Asynchronous Real-time Multiplayer Game With Distributed State

Glen Berseth, Ravjot Singh

April 23, 2015

Abstract

Real-time multiplayer games are complex systems that often have a single point of failure and are not scalable. In this work a prototype design is created to handle node failure during game simulation. The client server paradigm is modified to construct a distributed server at each node. Propagation of *gamestate* is performed across nodes keeping each node up to date. Node failure is handled gracefully without noticeable suspension of gameplay. Using distributed state across nodes also shows promise in the area of scalability.

1 Introduction

To build an online real-time multiplayer game, there are two popular methods. One, is *peer-to-peer lockstep*, which is based on the *peer-to-peer* architecture. In this approach, all nodes start in the same initial state and each node broadcast every move to the all other nodes. The overall performance of the system is dependent on slower nodes in the system. Moreover, since the nodes use broadcasts to communicate, the method generates a significant volume of messages.

A second method is based on the client-server architecture, in which the game is also simulated on a server and each clients sends updates to the server. The drawbacks of this method is that the server can become a bottleneck to the system performance and is a single point of failure. From the point of view of support for this system there needs to be a large investment in infrastructure. Also, due to additional game simulation load on the server, this approach is not scalable.

Cheating is serious problem in online games. The gameplay needs to be fair in order to keep players interested in the game. Cheating can be done by players in the game sending information to other clients that should not be possible according to the current game state and invariants on the capabilities of the players in the game. In the *peer-to-peer* architecture protecting against cheating is very difficult as clients send information directly to other clients. The *client-server* method handles malicious players by simulating the game on an *authoritative* server that verifies updates from clients. In this work the concept of an authoritative server is extended to a distributed authoritative server. Then, depending on the semantics of the game a particular authoritative server will verify propose *gamestate* update.

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 3D world called the *gamestate*. We assume no limits on bandwidth and have strong constraints on latency which significantly impacts online game experience [Claypool and Claypool, 2006].

All online multiplayer systems are best-effort. If we can create a system with the same responsiveness with fault tolerance then it is a success.

2 Overview of Technology

In this section an overview of online multiplayer systems is given. For a recent survey of multiplayer online game frameworks we refer the reader to [Yahyavi and Kemme, 2013]. We also refer the reader to [Alexander and others, 2003] for information on MMO game design.

2.1 Multiplayer Network Design

To build an online real-time multiplayer game, there are two popular methods. One, is *peer-to-peer lockstep*, which is based on the *peer-to-peer* architecture. A diagram of the *peer-to-peer* communication structure is shown in Figure 1(a). In this approach, all nodes start in the same initial state and each node broadcast every move to the all other nodes. With nodes communicating directly the *gamestate* can not advance until each node's move is received by every other node. The overall latency of the system is then dependent on the slowest node in the system. This system is also not tolerant to faulty nodes, as each node will wait and decide themselves if a node has failed. Since the nodes use broadcasts to communicate, the method generates a significant volume of messages.

To achieve more real-time simulation, systems switched to a *client-server* model (see Figure 1(b)) [Coleman and Cotton, 1995]. With this model the *gamestate* is stored on a server and clients send updates to the server. This model reduces latency, for each client the latency is determined by the connection between that client and the server. However, this model is still too slow for real-time online multiplayer 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 *gamestate*. 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.

The system designed in this work uses parts from both the *peer-to-peer* model and the *client-server* model. The features of each method that we want to incorporate are:

1. The *client-server* model tends to have less latency
2. The *client-server* model supports clients joining mid game
3. The *client-server* model is less susceptible to cheating/malicious clients

