Adaptive Server Selection for Large Scale Interactive Online Games

Kang-Won Lee
IBM T.J. Watson Research
Hawthorne, NY
kangwon@us.ibm.com

Bong-Jun Ko
Columbia University
New York, NY
kobj@ee.columbia.edu

Seraphin Calo
IBM T.J. Watson Research
Hawthorne, NY
scalo@us.ibm.com

ABSTRACT

In this paper, we present a novel distributed algorithm that dynamically selects game servers for a group of game clients participating in large scale interactive online games. The goal of server selection is to minimize server resource usage while satisfying the real-time delay constraint. We develop a synchronization delay model for interactive games and formulate the server selection problem, and prove that the considered problem is NP-hard. The proposed algorithm, called *zoom-in-zoom-out*, is adaptive to session dynamics (e.g. clients join and leave) and lets the clients select appropriate servers in a distributed manner such that the number of servers used by the game session is minimized. Using simulation, we present the performance of the proposed algorithm and show that it is simple yet effective in achieving its design goal. In particular, we show that the performance of our algorithm is comparable to that of a greedy selection algorithm, which requires global information and excessive computation.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications; C.2.4 [**Distributed Systems**]: Client/server

General Terms

Algorithms, Performance, Design

Keywords

MMOG, Server selection, Distributed algorithm, Synchronization delay model

1. INTRODUCTION

Large scale interactive online games, such as Massively Multi-player Online Games (MMOG), aim to support a very large number of clients. In practice, MMOG providers often are required to support tens of thousands of geographically distributed users simultaneously. In this respect, MMOG providers have so far focused on developing a highly scalable game server architecture and supporting network infrastructure that spans wide geographical regions while satisfying loose real time requirements [5, 16]. Recently, however, MMOGs are beginning to incorporate more interactive features and action sequences to attract users [12]; thus it becomes increasingly important to provision enough server resources to support real-time interaction 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 the same game session, 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. However, this conventional server-client architecture does not scale well as the number of clients increases. To overcome this limitation, a mirrored server architecture has been proposed [7, 2, 10], where a set of distributed game servers are orchestrated to support a large number of 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. In parallel with this architectural development, proposals have been made to dynamically provision game servers on the fly exploiting emerging Grid technologies [14, 5]. According to these *on-demand* game architectures, game servers can be provisioned to accommodate the capacity requirements as the user demand changes. Thus it now becomes an important issue to dynamically provision and utilize server resources in an efficient manner.

In this paper, we present a novel distributed algorithm that selects game servers for a group of game clients participating in large scale interactive online games. The goal of the server selection algorithm is to select the minimum number of servers while satisfying the real-time delay constraint of the game session. To this end, we develop a synchronization delay model for interactive games, formulate the server selection problem, and prove that finding an optimal solution to the considered problem is NP-hard. The proposed algorithm, called *zoom-in-zoom-out*, is adaptive to session dynamics (e.g. clients join and leave) and lets the clients select appropriate servers in a distributed manner such that the number of servers used by the game session is minimized. Using simulation, we present the performance of the pro-

NOSSDAV'04, June 16–18, 2004, Cork, Ireland.

posed algorithm with various zoom-in-zoom-out techniques and show that it is simple yet effective in achieving its design goal. In particular, we show that the performance of our algorithm is comparable to that of a greedy selection algorithm, which requires global information and excessive computation.

The remainder of this paper is organized as follows. Section 2 presents synchronization delay model and problem formulation. Section 3 presents the proposed server selection algorithm. Section 4 presents the performance of the proposed algorithm in comparison with more expensive greedy algorithm. Section 5 presents an overview of related work, and finally Section 6 concludes the paper.

2. PROBLEM FORMULATION

2.1 System model

In our model, 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 player. We assume that a virtual persistent world of the game is divided into multiple *regions*, where there are relatively few game players (compared to the number of users in the entire game world) in each region interact with one another in a direct manner. We call the persistent state of a region as the *sessions*.

