synchronization design. We discus what protocols are used, how the clients and servers communicate and the example game constructed for this system.

3.1 The Client

The client locally simulates its own version of the game. To support this the client needs to keep a list of the agents controlled by other clients in the game and the most recent location of each agent. This information will be stored in a *hashmap*. After every frame is rendered in the game the client will send a position update to all servers.

3.2 The Server

The server simulates its own version of the game. The server is not used just to reduce message passing but also to act as an authority over clients, to prevent malicious clients from propagating invalid information.

In the simulation loop for the server a number of actions occur

1. the server accepts event messages and puts them in a queue for processing
2. The server accepts position updates from clients
3. The server verifies these updates to be valid
4. The server's local copy of agent locations is updated
5. The server processes any events in its event queue
6. If events are valid, update server's game state and broadcast the result

This way the server keeps the true state of the game and informs the clients of valid updates to the server's state. To perform these operations the server needs to have state for the currently active clients and its own version of the game state.

3.3 The Game

We will simulate a very basic game to use as our state to synchronize. The game has two possible actions $updateLocation(a, p)$ and $fire(a_i, a_j)$. These actions can be executed at any point in the game but the server must validate the actions.

In order to simulate the game, information is needed on the other agents in the game. The only data stored for each agent is the agent's current location. The information needed for the game will be provided from the server or client the game is being simulated on. Computer animations and therefore game simulations use simulation time, similar to a vector clock, for synchronizing events.

3.4 Distributed Servers

We construct a distributed server model (see Figure 1(b)) to enable better failure handling in our system. Each client will be paired with a server. Each server will act as the authoritative server for a subset of the clients. The authoritative server for a particular client will depend on the event/action being processed.

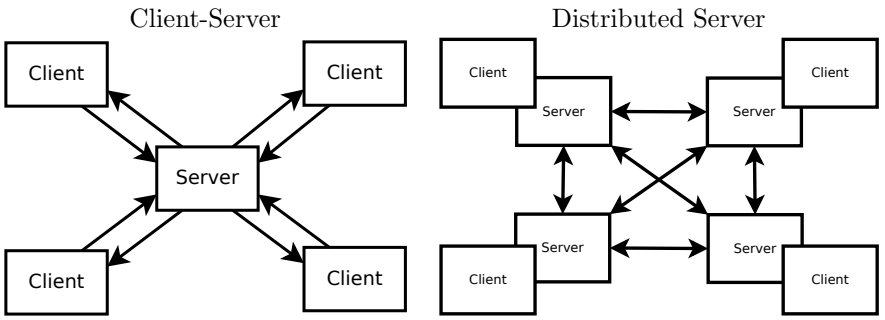

Figure 1: The first model (a) is an example *client-server* model. In this model all of the clients send updates to the server and the server send updates out to the clients. In the second model (b) the server is distributed and clients communicate with many servers.

3.4.1 Protocol and Messaging

The clients send updates/events to every server. For $fire(a_i, a_j)$ events the server that is paired with the client with a_j will determine the outcome. If the $fire(a_i, a_j)$ event is successful according to the authoritative server a $destroy(a_j)$ event is broadcast to every server. All communication is asynchronous without acknowledgements, except for the $destroy(a_j)$ event.

4 Prototype

In this section, we will describe the architecture and other details of the prototype system we developed.

4.1 Data Structures

Describe the data structures used

4.2 Flow of Controls

Describe the flow of control in server and client, and how they interect with each other.

5 Evaluation

We plan to evaluate our method in two ways. Primarily, the methods ability to handle failure. To cope with the failure gracefully and cause the least amount of disturbance in the gameplay as possible. A secondary goal is to keep the latency of the game down as more clients and server are added to the system.

6 Conclusion

Delivery Dates:

1. March 21: Iteration 1 (Single Server Prototype)
2. April 02: Iteration 2 (Distributed Server Prototype)
3. April 05: Project Report and Presentation Slides
4. April 12: Testing and Refining

References

[Bernier, 2001] Yahn W Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, volume 98033, 2001.

[Claypool and Claypool, 2006] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, November 2006.

[Coleman and Cotton, 1995] Scott Coleman and Jay Cotton. The tcp/ip internet doom faq. *www.faqs.org*, 1995.