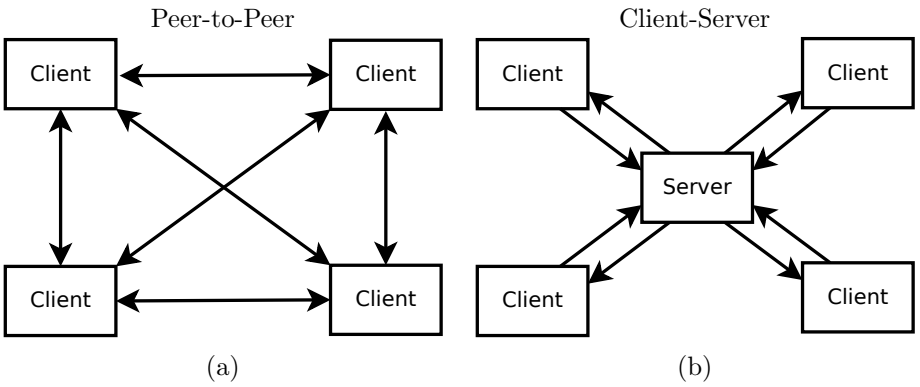

Figure 1: Two mutiplayer game networking models. The model on the left (a) is a *peer-to-peer* model where every client sends updates directly to ever other client in the game. The second model (b) is a *client-server* model. In this model all of the clients send updates to the server and the server send updates out to the clients.

4. The *peer-to-peer* model is more fault tolerant
5. the *peer-to-peer* model has distributed state

As in many multiplayer game networking systems, asynchronous communication is used. This is necessary to preserve the real-time nature of a game. Packet loss is considered not significant as a new packet with more up-to-date information will be sent soon after the lost packet.

3 Methodology

This section outlines our system design to overcome the facilitate distributed state synchronization. We propose a distributed server model (see Figure 2(b)) to enable graceful failure handling in our system. In this model, clients are paired with servers that run on the same machine. This arrangement results in a *fate sharing* between client server pairs. Paired servers validate actions performed by corresponding clients. If the server finds the action valid, the action of broadcast by the server to other nodes. The following subsections will discuss each component in the proposed architecture in detail.

3.1 The Client

In general, the client provides an interface to the user/player. Through this interface a player can give commands to their character (referred to as agent) in the game. The client simulates a local version of the game, where the player's commands are executed.

Each client in the system is paired with a server (called a *localServer*) which acts as a proxy to all distributed servers in the system. The state of a client's agent is forwarded to the *localServer* periodically.

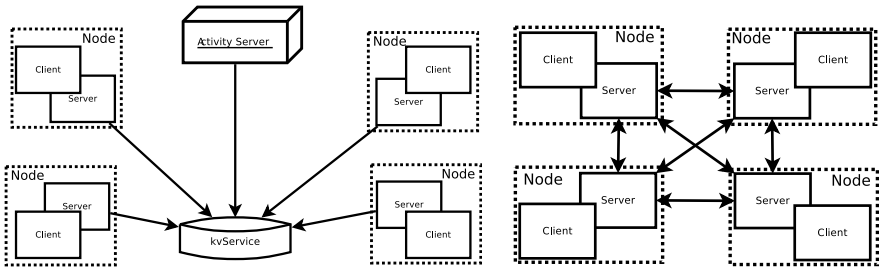

Figure 2: Figure (a) shows servers interacting with *activityserver* and *kvservice*. In this model all of the servers periodically update their entry in the *kvservice* and the *activityserver* periodically publishes a list of online servers in the *kvservice*. The model in figure (b) shows node-node communication in general.

3.2 The (Local) Server

The *localServer* also simulates its own independent version of the game. The *gamestate* at the *localServer* is guided in coordination with the other nodes in the system. The purpose of the *localServer* is not just to reduce message passing but also to act as a validator/authority over the client's *gamestate* and actions, to prevent malicious clients from propagating invalid information. The authoritative server for a particular client depends on the event/action being processed. Any client update is first forwarded to its *localServer* where it is validated against the current *gamestate* of the *localServer*. The update is published only if the requested action or update is found valid, else the action is rejected by the *localServer* and the client's state may be overridden by the *localServer*.

3.3 The Activity Server

The purpose of *activityserver* is to support the architecture of the system, which includes allowing a new nodes to join the system and publishing the list of nodes which are currently active. The *activityserver* doesn't perform game simulation, nor involves itself in game semantic actions.

3.4 The Key-Value Service

The *kvservice* is a hash table which acts as a communication channel between different system components. The sole purpose of this service is to provide the *activityserver* with a means to publish a list of active nodes. The nodes periodically update an entry in the *kvservice* and based on those entry updates, the *activityserver* publishes a list of nodes which are currently active. This list is periodically fetched by the nodes to determine which nodes are currently active.