For simplicity, we assume a mirrored-server architecture, where multiple game servers are interconnected via well provisioned links. In this 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 the same time slot, each game server independently calculates a new game state and sends the updated state to the directly connected clients.

For accurate game simulation and presentation, game events generated by the players must be ordered according to a global clock. In many practical game systems, however, time is divided into discrete slots and events that have been generated during the same time slot are considered to have happened simultaneously [8, 9]. We define the *synchronization delay* between a client and a server as the time difference between the instance that the client sends its players' actions and the instance that the client renders a new game state in response to the actions sent to the server. This synchronization delay depends on the network latency from the client to the server (*upstream latency*), processing time at the server, and the network latency from the server to the client (*downstream latency*).

To synchronize game play and interaction amongst all players participating in the same session, the game system must take into account synchronization delays between all clients in the session and the corresponding servers. More specifically, a game server must calculate a new game state *after* action data from the farthest client have arrived. Otherwise, the action from the farthest client will not be synchronized with others. Similarly, at the client side, a new game state should not be presented to the players until the same game state is delivered to the farthest client from the server. Otherwise, the game simulation becomes unfair to the farthest clients [3].

2.2 Delay model and problem statement

Let C denote a set of game clients that participate in the game session and let S denote a set of available game servers. Servers in S form an undirected connected graph, called the server network. Let $d_s(j,k)$ denote the shortest distance (or latency) between servers s_j and s_k, and let $d_c(i,j)$ denote the distance between a client c_i and a server s_j. 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 in C, 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 the 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 game states from its contact server s_j.

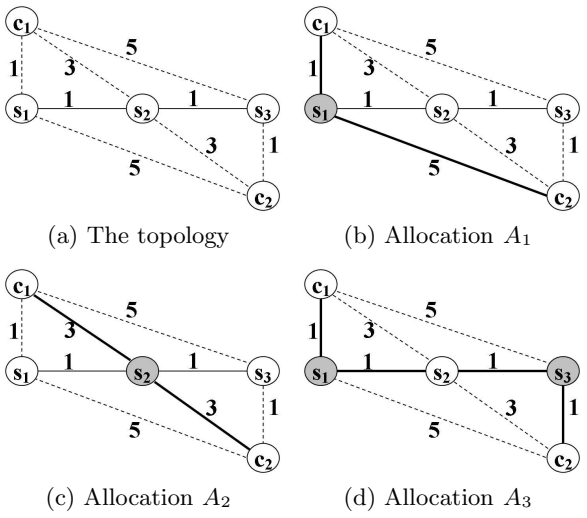
We now consider the overall synchronization delay for a session. As previously discussed, for a server to simulate game state in a fair manner it must wait until all the action data from all the clients in the session to arrive. In other words, a server s_j must wait for $\max_{i \in C} D_u(i,j)$ before game simulation. Then it processes the action data and sends out updates to all the clients it serves. This game state update takes up to $\max_{i \in C_j} d_c(i,j)$ for the farthest client. Since this condition must hold for *all* servers, the overall session synchronization delay $D_s(A)$ under allocation A can be written as:

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

Based on this session synchronization model, we now formulate our problem. Our goal in this paper is to minimize the number of servers allocated to a session, while satisfying a given synchronization delay requirement. Formally stated, given a network topology consisting of a set of servers S and a set of clients C, and a real-time delay requirement Δ, find a server allocation A_{min} that minimizes $|S(A)|$ subject to $D_s(A) \leq \Delta$ and $|S(A)| \geq 1$. This problem is NP-hard as it generalizes the set-covering problem [6].

