2011

Server assignment in mirrored server environments

Shawn Farlow
Louisiana State University and Agricultural and Mechanical College, sfarlo2@lsu.edu

SERVER ASSIGNMENT IN MIRRORED SERVER ENVIRONMENTS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by

Shawn Farlow
B.S., Louisiana State University, 2004
December 2011

Table of Contents

Abstract

Massively multiplayer online games (MMOGs) may use peer-to-peer, client-server, or mirrored-server environments. Since there exist multiple server options in mirrored server environments, a problem arises when deciding to which server each player should connect. We propose three distinct algorithms that assign players within Quality of Service (QoS) as each player joins and leaves, taking into consideration whether a player already in QoS can be moved to place a newly joining player in QoS without sacrificing QoS for the moved player. Our results show that for certain numbers of servers and values of QoS, our algorithms increase the total number of players in QoS over a static player to server assignment and bear adapting to a wider variety of environments.

1. Introduction

The computer and video game industry is very popular. The ESRB reports that as of 2009, 67% of U.S. households play video games, and over 10.4 billion dollars of revenue were generated by computer and video games in that year [ESRB]. One type of video game is the massively multiplayer online game (MMOG). In an MMOG, hundred to hundreds of thousands of players connect to a persistent online virtual world and interact with each other there via some sort of game avatar. The persistent nature of the game, coupled with the number of players that may interact in it, requires some architecture that ensures each player sees the same game world regardless of how he or she connects to it or how many players are located in the same area of the game world as him or her. Architectures that perform this hosting and maintenance of these virtual worlds include client-server, peer-to-peer, and mirrored server.

In a client-server architecture, one central server hosts the game world. Each player that connects to the game world connects to the same server. A benefit of this type of architecture is consistency. Since each player connects to the same server, each player will see the same game world. However, since players may connect only to one server, this server can become overloaded quickly. Also, if a player's connection to this server is poor, he or she has no other choice of connection method. Therefore, the client-server architecture is not scalable and is not suitable for most MMOGs.

Peer-to-peer architectures do not use central servers. Instead, players' machines each connect to one another, and each one shares the workload required to maintain the game world. This workload can be divided up in several ways, including partitioning

the game world into areas called regions, and assigning each region to a player to host. This type of architecture reduces overhead, since it does not need to establish or maintain a server, but has other drawbacks. One drawback is that region assignment could cause a player to become overloaded if too many players visit that game world region. This could also result in poor game performance for that player, as his or her machine would spend a great deal of processing time handling processing for the region, leaving less time to handle updates for the player. Also, game world performance can become uncontrollable, since it depends on the capabilities of each user's machine. Therefore, it is not largely used in MMOGs.

Lastly, the mirrored server architecture contains several servers, each with the same copy of the game world. Players may connect to any of these servers and receive the same game experience. Each server is responsible for sending any updates that occur to it as a result of player actions to the other servers to maintain consistency. Players have more choices of server to which they can connect, so one poor server connection will not hinder him or her from playing the game. The mirrored server architecture is the one we shall consider in this paper.

Since multiple server choices exist in the mirrored server architecture, we must determine to which server each player will connect. Some authors [ARM2][GAR10] propose methods for players to narrow down choices themselves and choose a server to which to connect. Others, such as Webb *et al*. [WEBB ET AL], propose methods of assigning players to servers under some constraint (minimizing overall delay for all players, in their case). We will look at the environment of Ta *et al.* [TAETAL], in which

players are assigned to servers while attempting to maximize the number of players within Quality of Service, or QoS. For a player to be considered within QoS, his or her delay, or time to send and receive updates to his or her connected server, must be below a specific threshold. They refer to this as the Client Assignment Problem (CAP), and we will discuss this problem and its variations in further detail in Chapter 2. One drawback to their approach is that they assume an environment where no players have joined or have been assigned and do not consider that players may join or leave as gameplay ensues. In a persistent MMOG, players may join or leave at any time, and these joins or leaves could produce opportunities for other players without QoS to achieve QoS. The work I have done is to create three distinct algorithms that attempt to capitalize on these missed opportunities and increase the number of players with QoS over the method utilized by Ta *et al*.

The rest of the paper is laid out as follows. Chapter 2 discusses the work of Ta *et al.* and the background of the problem in further detail. Chapter 3 goes over related work in areas related to our problem. Chapter 4 discusses the algorithms I have developed and their operation in detail. Chapter 5 goes over the simulated comparisons of my algorithms with Ta *et al*.'s. Chapter 6 concludes the paper.

2. Problem Background

In this chapter we will discuss the background for the problem to which we are proposing a solution – namely, some of the environments that exist in this field of research (and more specifically, the one in which we will operate), some of the general work that has been done in this area, and the exact nature and parameters of our problem. We will go into detail about the work done by Ta *et al.* to solve this problem, highlight some areas of opportunity, and propose some initial solutions to capitalize on that opportunity.

2.1 Problem Background and Terms

In a *massively multiplayer online game* (MMOG), hundreds to hundreds of thousands of players connect to a persistent online virtual world and interact with each other there via some sort of game avatar. As discussed previously, there are a few environments that perform the hosting and maintenance of this virtual world. In a *client-server* environment, a single, centralized server hosts the game world, and each player connects to the same server. In a *peer-to-peer* environment, all players connect to each other, and each is responsible for a portion of the overall hosting and processing. In a *mirrored server* [MIRRSERV] environment for an MMOG, when a player joins, he or she connects to a server network, with each server in the network containing a copy of the game world. Each server, or mirror, communicates with the others to ensure that they each have the same copy of the game world at all times, so that each player has the same experience no matter to which server he or she connects. There are a finite number of servers, each with finite capacity, or number of players that may connect to it

simultaneously. This capacity is set so that, at maximum load, the server can provide an acceptable game experience to each player without becoming overloaded and thus providing poorer performance.

Instead of mirrored servers each handling copies of the entire game world, the game world can be partitioned into areas called *regions* [BES5], with each server handling one or more regions. These regions can be static or may change dynamically to accommodate a changing load as players move around the game world. Call the server handling a region as the *target server* for that region. Two ways of handling a player as he or she moves through multiple regions are to either directly connect the player to the target server of the current region, or to connect the player to a static server responsible for communicating all updates to the target server of the player's current region. In the latter, the static server responsible for communicating updates is called the *contact server*. In the former, the target server and contact server are the same server. The time to send and receive updates to and from a player to his or her contact server is the *contact delay. Target delay* refers to the time to send and receive updates to and from a player's target server, which includes the contact delay. If target or contact delay is too high, a player's game experience may suffer [CLAY2], since they will be unable to send and receive updates quickly enough to perform game actions in a timely fashion. For example, in a first-person online game, there is a sharp decrease in performance for delay greater than 100 ms. Quality of Service, or *QoS*, refers to the maximum target delay a player may have without suffering this decrease in game

performance. For a player to be considered *in QoS*, he or she must be connected to a server such that his or her target delay is below or equal to the value of QoS.

Ta *et al.* [TAETAL] present the *Client Assignment Problem* (CAP) for an MMOG. In the basic CAP, given a set of players, a set of servers, and the delays from the players to servers and servers to servers, the goal is to construct an assignment of players to servers that maximizes the number of players in QoS. A number of refinements of the CAP exist. One such is whether the set of players is given offline or online. For offline assignment, the entire set of players is available initially. For online assignment, no players are present initially, and the problem is to assign players to servers as they join and leave over time. Player-to-server assignments may change over time as demands change. Another refinement is that CAP exists for both region-based environments (CAP-R) and in mirrored server environments (CAP-M), with offline and online versions of each. For CAP-R, we compare the target delay to the QoS threshold; for CAP-M, we instead compare the contact delay.

2.2 Ta *et al.*'s algorithms

Ta *et al.* develop three algorithms to attempt to maximize the number of players within QoS for a variation of offline CAP-R. Each of the three GDA algorithms – GDA-1, GDA-2, and GDA-3 – starts with a set of players, with each player in a region, and consists of two steps – an initial assignment step of regions to servers followed by a refinement step.

In GDA-1, the initial assignment algorithm iterates through a list of all regions and assigns each region (and the players within it) to a server that will not become

overloaded by the assignment. The algorithm sets both the contact and target servers of each player in that region to be the assigned server. Once initial assignment is complete, the refinement algorithm sorts any players without QoS by decreasing order of target delay (to its current target server, which is the same as its contact server), and then, for each of these players, attempts to find a new contact server (with available capacity) that will reduce the target delay within the QoS threshold to which to connect the player. If it fails to find such a server, it then sets the player's contact server to the server that provides the least total delay to its target server and also has capacity. (This server does not satisfy QoS.)

GDA-2 has a different initial assignment algorithm, assigning each region (and its players) to the server that puts the fewest number of players in that region outside of QoS. It then sets the contact server of each player in that region to be the assigned server. There is no refinement step for GDA-2. GDA-3 is a combination of parts of GDA-1 and GDA-2. The algorithm begins with the initial assignment of GDA-2, where regions are assigned based on minimizing the number of players without QoS in that region. Refinement is the same as GDA-1, iterating through the list of players without QoS and attempting to assign them to servers with QoS.

My work consists of alternative approaches to the Client Assignment Problem in an online CAP-M environment instead of offline CAP-R. Algorithms for each of those environments can be easily adapted to the other. To adapt online algorithms to offline environments, since all players are available initially, we can use the online algorithm to assign players as we would in an online environment, but instead attempt player

assignment in descending order of delay, so that players with higher target server delays are assigned first. To adapt offline algorithms to run in online environments, we perform the assignment algorithm for each player as he or she joins, instead of once with a single pool of joining players. In CAP-M, target servers and contact servers are not separate, but this changes only the way we calculate delays and does not affect algorithm performance. Instead of target delay consisting of delay from a player to his or her contact server plus delay from that contact server to the player's target server, target delay is simply delay from player to his or her assigned server. Conversely, we can also modify the target delay calculation – from contact delay only, to contact delay plus delay from contact server to target server – if converting from online CAP-M to offline CAP-R. Thus, algorithms for offline CAP-R can be easily modified to work in online CAP-M, and vice versa. The primary goal of each of my algorithms is the same as Ta *et al.*'s – to increase the total number of players within QoS. The other differences between my algorithms and Ta *et al.*'s will be discussed in further detail in the Method chapter.

3. Related Work

To my knowledge, few authors have researched the exact environment in which Ta *et al*. operate, but several authors have contributed work towards the individual elements that comprise it – namely, assigning virtual regions to servers and bettering the player experience (whether through reducing average delay or achieving QoS). Some of these approaches have heavily influenced the direction of my own algorithms and will be discussed in detail here. This chapter will also discuss other works that provide examples of alternative methods to similar aspects of the Client Assignment Problem.

Armitage [ARM2] uses autonomous systems to improve discovery of servers in an MMOG. When playing an online game that is server-based and the server must be chosen manually, a player typically cannot know which servers provide a suitable delay without probing each individual server. The time taken for a server to send back a reply after being probed is known as the round trip time (RTT). In an online game such as Counterstrike: Source (which boasts around thirty thousand servers), probing each server individually would be impractical. Armitage's system reduces total probe time and reduces the number of probes by probing servers likely to have low delay before servers likely to have high delay (thus finding suitable servers more quickly), stopping when the delay reaches a user-defined threshold. It also performs its operations on the client side so as not to generate unnecessary additional network traffic.

Armitage's algorithm first categorizes servers into clusters using each game server's IP address to identify topologically distinct regions of the Internet as a general

indicator of servers that are physically close together. Next, it probes a sample of servers in each cluster to rank a cluster of servers' potential RTT. It then probes clusters, in order of ascending rank, for their individual servers' RTTs. In this way, it can more easily discover servers with lower RTTs and find a suitable server for low latency gameplay.