3.5 ARM Game

This system support an Asynchronous Real-time Multiplayer Game (ARM Game). It consists of a few standard actions for players (characterized as agents in the

game). The first action is a *Move* which is represented as $updateLocation(a, p)$, where the first argument represents the agent intended to be moved and the second argument is the agent's new location. The other action is *Fire* which is represented as $fire(p, d)$, where the first argument represents current position of the agent firing the shot and the second argument represents the direction in which the shot is fired.

In order to simulate the game, information is needed on the other agents in the game. The information corresponding to each agent is called the *agentstate*. The attributes of *agentstate* are shown in Table 1. A collective database storing *agentstate* of all agents in the game is known as *gamestate*.

3.5.1 Protocol and Messaging

All communication is asynchronous without acknowledgements. We use messages formatted in JSON to facilitate information exchange in the system.

4 Prototype

In this section, we describe the construction and design of the prototype system we developed. The system is written in Go-lang. For the purpose of testing, the client is designed to simulate an agent randomly moving and firing in a random direction periodically. We use a centralized *kvservice* in our system, which could be enhanced to distribute over the *localServer*.

4.1 Data Structures

The system uses a *hashmap* to store the *gamestate* where agent names are keys. The data for each agent is stored as value in the *gamestate* in the form of a structure as shown Table 1.

Attribute name	**Type**	**Description**
Name	string	identifier for an agent
Location	Vector	current location of the agent
TimeStamp	int64	vector clock for the agent
LastUpdateTime	int64	the last time the node has received a message from the agent
Direction	Vector	the velocity of the agent

Table 1: Outline of the data stored by each agent (*agentstate*).

4.2 System Activation

To start the system, the *kvservice* and *activityserver* are executed. The *activityserver* initializes the keys in the *kvservice* that are used by the nodes to registered and active in the system.

4.3 Registration and Game State Construction

For a new node to join the game, the node has to register itself in the *kvservice*. The next step after registration is the *gameStateConstruction* for the new node. For this to take place, the new joining node waits for a short period of time to receive location broadcasts from all, already present, nodes in the system. The location broadcasts contain the latest location of each agent in the game. Once the *gamestate* is constructed, the newly joined node can begin simulating the game.

4.4 Move Event Processing

For our prototype, each client is responsible for a single agent. After every *gamestate* update by the agent, the client sends a position update $updateLocation(a, p)$ to the *localServer*. The *localServer*, based on the position of agent in its *gamestate*, verifies the validity of request. Any request that does not satisfy the following relation, will result in the request being ignored and the location agent overridden by the *localServer*.

if ((distance/deltaTime) > GameMaxVelocity)

where

distance := (new position of agent) − (previous position of agent),

clientDeltaTime := timeNow − LastUpdateTime,

deltaTime := float64(clientDeltaTime)/1000000000.0,

GameMaxVelocity := predefined maximum velocity for an agent

If the request is valid, the *localServer* broadcasts the received request to all other nodes in the system which update their own *gamestate* as well as forward the update to their paired clients. The clients will update their *gamestate* upon receipt of the message.

4.5 Fire Event Processing

The execution of a $fire(p, d)$ command is very similar to the move action execution. The client sends a fire request to its *localServer*. The *localServer*, based on its own *gamestate*, calculates if the shot hits any of the agents in the game. If it doesn't, the request is ignored. If the *localServer* does find a possible shot, the *localServer* forwards the request to the particular node paired with the client controlling that agent. Since that node is expected to have the most updated location of the agent being hit it is reasonable to have that node validate the event. The recipient node using its own *gamestate* verifies the shot. If the *localServer* finds the shot to be unsuccessful, the request is simply ignored. Whereas if the hit is successful, the recipient node broadcasts a $destroy(a_j)$ message to all the *localServers* in the system and respawns the agent at a different location.

Processing fire events on a different node is also advantageous in protecting against malicious clients. The event is first checked on the *localServer* to verify that the location of the agent and target agent are correct. This potential shot is then sent to the node with the most up to date information on the target agent and verified again. The client for the target agent could try to avoid being shot by blocking incoming fire events to its *localServer*. However, this would cause

the game state the client and *localServer* which will result in fire events from the client being invalidated by other nodes. We use this incentive-based method to discourage cheating with respect to fire events.