THEOREM 1. *The considered server allocation problem is NP-hard.*

PROOF. Consider, in our model, the case when the distance between every pair of servers is zero. Then the optimization goal in this specific case would be to minimize the number of servers subject to the condition that every client is within distance $\Delta/2$ from the server, to which it is allocated, where Δ is the sync-delay bound in our model. This is exactly a set-covering problem, in which a set is defined by the set of clients that are within the distance $\Delta/2$ from each server. Since our problem generalizes the NP-hard set-covering problem, it is also NP-hard. □



(a) The topology (b) Allocation A_1

(c) Allocation A_2 (d) Allocation A_3

Figure 1: Examples of game server allocation

3. SERVER SELECTION ALGORITHM

3.1 Impact of Server Allocation

In this section, we first analyze the impact of server allocation using an example. Figure 1 illustrates a simple example 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 the server network, and the dashed lines represent the latency between each server-client pair.

Figures 1(b) – 1(d) show three different server allocations, A_1, A_2, and A_3, where allocated session servers are marked in gray. According to the session delay model, the overall delay for allocation A_1 is 10 with both maximum upstream and downstream delays being 5 (between c_2 and s_1). Allocating s_2 instead of s_1 as the session server (allocation A_2) reduces the synchronization delay to 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 allocation A_3, the synchronization 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 this example, we can draw the following intuition to design our server selection algorithm:

- If we had to choose only one server to minimize the overall synchronization delay, then it would be optimal to select a server that minimizes the maximum distance (not the average) to all clients. In graph theory, such a node is called the *center* of a network. In this paper, we call it a *core server*.

- If it is faster to forward packets via the contact server over the server network 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 increasing the number of servers in the session.

Based on these observations, we design a distributed server selection algorithm as follows.

3.2 Server Selection Algorithm

Given a set of clients C, a set of servers S, and delay requirement Δ, do the following:

- **STEP 1:** Initially allocate each client $c_i \in C$ to the closest server in S. Denote this initial contact server of c_i by $s_0(c_i)$. This produces an initial allocation A. We assume that the session has been provisioned to satisfy the delay requirement Δ, i.e., $D_s(A) \leq \Delta$. Otherwise, note the violation, and call for a high level session regrouping.

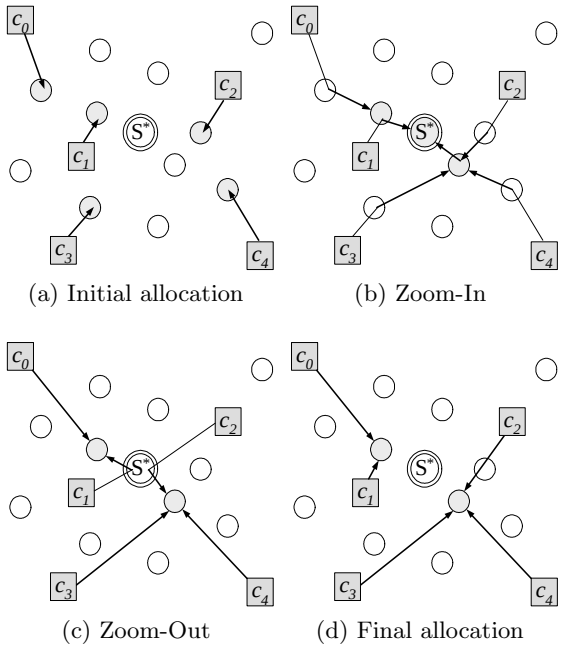
- **STEP 2:** Find the *core server*, $s^* \in S$, of the session that minimizes the maximum distance to the clients. Finding the core of a network in a distributed system has many interesting applications (e.g. core-based multicast tree construction), and thus has been studied in various contexts. In this paper we employ a tournament-based method studied in [17].

- **STEP 3 *(Zoom-In)*:** For each client c_i, do the following. Let $s_k(c_i)$ be the current contact server of c_i. Also, among the servers that are further from c_i than $s_k(c_i)$ but are closer to s^*, find the closest server to c_i, and denote it by $s_{k+1}(c_i)$. Then probe $s_{k+1}(c_i)$ to see whether the session synchronization delay would still be within the bound Δ if the client migrated to $s_{k+1}(c_i)$. If yes, $s_{k+1}(c_i)$ is the new contact server for c_i. Repeat this step for all clients until no client can migrate to a server that is closer to s^*.