Gargolinski *et al*. [GAR10] look at server selection from the client's perspective. Current game server selection tools typically consider only the perspective of a single client attempting to join a game session, but in some games, including MMOGs, clients may seek to play together on the same server or in a single session. The authors present an architecture for multiple people to select a server that is best for all of them without requiring some form of external communication to do so (such as a trial and error selection while talking over an instant messaging client or telephone). Clients who want to play together use a program called QStat, which sends packets to potential servers and uses the response time to estimate latency. Subject to some initial preferences (such as map selection or game type), one client, who is designated as the host, filters the potential server list and chooses a final server for all clients based on the lowest average latency to it. The authors do note that, while this is a simple approach, the host could choose based on some delay constraint or based on the standard deviation for fairness reasons, but this would be more complex.

Briceño *et al*. [BRI3] focus on fairness and scalability of MMOG environments, where fairness is measured as the difference in the maximum and minimum response times of users in the system, and scalability refers to the ability of the system to

maintain fairness regardless of the number of players. They consider an environment that has a main server (MaS) that controls game world state, with potential secondary servers (SSs) that assist the MaS with calculations. SSs are essentially users playing the game whose computers are chosen to become SSs. If there are any conflicts in game states among players on the same SS, then the SS resolves it and then communicates with the MaS, which resolves any further conflicts among SSs. There are four heuristics developed by the authors to choose the SSs, and they each perform the same tasks - determine the number of SSs, choose which users are converted to SSs, and assign users to the SSs and MaS. However, unlike my research, the system created by these heuristics does not maintain itself over time. If a player designated as SS leaves the game, then it does not choose another to take its place, nor are players reassigned to different SSs over time if a better choice exists. In addition, some of their heuristics perform too slowly to be considered for real-time use, whereas the algorithms I have developed are all intended for use in real-time.

In contrast to assigning players to static regions, some systems operate in an MMOG environment that partitions game regions transparently to players and spreads regions among multiple servers, allowing interaction between objects on adjacent servers. Systems can modify these regions as needed based on how players move and interact, so if a region becomes densely populated because of events in the game (known as “hotspots”), then the system can partition it further to be handled by more servers so performance does not degrade. Hotspots commonly occur in MMOGs and can overload a server under a static partitioning algorithm - hence the need for dynamic

algorithms. J. Chen *et al.*'s [CHE6] "locality aware dynamic load management" is such an algorithm. It is a decentralized dynamic partitioning algorithm that attempts to reduce QoS violations, run local to each server to reduce overhead. While other algorithms involve all possible nodes in a global heuristic, theirs considers the local server and neighbor servers to determine where repartitioning may be needed. Each local algorithm checks for QoS violations and determines the cause, whether from overloading or excessive communication. If the cause is overloading, the algorithm sheds load from the overloaded servers onto other servers (preferably neighboring servers), assuming they can handle the additional computation. If the cause is excessive communication, the algorithm combines underloaded neighboring server partitions. The algorithm takes into consideration its own effect on performance and attempts to not reassign too often, as this can have a considerable effect on game performance itself. Communication between servers is also taken into consideration and reduced, as too much of it could also impact performance. Thus the algorithm seeks to increase performance and reduce QoS violations, all while being mindful of its own impact on performance. Since my algorithms consider a mirrored server environment, they do not consider or perform game world partitioning, but the ability to reduce QoS violations while not impacting performance is a desirable trait in either environment.

Chertov and Fahmy [CF7] also investigate an adaptive partitioning architecture that partitions virtual space into regions. Similar to Chen *et al.*'s system [CHE6], theirs takes advantage of clients gathering around points of interest to dynamically repartition and balance loads. The architecture also aims to efficiently handle sparse

environments, or areas where few clients are gathering. Each server handles a rectangular region that can grow or shrink in size to accommodate a varying number of players. The servers attempt to keep the number of players handled by each server balanced as players move to different game regions. Regions may end up overlapping so that multiple servers handle the same player, but the system takes steps to eliminate these overlaps and assign the player to one server or the other. Their environment is similar to mine in that it sets a limit on number of players per server and can reassign players during the course of gameplay as needs change due to moving players. However, mine does not attempt to keep an even player distribution among servers and does not consider proximity of players' avatars in the game world (i.e., game regions).

Beskow *et al*. [BES5] consider an MMOG where the game world is dynamically partitioned into regions based on player distribution in an attempt to lower the overall latency of interacting players. There are three key elements to the authors' approach – region assignment, a distributed name server / migration, and latency estimation. (Region migration, while important to the authors' work, is not covered by my work and will not be discussed here.) Region assignment consists of choosing a server to which to assign a region such that all players interacting in the region can send each other updates within some latency constraint. This assignment tries to ensure that there is not a server that could provide the players in a region with better performance. Generally, the algorithm, given some set of players in a region as an input, will choose the server for which the sum of latencies of shortest paths to those players is the smallest. One

server can end up hosting all regions, whereas in my work each server has a finite capacity.

Latency estimation is used by region assignment as a replacement for complete network monitoring because probing the entire network is costly and not scalable. To save time and processing, these techniques measure a subset of links and then estimate the rest based on those measurements. Two techniques are used - Netvigator, which provides more accurate results but is more difficult to implement, and Vivaldi, which is easier to implement but provides less accurate results. Netvigator uses landmark-based estimation, in which a set of landmark nodes is used to estimate others' relative network position. Vivaldi uses multidimensional-scaling based estimation, which uses statistical comparisons of data to make its estimates. These techniques are less important in my work than the authors' since region assignment is not considered, but still hold value since estimating latency is beneficial in algorithms like mine that use latency as a determining factor of server assignment.

Lee *et al*. [LEE11] consider a client-server environment with many servers but few clients connected to each, with servers connected to one another by low latency links. Each player connects to a server that is responsible for forwarding his or her actions to other players with whom he or she is interacting in the same area of the game world. The authors' zoom in / zoom out (ZIZO) distributed algorithm assigns players to servers such that a small number of servers is selected while keeping the largest delay between any pair of interacting players, referred to as the synchronization delay, below

a bound. An ideal assignment would have all players assigned to one server, assuming that satisfied the synchronization delay bound.

The authors observe that the low latency links between servers can reduce the delay between interacting players that are separated by a large physical distance. Two players can each connect to a server that is physically close to him or her, and each of those servers can send updates to the other via the server network. While this reduces the synchronization delay, it increases the number of servers used. The ZIZO algorithm seeks to find a middle ground by choosing an assignment such that the synchronization delay is kept below the bound while reducing the number of servers used in the player-to-server assignment. Given a set of servers and an initial assignment of players to servers (players are initially assigned to the server physically closest to them), the algorithm chooses a server that is closest to the physical center of all players and designated as such. During the “zoom in” portion of ZIZO, the algorithm checks the servers nearby to each player’s initial server to see if assigning the player to one of those servers would still keep the synchronization delay below its bound, assuming that server is closer to the previously designated center server than the player’s current server. It continues to do this until it either can no longer find a server closer to the central server that satisfies the delay bound or gets to the central server itself. It then migrates the player to that server. If every player is migrated to the central server then the algorithm stops because we have satisfied the synchronization delay bound and every player is connected to the same server. During the “zoom out” portion, we perform a similar operation, but in the reverse direction, and considering only servers with

assigned players. Thus, if the algorithm can assign a player to a server that is farther away from the central server, closer to the player, and still satisfy the delay bound, it will. Since we consider only servers that have already had players assigned to them, this will either keep the number of servers used the same or reduce it by assigning more players to the same servers.

During the ZIZO algorithm, there are three methods by which clients can attempt to migrate to servers - full search, partial search, and tree search. The full search is simple - clients consider all servers as potential moves and pick the one with least delay. Partial search considers only servers that will move a client either closer to or further from the core (no lateral moves, in other words). Tree search designates a "parent" for each server (where the parent is a server that is closer to the core), and allows moves only to that parent server when moves occur. Thus, clients move in a designated path. According to the authors, this algorithm performs better than a centralized greedy algorithm would, since it does not require upkeep of global information. The zoom out process is essential to the function of this algorithm, as it contributes to an overall saving of twenty percent of server resources. While their algorithms seek to reduce overall delay by considering moves for all players at once, which is much different than my environment, the concept of continual upkeep of server assignments and looking for better choices on a regular basis is the foundation of my algorithms.

Y. Chen *et al*. [CHE8] present heuristic algorithms that create subnetworks of servers based on some delay constraint, given a set of servers and clients. These

algorithms work for both client-server models and peer-to-peer models, though the authors focus on the client-server aspect. Though their algorithms and mine both seek to reduce delay, theirs seek to reduce average delay (where each player satisfies QoS), while mine seek to reduce the number of players without QoS.

The authors have designed two algorithms - the first to reduce overall delay, the second to reduce the number of servers involved. The first works by representing each server as a graph node. It constructs a shortest path tree for each server, with the root being that particular server. The algorithm assigns clients to these servers, with minimal delay from client to server. Once it constructs all of these trees, the algorithm chooses the tree with the lowest overall latency. This algorithm uses only a subset of the overall server group as its solution, unlike my algorithms, which attempt to utilize all available servers. The second algorithm starts with a single server node and then adds clients subject to satisfaction of some delay constraint until no more clients can be adding without violating the constraint. It then chooses a neighboring server, joins it to the tree, and then adds clients to it in the same fashion. It performs this repeatedly until it has joined all clients without exceeding the delay bound. Again, this algorithm does not use all possible servers in its solution.

The same authors went on to further improve these algorithms [CHE9] by subjecting their algorithms to an additional goal - minimizing the variance in delay between players, which is similar to the idea of fairness proposed by Briceño *et al*. [BRI3]. For the client-server setup, given an initial assignment of a root server and a subset of overall servers, the algorithm finds shortest paths from the root to each server

in the subset. It then finds the shortest distance from each client to the root using these paths. It then uses this information to form minimal delay variation chains, choosing the one with the smallest variance.

Zhang and Tang [ZHANGTANG] seek to assign clients to distributed servers. However, instead of seeking to minimize client-to-server delays like some of the other authors we have discussed, they seek to also minimize client-to-client delays as part of the overall delay minimization. The authors implement two algorithms – Greedy-Assignment and Distributed-Greedy-Assignment.

The Greedy-Assignment algorithm starts with the set of all clients and the set of all servers. At each step of the algorithm, for each joining player, the algorithm calculates the potential interaction path length from that player to all other players that have joined. The interaction path length consists of the delay from the player to their potential server, the delay from that server to each of the other servers to which players who have already joined are connected, and the delay from those already joined players to their servers. The algorithm then makes the assignment with minimum interaction path length. The Distributed-Greedy-Assignment algorithm starts with some initial assignment. Each client then checks to see if changing its assigned server will decrease the overall interaction path lengths. This continues for each client, one at a time, until no more changes can be made that will reduce the overall path lengths. When considering path length reduction, each client considers only paths that involve him or herself, since a change in server assignment would not reduce path length for paths where that client is not included. The authors' simulation results show that for the changes that are made

using this algorithm, ninety percent of changes occur in the first iteration, and ninety nine percent occur within three iterations, so one iteration is sufficient to produce adequate results. The idea of factoring client-to-client delays into calculations for client to server assignment is novel and provides some interesting ways that our algorithms may be adapted to operate in other environments.

