Dynamic Server Selection for Large Scale Interactive Online Games (Work-in-Progress Paper)

Bong-Jun Ko Dan Rubenstein
Columbia University

Kang-Won Lee Seraphin Calo
IBM T.J. Watson Research

1 Introduction

Large scale interactive online games, such as Massively Multiplayer Online Games(MMOG), aim to support a very large number of clients. In many game scenarios, MMOG providers are required to support tens of thousands of geographically distributed users simultaneously. Thus they have focused on developing a highly scalable game server architecture and supporting network infrastructure that spans wide geographical regions while satisfying loose real time requirements [3, 9]. Recently, however, MMOGs are beginning to incorporate more "action scenes" to attract users [8], and thus it becomes increasingly important to support small response time and interaction delay between users.

There are several challenges in augmenting MMOGs with such interactive features. First, unlike in online First-Person-Shooting (FPS)-type games where a small number of nearby users are assigned to a same game session and interact with each other, MMOG games must maintain a persistent virtual world view for a large number of game players that are distributed over the network. Second, the maintenance of a long-lived persistent world mandates a server-based game architecture, where clients interact with a central server that keeps track of the game states. In this server-client architecture, data packets typical travel over a longer path than in the peer-to-peer architecture, incurring additional delay. To overcome this limitation, [4, 1, 7] have proposed a mirrored server architecture, where a set of distributed game servers are orchestrated to support a large number of geographically distributed clients. In this architecture, the game servers are typically interconnected via well provisioned network links, and each game client is directed to connect to the closest game server. However, related work in this area has not addressed how a game session must be allocated to each server and how each client should connect to game servers to minimize the resource consumption while satisfying the delay requirements.

In this paper, we explore the problem of server allocation in a distributed online game environment and its impact on the interaction delay experienced by the participants. To this end, we first present a novel synchronization delay model for a given server and client configuration. We then present a sketch of a distributed algorithm for server selection that optimizes server resource consumption while satisfying the real-time delay requirement of the distributed game. Due to the non-linear characteristics of the synchronization delay model, we identify that finding an optimal solution to the proposed server selection problem is much harder than related problems such as bounded delay multicast tree construction problem or facility location problem. This is a work-in-progress report. Currently we are evaluating the effectiveness of the proposed algorithm using an extensive simulation.

2 Modeling the Synchronization Delay

In this section, we present a synchronization delay model for a distributed interactive online game session given a server and client configuration.

2.1 Synchronization delay

In the online game context, a *server* refers to an entity that calculates and simulates the game states based on the players' actions, and a *client* refers to an entity that renders and presents the game states to the user.

When a server runs the game simulation or when a client presents the game state, they synchronize the events or states based on the time that they have been generated. In many practical game systems, however, events are typically divide into discrete time slots. Thus during the operation, the events or states generated within the same slot are considered to have been generated simultaneously. We assume this relaxed concurrency model in our paper.

We define the *synchronization delay* in online games as the time difference between the instance that all participating clients send the players' actions and the instance that the new state corresponding to the actions is presented to the players. This synchronization delay depends on two types of latency between the servers and the clients, namely the *upstream latency* and the *downstream latency*. The former refers to the time taken to deliver the player's action from the client to the server, and the latter refers to the latency to deliver the updated game state from the server to the client.

To synchronize the game play and interaction amongst the users, the online game system must taken into account these latencies between the servers and the clients. More specifically, in order for the server to synchronize the players' actions based on the actual instances that they were generated, the game server calculates the new game state *after* the action data from the furthest client has arrived. Otherwise, the action from the furthest client will not be synchronized with the other clients'. On the other hand, at the client side, the game state should not be presented to the players until it is delivered to the furthest client from the server. Otherwise, some clients further from the server than the others would always be presented with late game state [2]. To address these issues, several techniques have been proposed including bucket synchronization [5], Sync-MS [6].

Note that, the overall synchronization delay that the players will experience is determined by the location of the game servers and their distances from the clients, and this delay will be determined by the *largest* upstream and downstream latencies.

2.2 Delay model and problem formulation

Now we present the generalized model of the synchronization delay, which will be used in this paper to develop our server selection algorithm. For simplicity, we assume a mirrored-server architecture, where multiple game servers are interconnected via well provisioned network links and run the identical copies of game simulation. We later show this model is applicable to the other game architectures, namely peer-to-peer and server-client architectures.

In the mirrored-server architecture, each client is assigned to one of the servers, called the contact server, which is responsible for forwarding the client's action data to all the other servers participating in the same game session. This effectively defines a one-to-one mapping from a client to a server. We call this mapping an *allocation*. Upon receiving all clients' actions that belong to a same time slot, each game server independently calculates a new game state and sends the updated state to the directly connected clients.