- **STEP 4 *(Zoom-Out)*:** For each client c_i, do the following. Let $s_k(c_i)$ be the current contact server of c_i, and S' be the current set of contact servers for all clients. Then among the servers in S' that are closer to c_i than $s_k(c_i)$, find the farthest server from c_i, and denote it by $s_{k+1}(c_i)$. Then probe $s_{k+1}(c_i)$ to see whether the session synchronization delay would still be within the bound Δ if the client migrated to $s_{k+1}(c_i)$. If yes, $s_{k+1}(c_i)$ is the new contact server for c_i. Repeat this step for all clients until no client can migrate to a server that is farther from s^*.

The procedure in STEP 3 has the effect of moving a cluster of the session servers toward the core server. With this zoom-in procedure, the overall synchronization delay tends to increase. When this step cannot proceed without increasing the synchronization delay beyond the real-time delay, this zoom-in process terminates. By performing the procedure in STEP 4, we seek to further reduce the number of session servers. This is possible because after the zoom-in procedure, servers near the core server are most likely to have been selected. However, some of them can be removed without affecting the overall synchronization delay.

The computation overhead incurred at each client is not too severe. The step 1 requires the communication overhead of $O(|S|)$ for each client for exact computation. In most practical situations, however, it will be $O(k)$ where $k \ll |S|$ is the number of servers in the client's region. The step 2



(a) Initial allocation (b) Zoom-In

(c) Zoom-Out (d) Final allocation

Figure 2: ZIZO algorithm

requires a total of $\log_2 |S|$ tournaments for the entire session. The step 3 requires either $O(1)$ or $O(k)$ depending on the type of the zoom-in algorithm (see Section 3.3). The step 4 requires $O(m)$ where $m \ll |S|$ is the number of session servers.

Figure 2 illustrates the above procedure through an example. In Figure 2(a), each of 5 clients, $c_0, \cdots, c_4$ is initially allocated to the closest server. In this allocation, we have 5 servers participating in a game session by the clients. Now, in Figure 2(b), each client incrementally probes servers that are closer to the core server, for example, client c_2 migrates twice toward the core server as long as the delay bound is satisfied. As a result of the zoom-in procedure, three servers near the core server are selected. Now, clients further seek to reduce the number of session servers by moving out from the core server. For example, in Figure2(c), two clients, c_1 and c_2, were able to migrate to the session servers closer than the core server, with s^* no longer being allocated to any client. The final server allocation is shown in Figure 2(d), with only two servers being selected by the procedure.

3.3 Implementing the Selection Algorithm

There are several issues when one tries to implement the above mentioned server selection algorithm. This subsection briefly discusses them.

The first issue concerns game session migration. Although a game architecture based on on-demand technologies enables our dynamic server selection, migrating a game session from one server to another is a relatively expensive procedure. As a result, probing a new server and trying to migrate the game session in each zoom-in and zoom-out step by a client is not a good idea. We address this issue by handling the probing and zoom-in zoom-out steps in the control and management plane, not in the actual game session plane. To support this operation, a server that has been probed by a client runs a simulation on synchronization delay computation and maintains a virtual state. The clients keep probing the next possible server following the server selection algorithm until they reach a steady state. Only when the entire zoom-in and zoom-out process terminates, do the clients coordinate and move the game session to the newly allocated servers. In this way, we can minimize the management and session migration cost.

Secondly, the performance of the "zoom-in" and "zoom-out" procedures in our proposed algorithm relies on the effectiveness of the server search algorithm. We consider two types of search methods. The first method is a full search of all servers in S. This may be costly if S is large, but will provide the most accurate result. The second method is an approximation based on a hypothesis that when a client selects the next server, it will be most likely on the shortest path from the client to the core server. For the second method, once we discover a core server s^*, we first construct a shortest-path tree spanning all the servers in $S(A)$ with s^* being the root. This tree may include a non-session server but all the leaf nodes are session servers. We migrate the clients along this tree when they zoom-in or zoom-out. More precisely, when zooming in, each client probes the parent of the current contact server in the tree. Similarly, when zooming out, a client probes the server that is closest to the client among the children of the current contact server.