5 Evaluation

The system design is evaluated two ways. Primarily, the methods ability to handle failure. To cope with the failure gracefully and cause the least amount of suspension in the gameplay as possible. A secondary goal is to maintain consistency of the *gamestate* as more nodes are added to the system.

5.1 Fault Tolerance

The goal of the system was to gracefully cope with failing nodes without introducing latency. A game system design may choose to pause the processing of the game until the *gamestate* becomes consistent again, during this time the players must wait for the game state to synchronize. Game flow and minimal latency (as seen from a player of the game) is prioritized for consistency in this project. In order to reduce latency consistency is sacrificed. When a node fails it could take a short amount of time to detect this failure as an inactivity. During this time the agent for that node still exists in the game but is immobile.

Overall the system works as desired for graceful fault tolerance. When a node fails the agent exists in the *gamestate* for every node for a short period (a few seconds) after which all data related to the agent is removed from the *gamestate*. A graph of the latency is not shown as there is no latency induced by a failed node. In fact if a node fails latency should decrease thanks to the number of messages in the system will decrees.

5.2 Scalability vs Consistency

It is difficult to measure the consistency between each of the nodes. We would have to employ a large logging system that would take snap shots of the *gamestate* at points in time. In order for these snapshots to be effective there would need to be additional time synchronization across the nodes for comparison. Instead the relative number of successful shots is used to measure consistency. A successful shot is one that is initiated by a client, forwarded to the client's local server and again forwarded to the proper server for validation. The shot is successful if the last server agrees with the result of the shot. The occurrence of successful shot implies that 3 *gamestate*s where at least partial consistent. We measure the scalability by increasing the number of nodes in the system and comparing the relative number of fire events that are successful. Figure 3(top) shows us how well the system's consistency copes with the number of nodes in the system. As expected, as the number of nodes increases the number of successful shots decreases. The decrease in successful shots is a proxy for the system consistency. This is the result of increased latency and dropped packets in the system. The total number of possible shots[1] is shown in Figure 3(bottom), exhibiting a linear trend between the number of nodes and the total number of possible shots. The linear increase in shots tells us that the consistency analysis

[1] A possible shot is a shot one node believes to be correct.

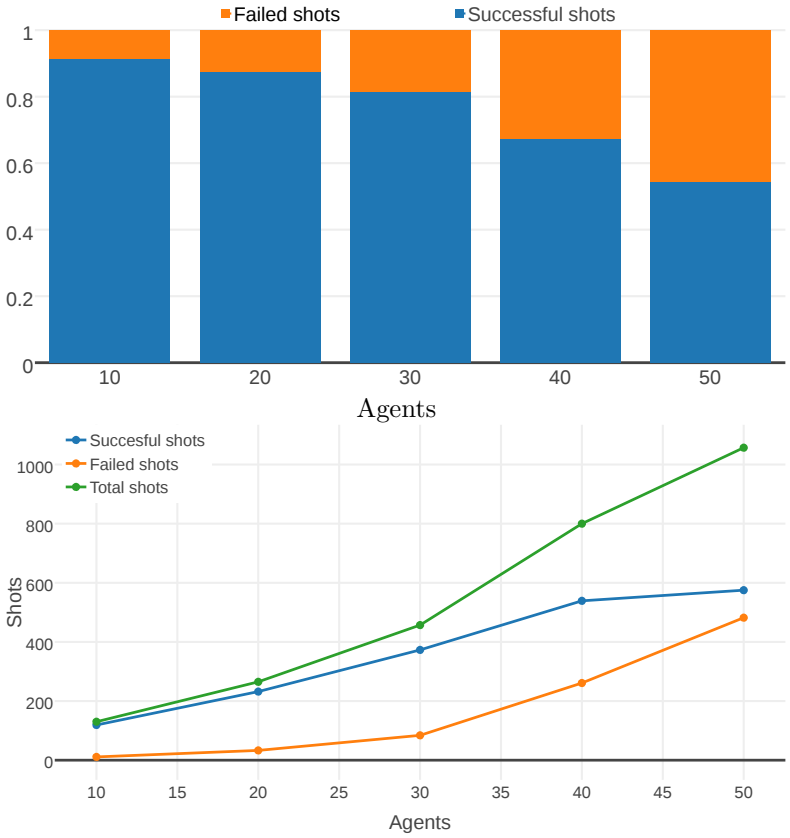

Figure 3: On the top a chart of the relative numbers of successful shots vs failed shots. As the number of nodes in the system increases the relative number of successful shots decreases. On the bottom a line plot of the number of possible shots fired. The number of possible shots increases linearly with the number of nodes in the system.

was not dependant on a non-linear growth in the number of possible shots due to packing more agents in the same size space.