Ta and Zhou [TAZHOU1] [TAZHOU2] propose the client assignment problem. The client assignment problem proposed in this paper is similar to the CAP in our problem background, although here the authors are assigning zones to servers such that the assignment minimizes the number of players outside of QoS. In the paper used for our problem the authors mentioned zones, but ended up assigning individual players. The authors propose two algorithms for this version of the CAP – Greedy-1 and Greedy-2.

For Greedy-1, zones are assigned to servers based on a “desirability” factor, which is based on, among other things, the number of players that will be assigned without QoS. The assignments are made so that desirability is high, or number of players assigned without QoS is minimal. For Greedy-2, the algorithm performs assignments in the same way, but the desirability is measured differently – instead of minimizing the number of players without QoS, the algorithm measures the average of delays from all clients in the zone to be assigned to the potential server and makes the assignment with minimum average. Greedy-1 outperforms Greedy-2, except in the case when imperfect data is considered. In this case, the authors consider an error factor in the delay data, instead of considering it to be absolute. This idea of imperfect data is an

interesting concept that will be added to future work for our algorithms. Also, in the evaluation of their algorithms in [TAZHOU2], the authors use a correlation parameter to simulate the tendency of players who are close in the physical world to be near each other in the virtual world. This is not an aspect considered in our environment, but may also prove interesting for future modifications to our simulations.

Webb *et al*. [WEBBETAL][WEBBETAL2] present the Client to Mirror Assignment (CMA) problem in a mirrored server architecture, which closely corresponds to the foundation of my work, and is similar to the CAP problem that has been previously discussed. Although their approach to the CMA problem involves reducing the average delay of players and not only satisfying QoS, our algorithms have similar ideas and both seek to improve upon some flaws apparent in other methods, such as Ta *et al.* 's.

The authors propose two pairs of algorithms – J-SA / L-SA and J-Greedy / L-Greedy. Each pair handles joins and leaves, respectively. The SA algorithms improve player delay by building chains of player moves – if an opportunity exists to move a player from one server to another and improve his or her delay by moving another player, they capitalize on it, and continue to seek opportunities for the player being moved, building a "chain" of moves that will reduce the average player delay. The Greedy algorithms narrow the scope, instead seeking to quickly assign players to servers based on which servers are available and not what could become available if players were moved. Both of these methods are important to my work – my algorithms will move players to different servers if QoS could be achieved for a joining player by making that move, although one of them makes only one move per joining player

instead of constructing a chain of moves. A form of greedy joining also occurs as a failsafe in my algorithms, seeking to quickly join a player to a server in case other methods do not provide an adequate result.

4. Method

Examining Ta *et al.*'s approach to the client assignment problem reveals some drawbacks. In GDA-1 (and in the other Ta *et al.* algorithms), refinement occurs only once. Dynamic player joins and leaves are not considered. Space may open up on a previously full server that would satisfy QoS for a player without QoS, or a player within QoS might be able to be moved to another server satisfying QoS to make room for a joining player. Several possible solutions present themselves in solving for these scenarios that go unaccounted for in Ta *et al.*'s algorithms.

One approach is to keep a list of players without QoS and their delay to each server (or to which server would give them the lowest latency). When a player leaves a server that is full, check this list of players in an attempt to find a player whose "best server," or server with lowest latency, matches the previously full server. If a match is found, then move that player into the vacant spot. If none match, then check the "second best server" for each player in the list in the same way. This continues until an assignment is found. I have loosely based my approach in the BUMP-ON-LEAVE algorithm on this.

Another approach is to initially attempt to assign a player to the server with least latency (whether it has capacity is not considered). If the server has capacity and satisfies QoS, then finalize the assignment. If there is not capacity, then attempt to move a player that does have QoS on that server to another server that will still satisfy QoS and then join the new player to the opened position. The concept of moving

players around to better assign joining players is the basis for the BUMP-ON-JOIN-OR-LEAVE algorithm and the REASSIGNMENT THRESHOLD algorithm also uses it.

Third, instead of incrementally changing assignments with each join and leave, set a threshold for the ratio of players without QoS to overall players. While below the threshold, take no action. If the ratio crosses the threshold, then attempt to reassign those players without QoS to servers that would satisfy QoS. If no players are reassigned, then increase the threshold. This should reduce the amount of time spent in reassignment and reduce the frequency of reassignment. These ideas are the basis for the REASSIGNMENT THRESHOLD algorithm.

4.1 BUMP-ON-LEAVE

BUMP-ON-LEAVE consists of a joining algorithm and a matching leaving algorithm. The joining algorithm attempts to assign a player to the first server that satisfies QoS. It randomly permutes the server list for each join, because if we were we to attempt player joins to servers in the same order – say, S_0, S_1, …, S_{m-1} – S_0 would become full quickly since it would always be the first server checked. (BUMP-ON-JOIN-OR-LEAVE and REASSIGNMENT THRESHOLD also perform this random permutation for each join.) If no server that satisfies QoS and has capacity can be found, then it joins the player to the server with least delay among those that do have capacity and keeps track of which server overall has least delay for the player (assuming this server will satisfy QoS). It stores the identity of this player in a list for the least delay server. Each server has such a list of players for whom the server is the best choice. The leaving algorithm uses these lists to determine player reassignments. If a player leaves a server that previously had

no capacity and the server's list is non-empty, then the algorithm moves the first player in that list from his or her non-QoS server to this newly-opened server. If this move produces an opening on a previously full server, the algorithm will attempt to move another player into that opening. This will continue until we reach a server with an empty master list or a player moves from a server that was not already full. If the list contains no players or the server was not full before a player left it, then the algorithm takes no action. This is because either this server is not best for any non-QoS player or this server does not provide QoS to any non-QoS player as a player would have otherwise joined earlier. The full description and pseudocode for this algorithm are as follows.

BUMP-ON-LEAVE Joining Algorithm

Inputs:

S - set of *m* servers = $\{S_0, S_1, ..., S_{m-1}\}$. The algorithm randomly permutes the order of these servers for each join.
ML – set of lists = $\{M_0, M_1, \ldots M_{m-1}\}$. ML_j is a list of players P_i without QoS for whom S_j is P_i's best server; that is, the server that would minimize P_i's delay and put P_i in QoS. Each entry in the list has two elements: the player index and the player's assigned server.
SB - an array of best servers for players without QoS, indexed by player. Used to find to which *ML* a player belongs (if any).
SC - an array of server availabilities (the number of spots open on each server). If a particular server is full, the value for the server in the array will be 0.
q - QoS threshold

Outputs: *ML, SC, SB*

Variables:

d(i, j) - delay from player *i* to server *j.*
smin – index of server with least delay from player to it
scap – index of server with least delay from player to it and available capacity

Steps:

Upon a player P_i joining
smin, scap <- NULL // initialization
for *j* <-0 to *m-1*
    determine *d(i, j)* // determine delay to server *j. d(i,NULL)* is assumed to be infinity
    if *d(i, j)* ≤ *q* && *SC*[*j*] > 0 // if the server satisfies QoS and has capacity

```
                join P_i to S_j
                decrement SC[ j ]
                end operation

        else
                if d(i, j) < d(i, smin)        // if the server has a smaller delay than any server thus far
                        smin <- j
                if d(i, j) < d(i, scap) && SC[ j ] > 0 // as above, but it also has available capacity
                        scap <- j

// if we get here we were not able to find a server that had capacity and satisfied QoS
if (smin == scap)          // if these servers are the same, there is no server that will satisfy QoS, simply
                        // join the one with least delay...assumed there exists a server with capacity
        join P_i to S_scap
        decrement SC[ scap ]
        end operation

else    // else P_i will join S_scap and be added to the list to try to join S_smin later
        add (i, scap) to ML_smin    // add P_i and S_scap to the ML of P_i's best server
        SB[ i ] <- smin             // store the identity of P_i's best server
        join P_i to S_scap
        decrement SC[ scap ]
        end operation
```

BUMP-ON-LEAVE Leaving Algorithm

Inputs:

S - set of m servers = $\{S_0, S_1, ..., S_{m-1}\}$
ML – set of lists = $\{M_0, M_1, \ldots M_{m-1}\}$. ML_j is a list of players P_i without QoS for whom S_j is P_i's best server; that is, the server that would minimize P_i's delay and put P_i in QoS. Each entry in the list has two elements: the player index and the player's assigned server.
SB - an array of best servers for players without QoS, indexed by player. Used to find to which ML a player belongs (if any).
SC - an array of server availabilities (the number of spots open on each server). If a particular server is full, the value for the server in the array will be 0.

Outputs: *ML, SC, SB*

Steps:

Upon a player P_i leaving server S_j

```
increment SC[ j ]
if k <- SB[ i ] != NULL     // if the player is not already connected to his or her best server
        remove P_i from ML_k
        SB[ i ] <- NULL
        k <- NULL
if SC[ j ] != 1      // if the server wasn't previously full then no opportunity for a move exists
        end operation
 while ( ML_j != NULL ) // while there exists a master list for the server, a player can be moved there
```

```
(x, sx) <- ML_j ( 0 ) // get the first element of the list, which shows player P_x connected to server S_Sx
move P_x to server S_j
remove P_x from ML_j
SB[ x ] <- NULL
increment SC[sx]
decrement SC[ j ]
if SC[ sx ] != 1 // if the server wasn't already full then end operation
        end operation
j <- sx // set S_sx as the new server to try to move a player to and continue the loop
```

4.2 BUMP-ON-JOIN-OR-LEAVE

The BUMP-ON-JOIN-OR-LEAVE algorithm consists of a matching join and leave algorithm. Instead of handling all of the movement work during player leaves like the BUMP-ON-LEAVE algorithm, this algorithm attempts to make space on QoS servers when a player joins as well as when a player leaves.

When a player joins, the algorithm iterates through servers until it finds one that satisfies QoS for that player and checks its capacity. If the server has capacity, the algorithm joins the player to it while looking for another server that satisfies QoS to set as the player's alternative server. If the server does not have capacity, the algorithm sets it as the player's alternative and continues to look for another server that satisfies QoS and also has capacity. If no adequate server is found, then the algorithm attempts to bump a second player to his or her alternative server from a server that would satisfy QoS for the joining player. If the joining player still cannot be joined within QoS in any of these ways, the algorithm joins him or her to a server with capacity and adds him or her to a list of players without QoS to be later considered for movement into QoS when another player leaves.

The leaving algorithm is similar to the leaving algorithm of BUMP-ON-LEAVE. When a player leaves a server that was previously full, the algorithm iterates through

the list of players without QoS and checks to see if QoS would be satisfied for any of those players if they moved to the newly-open server. If so, then it moves that player into the newly-opened spot. However, the algorithm does not form a chain of moves, unlike BUMP-ON-LEAVE. The full description and pseudocode for the algorithm are as follows.