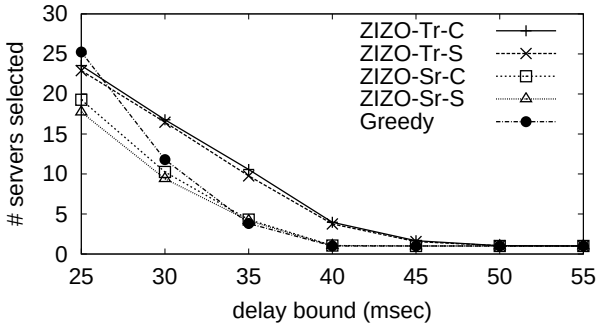
Finally, we consider two different types of client migration strategies for the zoom-in and zoom-out process. In a naive implementation, each client may individually probes candidate servers and make a decision on migration. In this case, it is possible that, while some clients connected to a server could change their contact server, others may fail to migrate because it would violate the overall delay constraints. Therefore this *uncoordinated* migration of clients may result in a suboptimal result by temporarily increasing the number of servers allocated to the session. An alternative approach we consider in this paper is to coordinate the migration of all clients on the same server simultaneously, and move them only when all the clients can migrate to new servers. In the next section, we present the effectiveness of the considered design alternatives using a simulation-based study.

4. PERFORMANCE EVALUATION

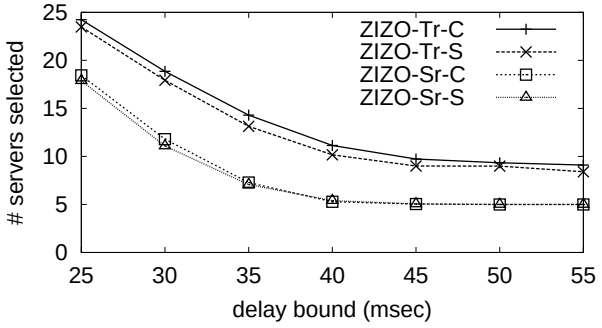
In this section, we evaluate the performance of the proposed algorithm through simulations. For simulation, we use a two-level, Internet-like topology generated by the BRITE topology generator [4]. Out of 5,000 nodes (50 AS $\times$ 100 nodes per AS) generated by BRITE, we randomly select a total of 100 servers and 50 clients to participate in the same game session. Note that we have selected a relatively small number of clients (with respect to that of the servers) in the simulation since our goal is to evaluate the performance of the algorithm for a "single" session. In the aggregate, there will be many such sessions and the total number of clients will be much larger than that of servers. We measure the performance results by repeating each simulation 100 times. To emulate the well-provisioned server network with little congestion, the inter-server latency is set to be smaller than the client-to-server latency. In particular, we show the case when the latency between two servers is reduced to 25% of the latency in underlying topology given by the topology generator, while our findings hold regardless of the particular reduction factor.

For evaluation, we first compare the performance of the proposed algorithm to a centralized algorithm that performs the server selection in a greedy manner. To briefly describe the idea, the greedy algorithm tries to add a new server to a session server set by exhaustively searching for a server that, when added, results in the minimum increase in the synchronization delay. When a new server is added, each client calculates the delay to the new server and connects to it if it gives lower synchronization delay. In this way, the synchronization delay is guaranteed to decrease monotonically as the number of servers increases. This greedy algorithm is similar to the one presented in [13] in the context of the Web server replica placement problem, which has been shown to perform closely to the optimal solution. Note that this greedy algorithm is not only impractical to be applied in a distributed networking environment, but also is computationally expensive because it must evaluate the new synchronization delay for each of the servers that have not yet been added to the session server set.