Let C denote a set of game clients that participate in a game session and let S denote a set of available game servers. In our model, S and C are not necessarily disjoint, i.e., some clients may act as servers, and vice versa, depending on the configuration. Servers in S form an undirected connected graph, in which $d_s(j,k)$ denotes the shortest distance between server s_j and s_k, and let $d_c(i,j)$ denote the distance between a client c_i and a server s_j.[1] Suppose client c_i is mapped to server s_j in some allocation A. Then we say server s_j *serves* client c_i under A. We define $C_j \subset C$ as the set of clients that are directly connected to the server s_j, i.e., $C_j = \{c_i \in C | s_j \text{ serves } c_i\}$. We also define the session server set, $S(A)$ under the allocation A as the set of servers that serve at least one client, i.e., $S(A) = \{s_j \in S | C_j \neq \phi\}$.

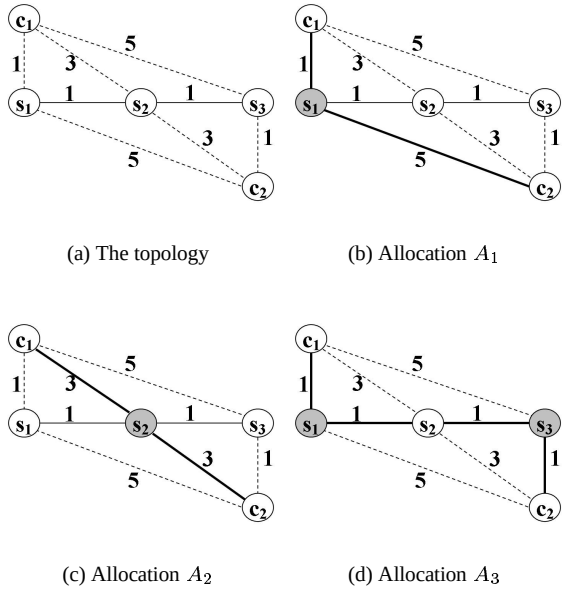
Let $D_u(i,k)$ be the *upstream* distance between a client c_i and a server s_k. We note that $D_u(i,k)$ is defined not only for a client and a server that are directly connected, but also for a client and a server connected indirectly via some other servers forwarding the client c_i's actions to server s_k through the shortest path in the server graph, i.e., $D_u(i,k) = d_c(i,j) + d_s(j,k)$, where s_j is a contact server for c_i. On the other hand, the *downstream* distance from s_j to a client $c_i \in C_j$ is just $d_c(i,j)$ as c_i receives the game states from its contact server s_j.

In order to compute the overall synchronization delay, it is easier to look at the latencies from the perspective of each server since the servers are responsible for fairly simulating the game and updating the game states. The largest delay for the clients' actions to reach a server s_j is $\max_{i \in C} D_u(i,j)$. Note that the action delivery delay from *all* clients should be taken into account because each server has to eventually receive the action from all clients in the session to update the game states. On the other hand, the largest delay for an updated state from a server s_j to arrive at all relevant clients is $\max_{i \in C_j} d_c(i,j)$. Now, since *all* clients should be provided with a consistent view, the overall synchronization delay $D_s(A)$ under allocation A can be written as:

$$D_s(A) = \max_{j \in S(A)} \{\max_{i \in C} D_u(i,j) + \max_{i \in C_j} d_c(i,j)\} \quad (1)$$

As mentioned above, this model can be easily applied to the other game architectures. For instance, setting $S = C$ yields the model for the peer-to-peer architecture, in which case the overall delay becomes just the maximum client-to-client delay because the downstream delay is effectively zero. Adapting this model to a single centralized server-client architecture is straightforward by assigning only one element in S.

[1] In this paper, we use "distance" and "delay" interchangeably.



(a) The topology (b) Allocation A_1

(c) Allocation A_2 (d) Allocation A_3

Figure 1: Examples of game server allocation

Our goal in this paper is to minimize the number of servers being used by a session, while satisfying a given synchronization delay requirement. Formally stated, we want to find a server allocation A_{min} that minimizes $|S(A)|$ subject to $D_s(A) \leq \Delta$ and $|S(A)| \geq 1$. The non-linearity of the synchronization delay model makes this problem challenging. Because it does not have the optimal substructure, a greedy algorithm won't be able to provide an optimal solution. And the problem seems to demand exhaustive search of all possible combinations of server-client mappings to determine the optimal solution. Currently, we are investigating to analyze the complexity of the problem.

3 Server Selection Algorithm

In this section, we investigate the general properties of the synchronization delay model presented in the previous section through examples. We then present a sketch of the proposed server selection algorithm that aims to minimize the number of servers allocated for a session while satisfying the delay bound.

Figure 1 illustrates a simple example in a system with 3 servers, s_1, s_2, and s_3, and 2 clients, c_1 and c_2. Figure 1(a) shows a network configuration, where the solid lines represent the latency between the servers in server network, and the dashed lines represent the latency between each pair of server-client. Since our synchronization delay model is a nonlinear function, in which each client-server path contributes to the overall delay, it is important to identify which path becomes the "bottleneck" that effectively determines the overall delay.