BUMP-ON-JOIN-OR-LEAVE Joining Algorithm

Inputs:

S - set of *m* servers = $\{S_0, S_1, ..., S_{m-1}\}$. The algorithm randomly permutes the order of these servers for each join.
SC - an array of server availabilities (the number of spots open on each server). If a particular server is full, the value for the server in the array will be 0.
Z - an unsorted list of players that have been joined to a server but are without QoS, encompasses all servers. Each element is of the form (*w, x, y*), where *w* is the player index, *x* is index of the player's secondary server, and *y* is the index of the player's current server.
PD – set of lists = $\{PD_0, PD_1, \ldots, PD_{m-1}\}$. Each list PD_j is unordered and contains the indices of all players on S_j that have QoS and also have a secondary server. Each element of PD_j is in the format (*w, x*), where *w* contains the player's index and *x* contains the index of the player's secondary server.
q - QoS threshold

Outputs: *C, Z, PD*

Variables:

spri - primary server candidate satisfying QoS for P_i
ssec - secondary server candidate satisfying QoS for P_i
d(i, j) - delay from player *i* to server *j*
Min - server index of assigned server from greedy join

Steps:

```
Upon a player Pi joining
initialize spri <- NULL, ssec <- NULL
for j <- 0 to m-1
        Determine d(i, j)
        if d(i, j) ≤ q
                if SC [ j ] == 0    //if we find a full server that satisfies qos
                        if ssec == NULL//if we have no secondary server
```

```
                    ssec <- j
                    if spri != NULL   //this means we have both a sec and pri, done
                        add (i, ssec) to PD_spri
                        end operation
            else                    //we've found a server satisfying qos with capacity
                if spri  == NULL            //we have no primary server
                    spri <- j
                    join P_i to S_spri      //join immediately - server with qos and cap is ideal
                    decrement SC [ j ]
                    if ssec != NULL             //we have pri and sec - done
                        add (i, ssec) to PD_spri
                        end operation
                else      //we already found a primary and joined, now we have our secondary
                    ssec <- j
                    add (i, ssec) to PD_spri
                    end operation

if spri != NULL and ssec == NULL
        end operation    // we have already joined a primary but found no secondary so we simply end

else      //went through the entire list and don't have a primary server, might have sec.
        for j <-0 to m-1
            if d(i, j) ≤ q        //no server had capacity and QoS at this point so only check QoS
                t <- length ( PD_j )
                for k <- 0 to t-1  //iterating through PD
                    (x, sx ) <- PD_j (k)                    //get an element of PD
                    if SC[ sx ] != 0   //potential bumpee can be bumped
                        move P_x to S_sx
                        decrement SC[ sx ]
                        join P_i to S_j       //join player to newly-opened spot
                        if ssec != NULL             //only add to bump list if secondary
                            add (i, ssec) to PD_j
                        end operation

//if we get here either no server satisfied QoS or we could not bump anyone on servers that satisfied QoS
//to another server
join P_i to first server with capacity, store index of joined server in Min
//we can't join a server that satisfies QoS so join and let leave alg. handle
add (i, ssec, Min) to Z    //add player to Z list for leaving algorithm's consideration
end operation
```

BUMP-ON-JOIN-OR-LEAVE Leaving Algorithm

Inputs:

S - set of *m* servers = $\{S_0, S_1, ..., S_{m-1}\}$. The algorithm randomly permutes the order of these servers for each join.
SC - an array of server availabilities (the number of spots open on each server). If a particular server is full, the value for the server in the array will be 0.

Z - an unsorted list of players that have been joined to a server but are without QoS, encompasses all servers. Each element is of the form (*w, x, y*), where *w* is the player index, *x* is index of the player's secondary server, and *y* is the index of the player's current server.
PD – set of lists = {PD_0, PD_1, …, PD_{m-1}}. Each list PD_j is unordered and contains the indices of all players on S_j that have QoS and also have a secondary server. Each element of PD_j is in the format (*w, x*), where *w* contains the player's index and *x* contains the index of the player's secondary server.
q - QoS threshold

Outputs: C, PD, Z

Variables:

d(i, j) - delay from player *i* to server *j*
ssec - secondary server candidate satisfying QoS for P_i

Steps:

Upon a player P_i leaving server S_j

```
remove Pi from PDj
increment SC[ j ]
if SC[ j ] == 1     //if the server was full but now it's not, look for someone to move
        l <- length(Z)
        for k <-0 to l-1    //iterate through Z list
                (x, ssec, sx) <- Z( k )      //get an element of Z
                determine d(x, j)
                if d(x, j) <= q
                        move Px from Ssx to Sj    //move the player into the newly opened spot
                        remove Px from Z
                        increment SC [ sx ]       //increment Px's previous server's capacity
                        decrement SC [ j ]
                        if ssec != j        //if new server isn't also the secondary, put into bump list
                                add (x, ssec) to PDj
                        end operation    //else sec. and new server are same, so doesn't go into bump list

        end operation    //if we get here there is no possible move so quit
else take no further action        //the server wasn't previously full so we don't consider it for movement
```

4.3 REASSIGNMENT THRESHOLD

The REASSIGNMENT THRESHOLD algorithm consists of a joining algorithm, leaving algorithm, and refinement algorithm. The overall algorithm attempts to reduce the number of times the "bumping" portion of the algorithm needs to be run. The goal of this

algorithm is place an acceptable number of players in QoS, but with fewer operations than the previous two algorithms, thus reducing computation time. In BUMP-ON-LEAVE and BUMP-ON-JOIN-OR-LEAVE, bumping takes place each time a player leaves or each time a player joins or leaves, respectively. The REASSIGNMENT THRESHOLD algorithm sets a threshold for the ratio of players without QoS to the total number of players. Until the player ratio exceeds this threshold, the algorithm does not attempt to improve any assignments.

The algorithm checks the ratio against the threshold at two times when the ratio increases: when a player joins without QoS or when a player with QoS leaves. Below the threshold, the algorithm adds a joining player to the first server that satisfies capacity and QoS. If such a server does not exist, then the algorithm adds the player to a list of players without QoS.

When the ratio crosses the threshold, the algorithm performs an assignment refinement process that iterates through the list of players without QoS and attempts to find a server for each that would satisfy QoS and moves the player to that server, if found. Once the algorithm has iterated through the entire list, if the ratio still exceeds the threshold, then the algorithm increases the threshold, since it cannot decrease the ratio of players without QoS at this time. The amount of increase in threshold is multiplicative and is specified before each simulation. If the ratio is below the threshold after refinement, then the algorithm decreases the threshold by the same factor until the threshold equals a base threshold value or cannot be decreased further without the ratio exceeding the threshold.

As a backup, the algorithm sets a trigger for the number of player joins and leaves that may occur without performing a refinement. If the number of joins and leaves exceeds this trigger, then the algorithm performs a refinement, regardless of ratio. This provides a failsafe in case the threshold is never exceeded so that players are not without QoS indefinitely. The full description and pseudocode for the REASSIGNMENT THRESHOLD algorithm are as follows.

REASSIGNMENT THRESHOLD Joining Algorithm

Inputs:

S - set of *m* servers = {S_0, S_1, ..., S_{m-1}}. The algorithm randomly permutes the order of these servers for each join.
SC - an array of server availabilities (the number of spots open on each server). If a particular server is full, the value for the server in the array will be 0.
Z - an unsorted list of players that have been joined to a server but are without QoS, encompasses all servers. Each element is of the form (*x, y*), where *x* is the player index and *y* is the index of the player's current server.
base_thr - lowest desired value of threshold
T - threshold for fraction of players without QoS; initially, *T* = *base_thr*
PC - total player count
jlcount - tracks number of joins/leaves since last refinement; initially 0
trigger - number to check total joins/leaves against to see if we need to try to run refine again
q - QoS threshold

Variables:

d(i, j) - delay from player *i* to server *j*

Outputs: *SC, Z, PC, T, jlcount*

Steps:

```
Upon a player P_i joining
increment jlcount
increment PC
for j <-0 to m-1
        if SC[ j ] > 0       // if we have capacity
                determine d(i, j)
                if d(i, j) ≤ q       // if we have capacity and QoS
                        join P_i to S_j
```

```
                    decrement SC( j )
                    end operation    // simply join player and adjust capacities/counts


for j <-0 to m-1  // if we get here we weren't able to find a server with capacity satisfying QoS – join to first
                  // server with capacity
        if SC[ j ] > 0
                  join P_i to S_j
                  add (i, j) to Z     // add player to the list of players without QoS
                  decrement SC[ j ]
                  // if the ratio of players without QoS to total players is bigger than T, call refinement
                  if ( length(Z) / PC ) > T
                            call refinement algorithm

if (jlcount > trigger ) //check # of moves & ratio vs threshold to see if we need to refine
        call refinement algorithm
end operation
```

REASSIGNMENT THRESHOLD Refinement Algorithm

Inputs:

S - set of *m* servers = $\{S_0, S_1, ..., S_{m-1}\}$. The algorithm randomly permutes the order of these servers for each refine.
SC - an array of server availabilities (the number of spots open on each server). If a particular server is full, the value for the server in the array will be 0.
Z - an unsorted list of players that have been joined to a server but are without QoS, encompasses all servers. Each element is of the form (*x, y*), where *x* is the player index and *y* is the index of the player's current server.
base_thr - lowest desired value of threshold
T - threshold for fraction of players without QoS; initially *T* = *base_thr*
jlcount - tracks number of joins/leaves since last refinement; initially 0
q - QoS threshold
threshChg – factor to change *T* by when adjusting threshold

Outputs: *SC, Z, jlcount, T*

Variables:

d(i, j) - delay from player *i* to server *j*

Method:

```
l <- length(Z)
for k <-0 to l-1   // iterate through Z
        (x, sx) <- Z( k )  // get an element of Z
        for j <-0 to m-1  // iterate through S
                  if SC[ j ] > 0
                            determine d(x, j)
                            if d(x, j) ≤ q       // if we have capacity and server satisfies QoS
                                      move player from S_sx to S_j
```

```
                                    increment SC[ sx ]      // adjust server capacities
                                    decrement SC[ j ]
                                    remove Px from Z
                                    if(jlcount != 0)
                                            jlcount = 0     // had 1 move so success, reset jlcount

if(jlcount != 0)    //if jlcount !=0 here then we moved no one, so increase T and reset count
        T <- T * threshChg
        jlcount = 0
else
        while(T ≥ base_thr && T > length(Z) / PC ) // if refinement was successful, we check vs. base
                                                   // threshold and lower the threshold just above where
                                                   // the current ratio is
                T <- T / threshChg
        T <- T * threshChg // adjust threshold back up one level, as it will be one level below where it
        //should be
end operation
```

REASSIGNMENT THRESHOLD Leaving Algorithm

Inputs:

S - set of *m* servers = $\{S_0, S_1, ..., S_{m-1}\}$.
SC - an array of server availabilities (the number of spots open on each server). If a particular server is full, the value for the server in the array will be 0.
Z - an unsorted list of players that have been joined to a server but are without QoS, encompasses all servers. Each element is of the form (*x, y*), where *x* is the player index and *y* is the index of the player's current server.
base_thr - lowest desired threshold value
T - threshold for fraction of players without QoS; initially *T* = *base_thr*
PC - total player count
jlcount - tracks number of joins/leaves since last refinement; initially 0
q - QoS threshold
trigger - number to check total joins/leaves against to see if we need to try to run refine again

Outputs: *SC, Z, PC, T, jlcount*

Variables:

d(i, j) - delay from player *i* to server *j*

Steps:

```
Upon player Pi leaving server Sj
increment jlcount
increment SC[ j ]           // increment capacity of server player is leaving
decrement PC                // decrease total player count
if d(i, j) ≤ q          // if the player had QoS
        if( length(Z) / PC ) > T    // if the result of the leave causes ratio to go up above threshold
                call refinement algorithm
```

```
//else the player did not have QoS, and by leaving our ratio gets better
else remove P_i from Z

if (jlcount > trigger) //check # of moves & ratio vs threshold to see if we need to refine
        call refinement algorithm
end operation
```

5. Simulation

5.1 Background/Operating Environment

In order to compare the performance of my algorithms with Ta *et al.*'s, we simulate the operation of each one in the same environment and take several measurements. For each algorithm, many simulations are run, tweaking several variables for each simulation. These variables include the number of servers available for players to join, the capacity of each server, the particular value of QoS, and the range of potential server delays each player can have. We also consider the effect that server load has on each algorithm's performance by making it a variable. Each simulation, the simulator loads each simulated server to a specific amount and then takes a snapshot of the simulated algorithm's actions over a period of time. Our aim is to be able to simulate many types of environments by allowing control over nearly all aspects of the simulation using these variables.