In the results shown in this section, we use some abbreviated indices to indicate different ways of migrating the clients as explained in Section 3.3: -*Tr*(migrate clients along a tree) and -*Sr*(migrate clients by "search") distinguish two different ways of selecting the next server to migrate to, and -*C*(migrate each client individually) and -*S*(migrate all clients in a coordinated manner) represent two different ways in moving clients from one server to another.



(a) server capacity unlimited



(b) server capacity limited

Figure 3: The number of servers allocated as a function of delay bound

Figure 3 depicts the number of servers allocated by the proposed algorithms and the greedy algorithm. We vary the sync delay bound Δ along the x-axis. Figure 3(a) shows the results when the servers' capacity is unbounded, i.e., each server can serve an arbitrary number of clients, and Figure 3(b) is when each server's capacity is bounded by 10 clients.[1] From Figure 3(a) we find that the performance of our proposed algorithm is comparable to that of the greedy algorithm. We also note that if the delay bound is sufficiently large, the optimal number of servers for the unbounded case is 1, and the optimal number of servers for the bounded case is 5. We observe that both ZIZO (zoom-in-zoom-out)-Sr-C and ZIZO-Sr-S find this solution as the delay bound increases. On the contrary, ZIZO-Tr-C and ZIZO-Tr-S are slow in convergence in the unbounded case, and fail to reach the optimal point in the bounded case. From this result, we find that the performance of the proposed algorithm is sensitive to the server search mechanism. More specifically, we conclude that the full search algorithm provides much better performance than the tree-based search does. On the other hand, we observe that the coordination in client migration offers little benefit. This observation implies that each client can independently make a greedy decision regarding its server selection, and they can still achieve as good allocation of the server resources as the case with explicit client coordination.

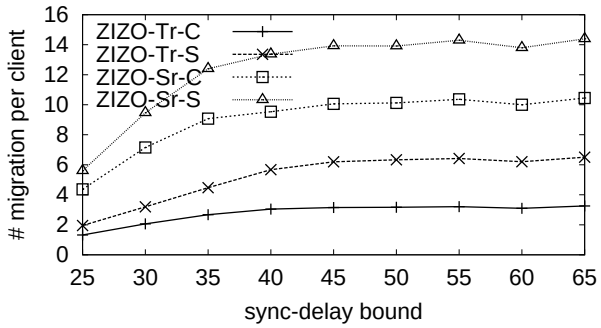


Figure 4: The number of server migrations per client

Figure 4 plots the average number of migrations for a client until it converges. Recall that these migrations do not happen in the data session domain but in the control domain. Therefore this graph indicates the computation overhead and message exchange overhead of each algorithm rather than the actual session migration overhead. With no surprise, the tree-based search scheme results in less migrations than the full search case. However, the non-tree search case does not require an excessive number of migrations, either, increasing the number of migrations by only a small constant factor. In particular, the uncoordinated client migration (ZIZO-Sr-C) seems to strike a good balance between performance and complexity.

Finally we evaluate the adaptivity of the proposed algorithm by considering the case when the clients incrementally join the gaming network. Whenever a client joins the network, we perform the server selection algorithm on top of the current allocation obtained when the last client joined. Figure 5 presents the result when finally all 50 clients have joined. Compared to Figure 3(a) when all the clients are given initially, we observe that the "on-line" performance of our algorithm is similar to the "off-line" case. This is particularly true for the non-tree search case (ZIZO-Sr-C and

[1]We do not compare the results of the case of bounded server capacity to the greedy method since the greedy algorithm cannot be properly applied to such cases.

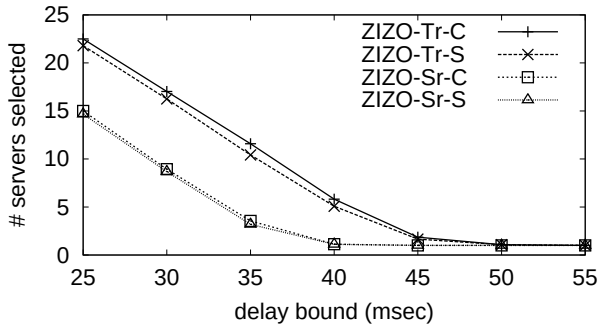


Figure 5: The number of servers allocated after the clients incrementally join

ZIZO-Sr-S). Thus we conclude that the proposed algorithm is adaptive to the session dynamics.