Figures 1(b)-1(d) show three different server allocations, A_1, A_2, and A_3, in each of which the allocated session servers are colored in grey and each client is served by the closest session server. According to the delay model, the overall synchronization delay in Figure 1(b) is 10 with the maximum upstream and

downstream delay each being 5 (between c_2 and s_1). In this case the Internet path between c_2 and s_1 is the bottleneck of the overall delay. Allocating s_2 instead of s_1 as the session server (Figure 1(c)) results in a reduced synchronization delay of 6 (upstream/downstream 3 each). Note that, though the end-to-end distance between the clients are the same in A_1 and A_2, the synchronization delay is reduced by placing the server at the center of the network. Finally, if we allocate two servers at the edges of the server network as in Figure 1(d), the sync-delay decreases to 4 with the upstream latency 3 (e.g., along the path c_1-s_1-s_2-s_3) and the downstream latency reduced to 1. In this case, the reduction comes from placing the servers near the clients at the expense of using two servers.

From the simple example presented above, we draw a following intuition to design our server selection algorithm. Suppose we want to choose only one server that minimizes the overall synchronization delay. Then the optimal placement would be selecting the server that minimizes the maximum distance (not the average) to all clients (e.g., in the above example, s_2). We call such server a *core server*. Choosing any other server as the single point of synchronization would clearly increase the delay.

If allocating the core server to the session satisfies the delay requirement, then this is clearly an optimal solution since we use only one server to support the session. However, if the core server cannot satisfy the delay requirement, we can try to allocate more servers that are closer to the clients to reduce the synchronization latency. By allocating the servers close to the clients, the chance of reducing the overall synchronization latency increase (as in allocation A_3 in the example) because the inter-server connections are typically well provisioned. When trying out alternative servers to participate in the session, assigning any client to a server further than the core server would only increase the overall delay. Thus one only needs to try to assign a server that is closer than the core server.

To summarize, we observe that:

- If any client is assigned to a server further than the core server from that client, it would only increase the overall synchronization delay and the distance between that client-server pair becomes the bottleneck.
- If it is faster to forward packets via the contact server over the server networks than to send packets to a remote server directly, allocating servers near the "edge" of the server network (and close to the clients) reduces the overall synchronization delay. However, this comes at the expense of increased number of servers in the session.

Based on the above observations, we propose a dynamic server selection algorithm in which client can try to contact a different server to reduce the number of servers without violating the real-time delay bound. The proposed mechanism proceeds in the reverse direction of the above mentioned search process, i.e., we look for an allocation with a smaller number of servers starting from the edge of the network toward the core.

Due to space constraints, we only present a sketch of the server selection below. Suppose a set of clients, C, and a set of session servers, $S(A)$, under some initial allocation A satisfy the delay requirement Δ, i.e., $D_s(A) \leq \Delta$.

- **STEP 1:** Find the *core server*, s^*, of the session that minimizes the maximum distance to the clients.
- **STEP 2:** Construct a shortest-path tree spanning all the servers in $S(A)$ with s^* being the source of the tree. This tree may include a non-session server but all leaf nodes are the session servers.
- **STEP 3:** For each client, do the followings. Probe the parent server of the current contact server along the tree constructed in STEP 2 to see whether the overall sync-delay would be still within the bound if the client would have migrated to the parent server. If yes, connect that client to the parent server. Also if there is any change in the set $S(A)$ due to the migration, then repeat from STEP 2.
- **STEP 4:** If no client can climb up along the tree in STEP 3, then terminate the process.

Note that constructing the tree in STEP 2 is to "guide" the clients in the direction toward the core, and it can be done in an efficient manner if the servers already maintain the shortest path routing tables necessary to forward the packets from the client to the other servers. However, it is important to note that this tree is used only for the search process, not for forwarding the clients' packets. Data packets are routed along the shortest path between servers. One other issue is how to find the core server in STEP 1 in a scalable manner, and we leave it as the future work.

Currently we are evaluating the effectiveness of the proposed algorithm in comparison with other server allocation algorithms including greedy, random, K-closest, etc. Preliminary simulation results show that this process allocates a significantly less number of servers than simply connecting the clients to the closest servers. Consequently it consumes a much less server resources and involved network bandwidth.

References

[1] Daniel Bauer, Sean Rooney, and Paolo Scotton. Network infrastructure for massively distributed games. In *NetGames'02*, April 2002.

[2] Paul Bettner and Mark Terrano. 1500 archers on a 28.8: Network programming in age of empires and beyond. In *Game Developers Conference 2001*.

[3] Butterfly.net. `http://www.butterfly.net/`.

[4] Eric Cronin, Burton Filstrup, and Anthony Kurc. A distributed multiplayer game server system, technical report, univ. of michigan. Technical report, May 2001.

[5] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet, IEEE networks magazine, vol. 13, no. 4, July/August 1999.

[6] Yow-Jian Lin and Sanjoy Paul Katherine Guo. Sync-MS: Synchronized messaging service for real-time multi-player distributed games. In *Proceedings of 10th IEEE International Conference on Network Protocols (ICNP 2002)*, November 2002.

[7] Martin Mauve, Stefan Fischer, and Jorg Widmer. A generic proxy system for networked computer games. In *NetGames'02*, April 2002.

[8] PlanetSide. `http://www.planetside.com/`.

[9] Terazona. `http://www.zona.net/`.