Each simulation starts by generating a two dimensional array of players and their delays to each server, using pseudorandom values for each delay based on the range specified. Once it fills each server to the specified load using the simulated algorithm's join algorithm, the simulation begins player joins, leaves and other operations based on the simulated algorithm. Each simulation cycle there is a chance of a player join and/or a player leave. The simulator pseudorandomly generates a one or zero value for each potential join and potential leave. If the value is one, the join or leave occurs. Since there are two values generated, each cycle there can be a join only, a leave only, both a join and a leave, or no action. If a player join or leave occurs, the player to join or leave

is chosen pseudorandomly. The simulation captures various results for each algorithm, starting from the beginning of the simulation and ending after a prespecified number of cycles. These results include number of players joined within QoS, number of players joined without QoS, number of players moved into QoS from outside of QoS, and algorithm-specific statistics such as number of refine calls by the REASSIGNMENT THRESHOLD algorithm. These statistics are captured for every leaving player – players that have joined but have not left are not considered.

Because each algorithm operates differently, some facets of them are not directly comparable and other considerations or modifications need to be made. Ta *et al.*'s algorithms consider a slightly different environment than my algorithms do, so I have adapted their algorithms to suit my environment in order to compare performance. GDA-1, GDA-2 and GDA-3 have very few differences among them when converted to my simulation environment since mine does not consider regions or target/contact servers, so my simulator simulates one overall "GDA" algorithm. This algorithm assigns players to servers using the GDA-1 refinement method. Ta *et al.* make no changes for leaving players in their algorithms, and neither does my simulated GDA algorithm.

We consider the REASSIGNMENT THRESHOLD algorithm in a different environment than the other algorithms during these simulations, since that algorithm is based on a ratio of players without QoS to overall players. If placed in an environment with a specific server load and a steady join/leave rate like the other two, it does little beneficial work since the threshold is rarely, if ever, crossed. Instead, it targets an environment with variable join and leave rates where the threshold is likely to be crossed.

The first set of simulation results below compares the Bump-On-Leave (B-O-L), Bump-On-Join-or-Leave (B-O-J-o-L), and GDA algorithms' performance. The REASSIGNMENT THRESHOLD algorithm simulation results immediately follow those. Unless otherwise noted, in these graphs we consider a QoS value of 100, with a potential delay range of 0-500. These values were chosen based on the acceptable delays set for online games using first-person avatars by Claypool [CLAY2]. The number of servers ranges from ten to forty. Fewer than ten servers results in poor or unpredictable results, because there rarely exist alternative servers to which to bump players based on the way delays are generated (pseudorandomly, not according to a distribution). For greater than forty servers, there are many options for players to join to upfront, so nearly all players are assigned within QoS initially. For similar reasons, server capacity ranges from forty open spaces per server to one hundred open spaces per server. The exact number of players that join and leave varies per simulation because of the randomness of joins and leaves, but averages twenty-five hundred players. Each data point on each graph represents an average of the results of ten simulations.

5.2 Simulation Results – B-O-L, B-O-J-o-L, GDA

Figure 5.1 shows the number of players leaving with QoS for B-O-L, B-O-J-o-L, and GDA in an environment of ten servers for varying loads. For this number of servers, until the load becomes high (ninety percent) the algorithms perform nearly identically. All three algorithms assign nearly every player within QoS by the time that player has left (and B-O-J-o-L assigns every player within QoS by the time he or she leaves). Once the

load gets to ninety percent, performance begins to vary slightly among the three algorithms. GDA performs better than B-O-L in this environment, though simulation results for this number of servers had high variance. For future work we will consider a different distribution of player delays so that we have a more predictable outcome even for an environment of ten or fewer servers.

Figure 5.2 shows the same comparison as 5.1, but for twenty servers instead of ten. With more servers, B-O-L performs slightly better than GDA, and B-O-J-o-L continues to assign each player within QoS by the time he or she leaves. As with the previous results, each algorithm's performance is similar until load reaches ninety percent, and there it is separated only by a few percent.

Figure 5.3 shows similar results as the previous two figures, with similar performance among the algorithms until ninety percent load, but the performance benefit of my algorithms over GDA is slightly smaller, since as the number of servers increases, the initial assignment for each algorithm assigns more players within QoS, so fewer players need to be moved. Figure 5.4 shows results for forty servers, at which point each algorithm is able to assign all players within QoS initially.

To further separate the work done by my algorithms, the following figures show the percentage of players leaving with QoS that were put into QoS by the algorithms. Since GDA does not place players into QoS after the initial assignment, for it this percentage is always zero. Figure 5.5 shows these results for an environment of ten servers at varying loads. Although the percentages are small for loads less than 90%, B-O-L outperforms B-O-J-o-L, which shows that bumping a chain of players has a

performance benefit over bumping a single player each operation in an environment of ten servers.

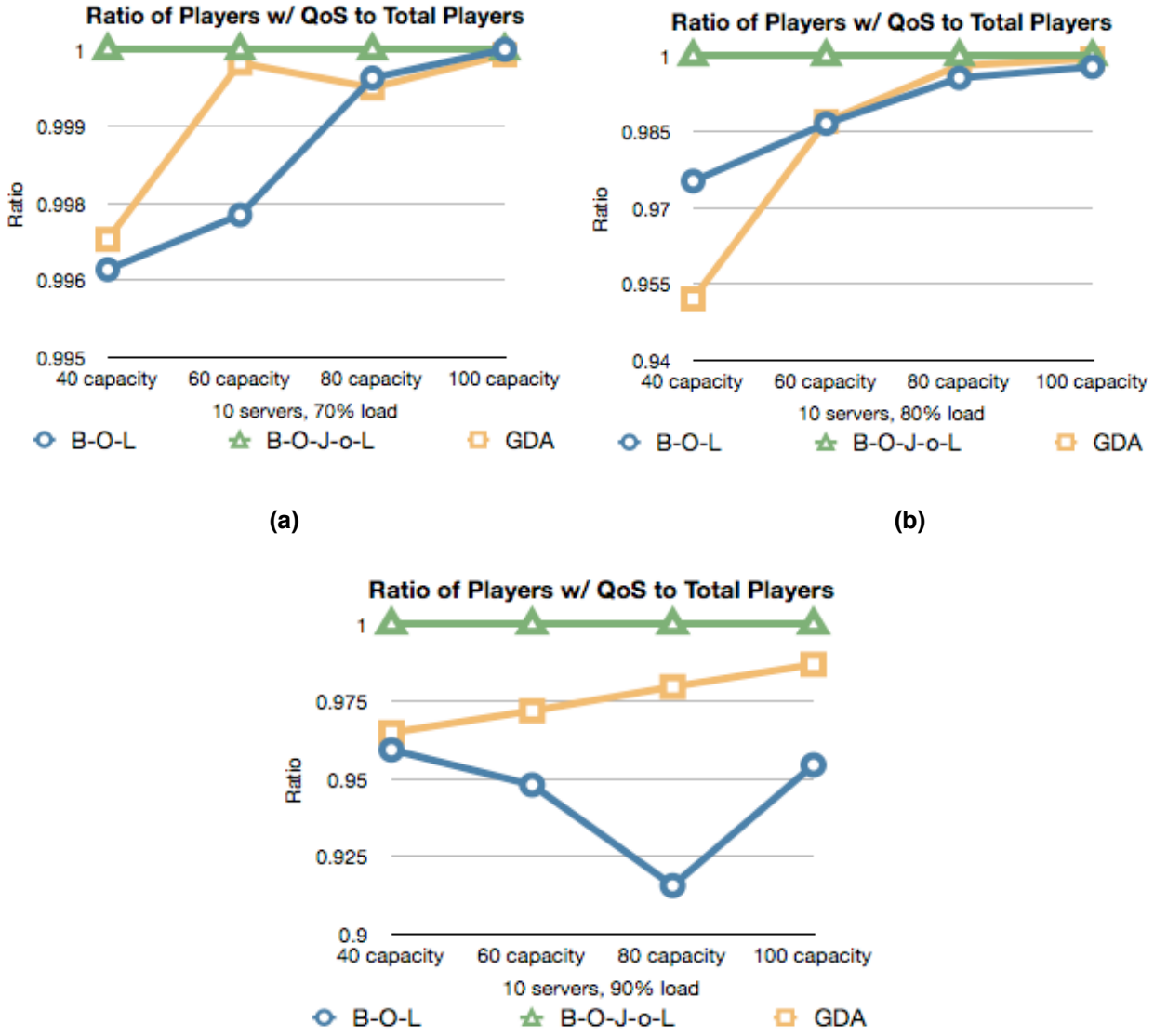


Figure 5.1. Ratio of players leaving within QoS (from initial assignment or movement during play) to total players that have left, for ten servers at loads of (a) 70%, (b) 80%, and (c) 90%.

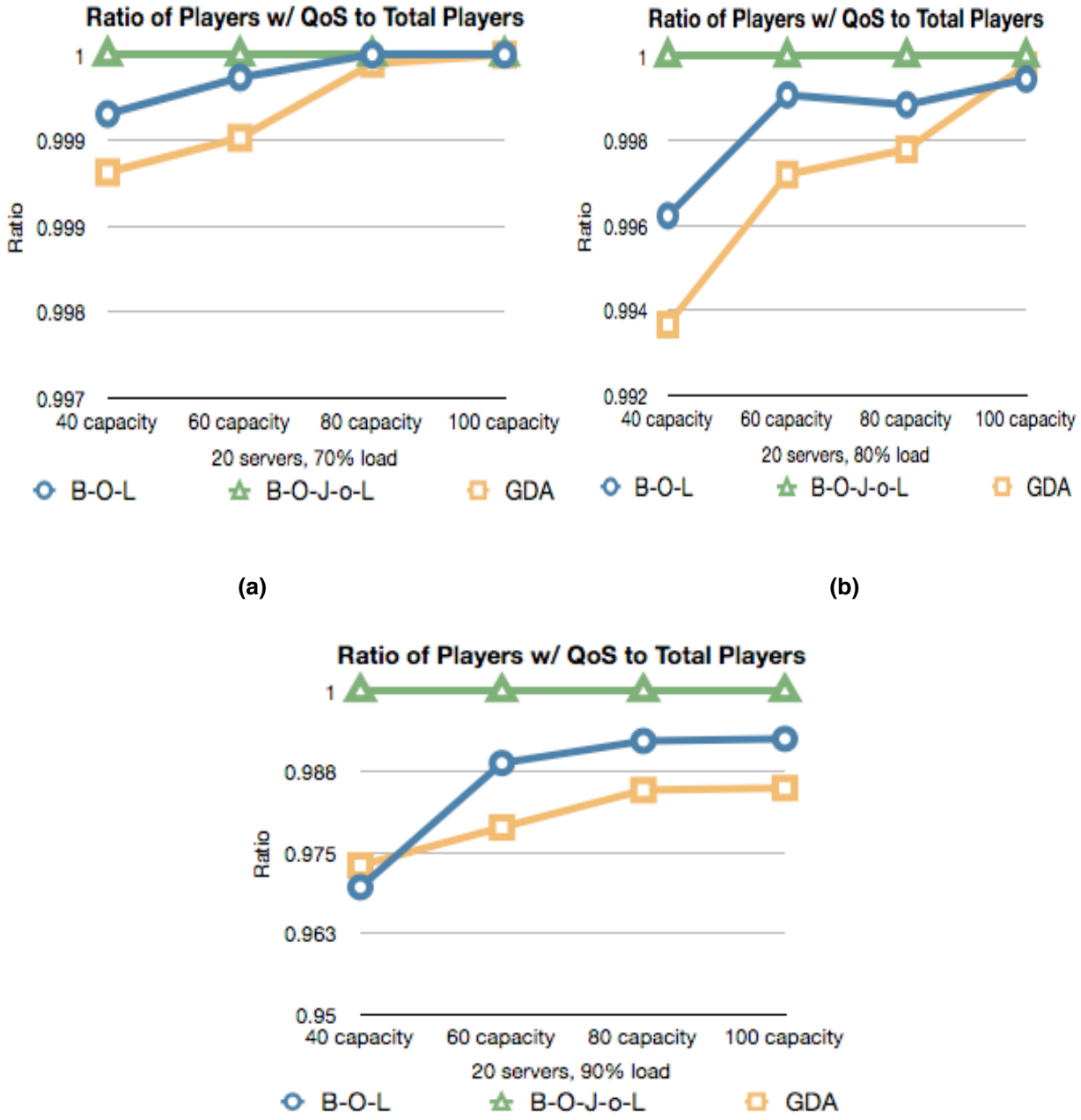


Figure 5.2. Ratio of players leaving within QoS (from initial assignment or movement during play) to total players that have left, for twenty servers at loads of (a) 70%, (b) 80%, and (c) 90%.