5. RELATED WORK

In the context of real-time group communication, our problem shares some similarity with the delay-constrained multicast tree construction problem [11, 15]. The goal of the delay-constrained multicast tree construction problem is to construct a lowest-cost multicast tree with an additional constraint of an upper bound on the end-to-end delay. Our work differs in that the delay present in server-client gaming networks is not the end-to-end delay but the nonlinear combination of client-to-server and server-to-server delay as given in our delay model.

Our work expands the dual of the minimum K-center problem [1], in which the number of servers is to be minimized when the maximum distance between clients and the nearest server is given, into a case where the inter-server delay must be taken into account as well. The $(log(N)+1)$-approximation bound is known for this problem, and we are currently investigating the possibility of finding a formal approximation bound for our problem.

Saha et al. [14] have proposed an on-line games hosting platform by developing middleware based on existing grid components. The proposed gaming architecture supports various aspects of game service provisioning and management including account management, game software update and distribution, and server resource provisioning. In the commercial world, Butterfly.net [5] is implementing the idea of building a scalable gaming infrastructure based on grid computing technology. However, both these works focus on architectural issues, leaving such issues as server allocation unaddressed.

6. CONCLUSION

In this paper, we presented a novel distributed algorithm that dynamically selects game servers for a group of game clients participating in large scale interactive online games. The goal of server selection is to minimize server resource usage while satisfying the real-time delay constraint. The proposed algorithm, called *zoom-in-zoom-out*, is adaptive to session dynamics and lets the clients select appropriate servers in a distributed manner such that the number of servers used by the game session is significantly reduced compared to when clients select the closest servers. We have considered various zoom-in techniques that the clients can implement. Using simulation, we have shown that the full search mechanism during the zoom-in procedure results in better performance than the tree-based alternative. We have also shown that clients can perform this zoom-in procedure without explicit coordination and still can achieve good results. Overall our algorithm has been shown to perform comparable to a greedy selection algorithm, which requires global state information and excessive amount of computation.

7. REFERENCES

[1] J. Bar-Ilan and D. Peleg. Approximation algorithms for selecting network centers. In *Proc. 2nd Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 519*, pages 343–354, 1991.

[2] D. Bauer, S. Rooney, and P. Scotton. Network infrastructure for massively distributed games. In *NetGames'02*, April 2002.

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

[4] BRITE. `http://www.cs.bu.edu/brite/`.

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

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[7] E. Cronin, B. Filstrup, and A. Kurc. A distributed multiplayer game server system, technical report, Univ. of michigan. Technical report, May 2001.

[8] 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.

[9] Y.-J. Lin and S. P. 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.

[10] M. Mauve, S. Fischer, and J. Widmer. A generic proxy system for networked computer games. In *NetGames'02*, April 2002.

[11] M. Parsa, Q. Zhu, and J. J. Garcia-Luna-Aceves. An iterative algorithm for delay-constrained minimum-cost multicasting. *IEEE/ACM Transactions on Networking*, 6(4), August 1998.

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

[13] L. Qiu, V. Padmanabham, and G. Voelker. On the placement of web server replicas. In *Proc. 20th IEEE INFOCOM 2001*, August 2001.

[14] D. Saha, S. Sahu, and A. Shaikh. A service platform for online games. In *NetGames'03*, May 2003.

[15] H. F. Salama, D. S. Reeves, and Y. Viniotis. An Efficient Delay-Constrained Minimum Spanning Tree Heuristic, Technical Report TR-96/46, North Carolina State University. Technical report, 1996.

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

[17] D. G. Thaler and C. V. Ravishankar. Distributed center-location algorithms. *IEEE Transactions on Selected Areas in Communications, vol. 15, issue 3*, April 1997.