5.3 Game Simulation

Here a description of the game is given, along with some features of the client Artificial Intelligence. The game is confined to a 3D box of dimensions $10x10x10$. Inside the box all the agents are simulated using a pseudo randomized AI. A visualization of the *gamestate* is shown in Figure 4(a and b).

RandomAI: When an agent is initialized the agent is given a random starting position in the game. The agent is also given a random direction that the agent will travel. From this the agent will navigate along that direction of travel and bounce off the walls of the game box.

Each client will send an update of its location every $100ms$ and will update its own internal state every $100ms$ as well. The client can updated its state at faster intervals if desired. Each node will update its status in the *kvservice*

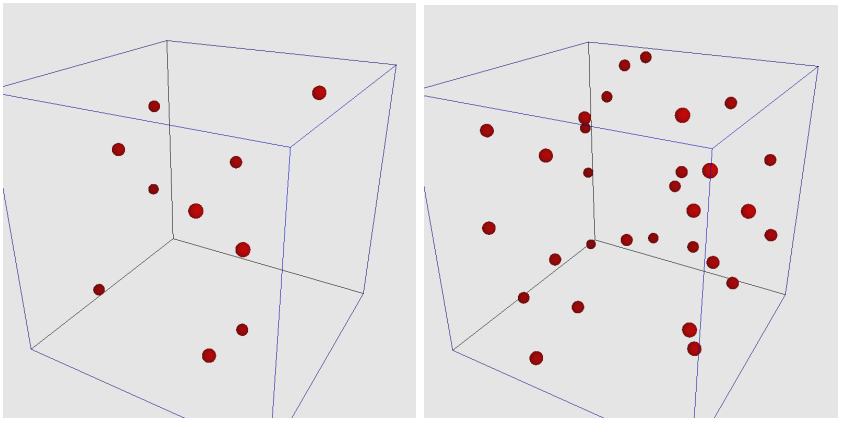

Figure 4: Two rasterized simulation frames. On the left with 10 agents and on the right with 30 agents.

every 2 seconds. The *activityserver* will collect the node updates and post the updated list of active nodes every second. If a node dose not post an update for a duration of 6 seconds, it is removed from the list of active nodes. In Table 2 the data a node stores to support the system is described. The *gamestate* takes up the most amount of memory and it the most frequently updated structure.

Attribute name	**Type**	**Description**
gamestate	map<string, Agent>	map of agent identifiers to agent objects in the game
Nodes	map<string, *net.UDPConn>	map of nodes to the UDP connections to send messages to the nodes
MyClientName	string	identifier of the client this node is paired with
ClientLink	*net.UDPAddr	UDP address of the client for this node
Connection	*net.UDPConn	This nodes UDP listening connection

Table 2: Layout of the data stored by each node.

6 Conclusion

We have presented a prototype design for a distributed *client-server* system that has a communication structure similar to a *peer-to-peer* model. The system supports a real-time online multiplayer game with distributed state. The design gracefully handles node failure without inducing any game play suspension. The system demonstrates promise in its ability to scale in the number of nodes with respect to the consistency of the system.

Limitations The system can suffer from DOS attacks by nodes that have a desire to reject fire messages. There is also a significant amount of messaging in the system due to the *peer-to-peer* structure between nodes.

Future Work The scalability of the system could be further improved by using a spatial partitioning scheme that would limit the number of relevant nodes each node should keep track of. There are additional methods that could be used to protect against malicious clients. One example is to encrypt the messages sent to nodes. This way a malicious client won't know which messages are being ignored, in-turn that client's own information will be out of date and rejected by other nodes. Another option is to send fire messages to all, or a random subset, of nodes and have the majority determine the outcome of the action.

References

[Alexander and others, 2003] Thor Alexander et al. *Massively multiplayer game development*. Charles River Media, 2003.

[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.

[Yahyavi and Kemme, 2013] Amir Yahyavi and Bettina Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, 46(1):9:1–9:51, July 2013.