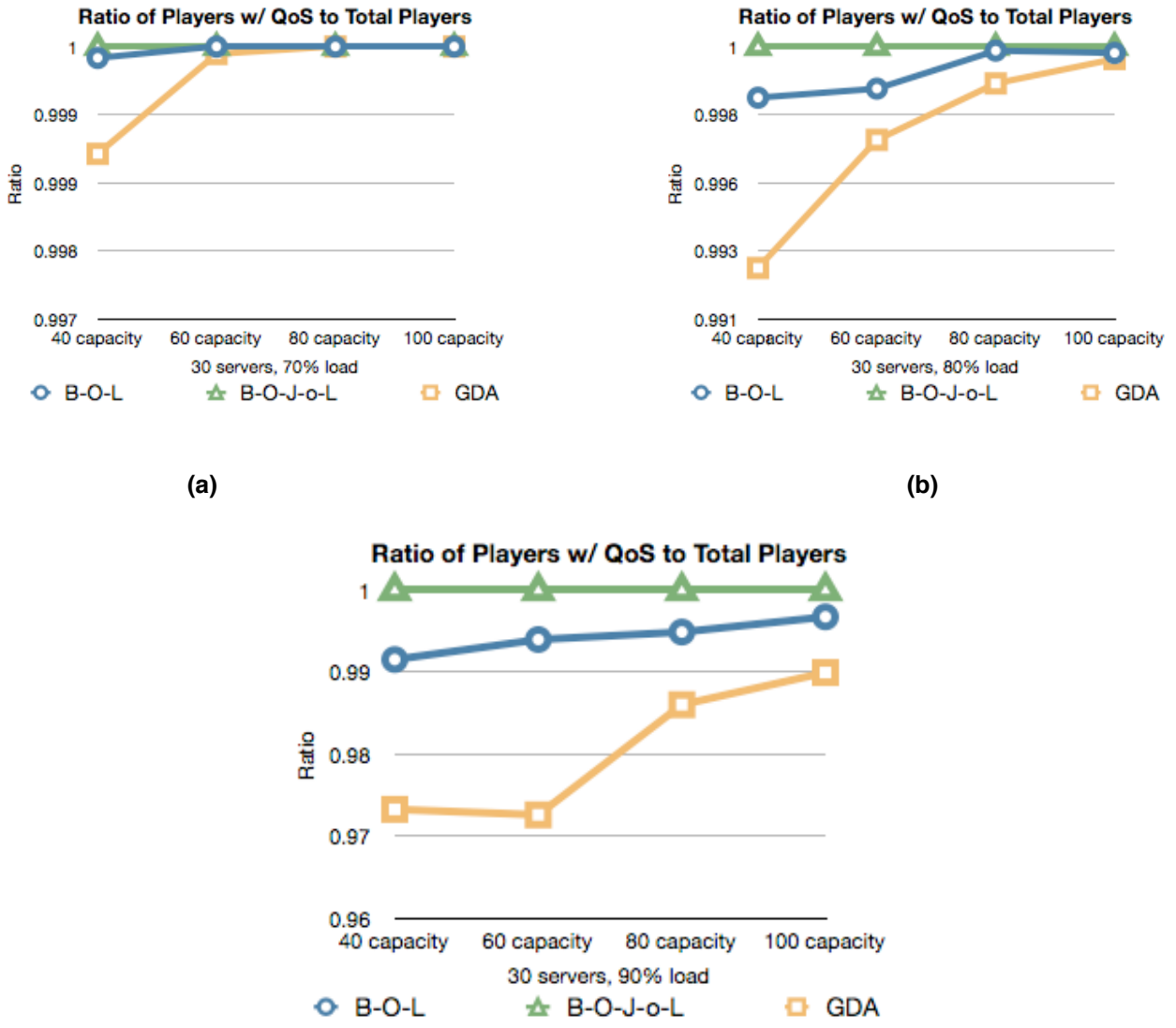


Figure 5.3. Ratio of players leaving within QoS (from initial assignment or movement during play) to total players that have left, for thirty servers at loads of (a) 70%, (b) 80%, and (c) 90%.

The next few figures (5.6 – 5.8) show that as the server capacity increases, the work done by each algorithm approaches zero, apart from initial assignment. This is also true as the number of servers increases. Once the number of servers increases to forty, less than one half of one percent of players leaving within QoS come from the work of the algorithms, regardless of server load. The ranges in which the algorithms

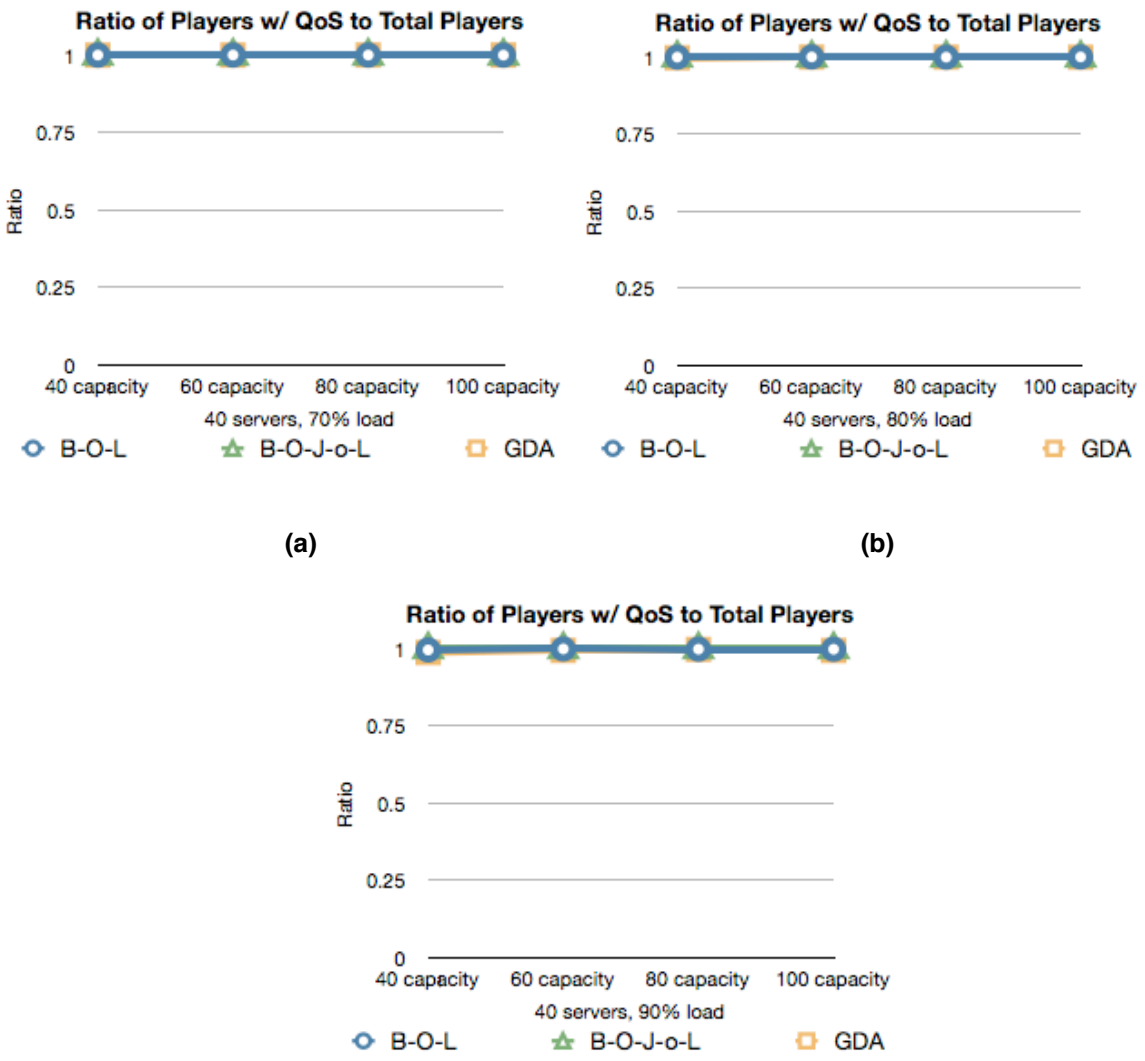


Figure 5.4. Ratio of players leaving within QoS (from initial assignment or movement during play) to total players that have left, for forty servers at loads of (a) 70%, (b) 80%, and (c) 90%.

have the greatest impact, for the given default values, is an environment of twenty to thirty servers, with forty to sixty capacity each. In the other cases, most players are assigned within QoS initially, so a static assignment is good enough to produce good results, as shown by the GDA simulations in Figures 5.1-5.4.

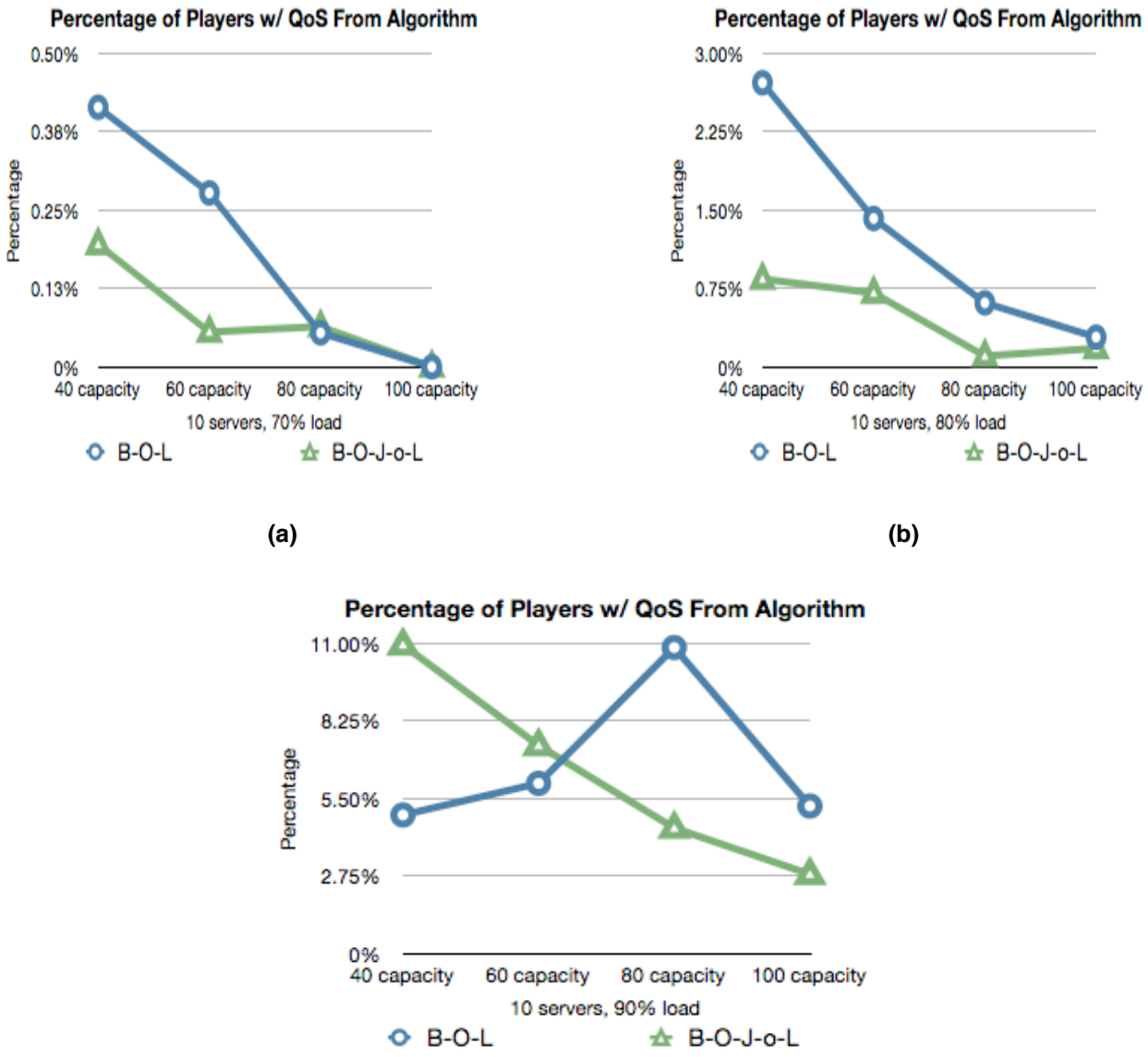


Figure 5.5. Percentage of players leaving within QoS, moved into QoS by B-O-L or B-O-J-o-L, for ten servers at loads of (a) 70%, (b) 80%, and (c) 90%.

Figures 5.9 and 5.10 show results for ten and twenty servers at ninety percent load with the value of QoS increased to 150 from 100. Performance in these scenarios is similar to the previous ones, so these values of QoS appear to cause little change in results.

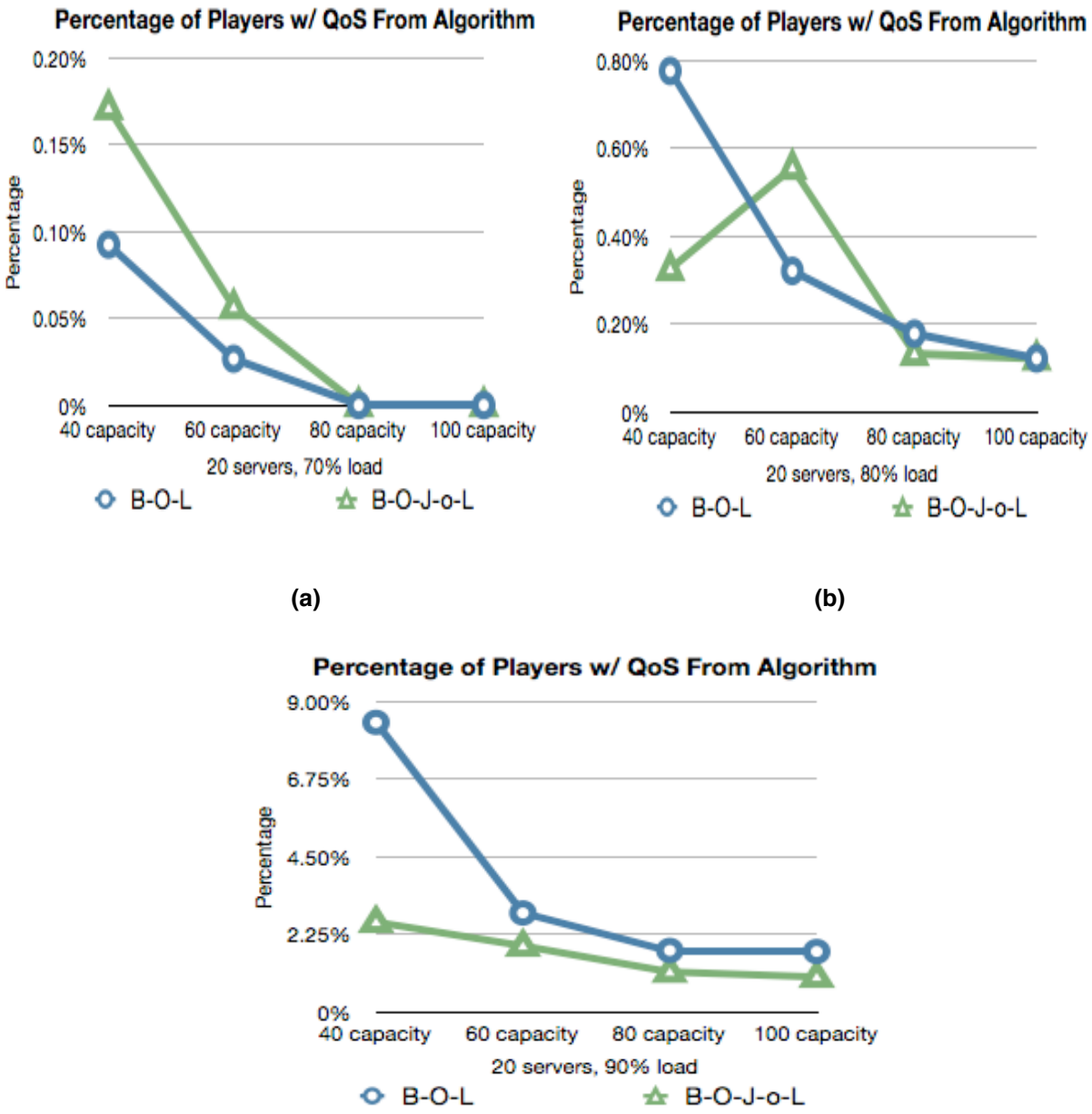
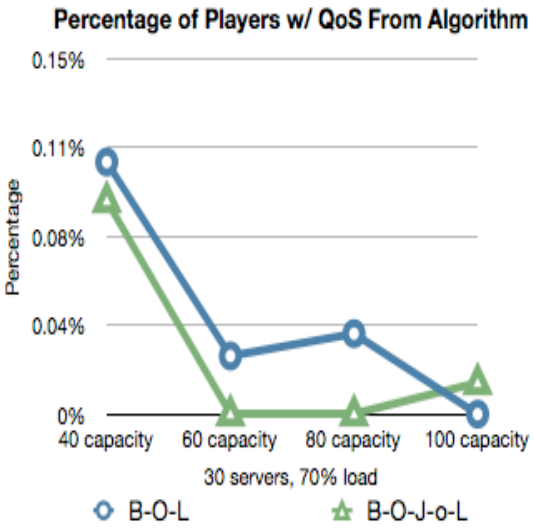
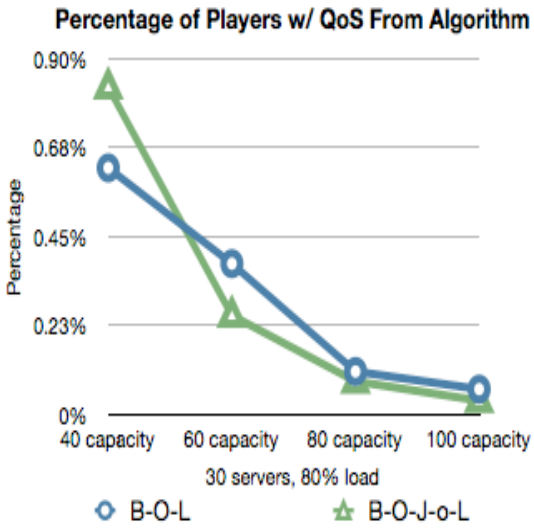


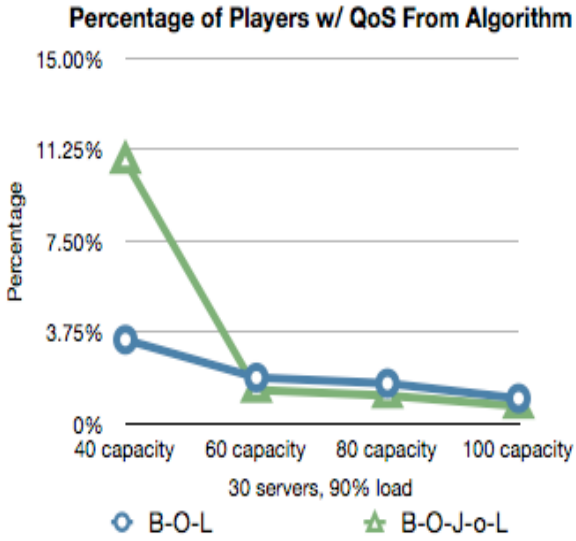
Figure 5.6. Percentage of players leaving within QoS, moved into QoS by B-O-L or B-O-J-o-L, for twenty servers at loads of (a) 70%, (b) 80%, and (c) 90%.



(a)

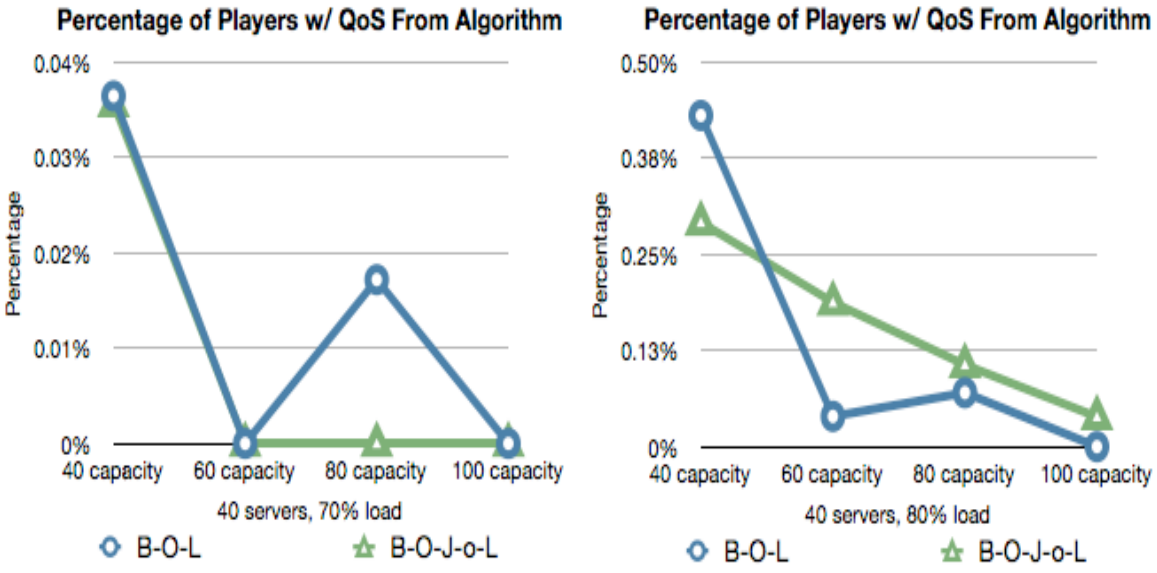


(b)

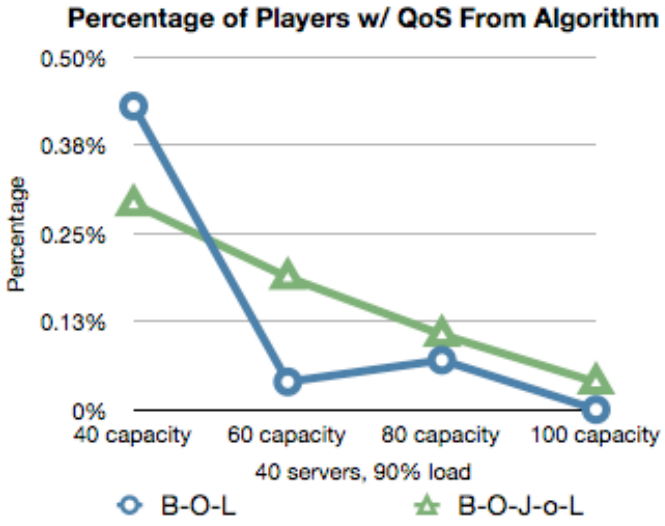


(c)

Figure 5.7. Percentage of players leaving within QoS, moved into QoS by B-O-L or B-O-J-o-L, for thirty servers at loads of (a) 70%, (b) 80%, and (c) 90%.

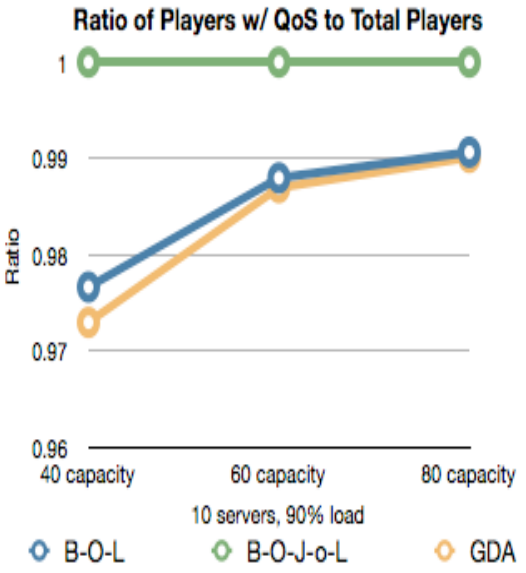


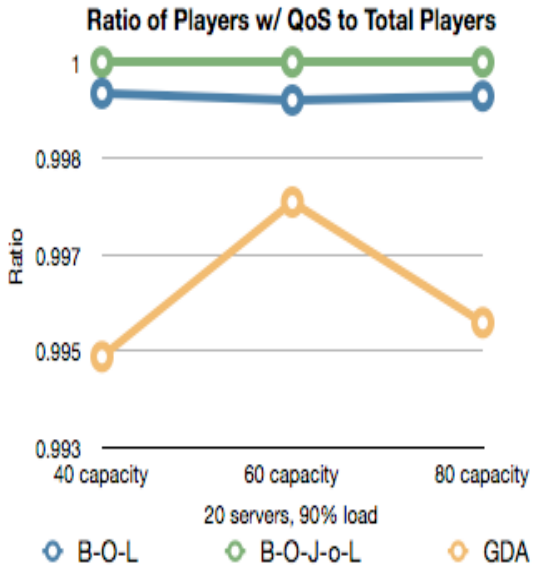
(a) (b)



(c)

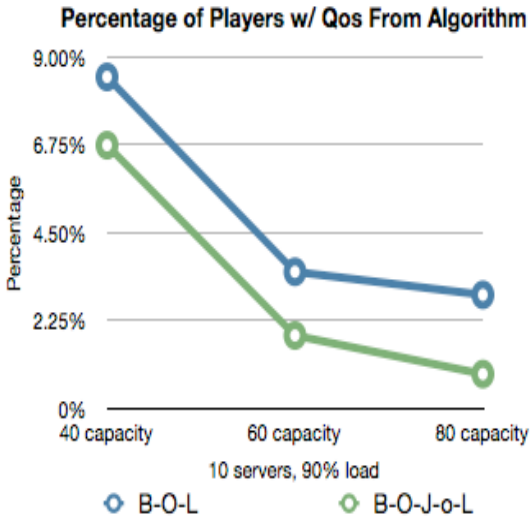
Figure 5.8. Percentage of players leaving within QoS, moved into QoS by B-O-L or B-O-J-o-L, for forty servers at loads of (a) 70%, (b) 80%, and (c) 90%.





(a) (b)

Figure 5.9. Ratio of players leaving within QoS (from initial assignment or movement during play) to total players that have left, for a load of 90% on (a) 10 servers, and (b) 20 servers, QoS = 150



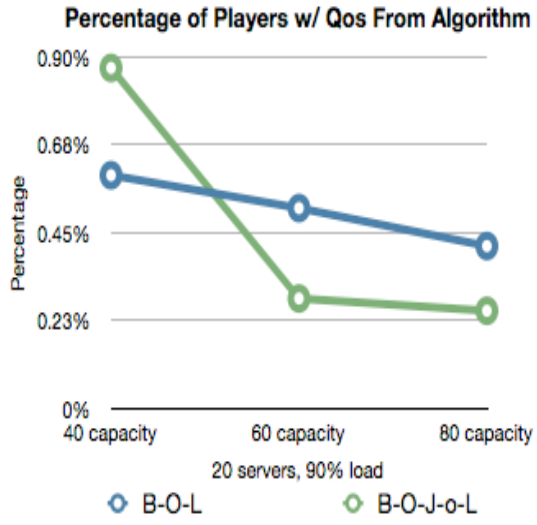


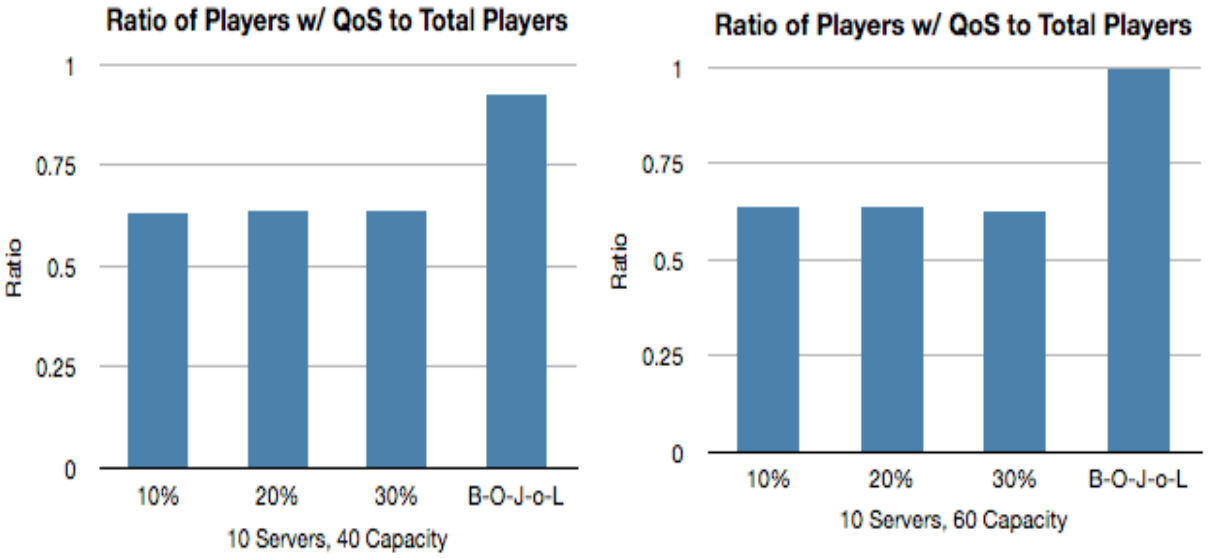
Figure 5.10. Percentage of players leaving within QoS, moved into QoS by B-O-L or B-O-J-o-L, for a load of 90% on (a) 10 servers, and (b) 20 servers, QoS = 150

5.3 Simulation Results – REASSIGNMENT THRESHOLD and B-O-J-o-L

Now we present the results of the REASSIGNMENT THRESHOLD algorithm simulation. For this simulation, the QoS value is also 100 and the delay range remains 0-500. However, we do not operate at a prespecified server load because the threshold of players without QoS is not crossed since there is a steady join and leave rate. Therefore, the join rate is increased to two potential joins per cycle, while keeping the leave rate at one potential leave per cycle. More players will join than leave, so there is a better chance of crossing the threshold of players without QoS, triggering a refine operation. Also, in an attempt to reduce unnecessary refinements, the simulator loads servers to thirty percent before allowing algorithm operation. Previous simulations have shown that few, if any, algorithm operations take place before achieving seventy percent load, so this is an acceptable value. The average number of players joining and leaving per simulation is fifteen hundred to two thousand.

Figure 5.11 shows a comparison of the REASSIGNMENT THRESHOLD algorithm (with refinement threshold values of ten, twenty and thirty percent) with the B-O-J-o-L algorithm, in an environment of ten servers. Note that for these comparisons (in this figure and all other REASSIGNMENT THRESHOLD figures), B-O-J-o-L is run with the same join and leave rate as REASSIGNMENT THRESHOLD, and all other variables are the same unless otherwise noted. The figures indicate that with so few servers, threshold value has little bearing on the ratio of players with QoS, in part for the same reasons as previously simulated using B-O-L and B-O-J-o-L – there are fewer servers, so there are fewer alternative servers to which players can be moved. B-O-J-o-L performs nearly as

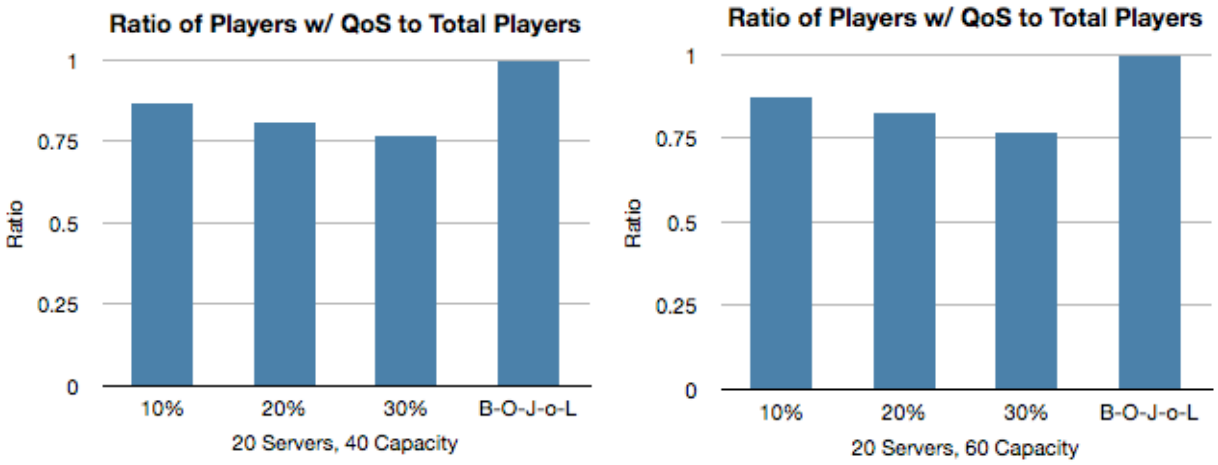
well as in the previous simulations, moving nearly all players into QoS before they leave.



(a) **(b)**

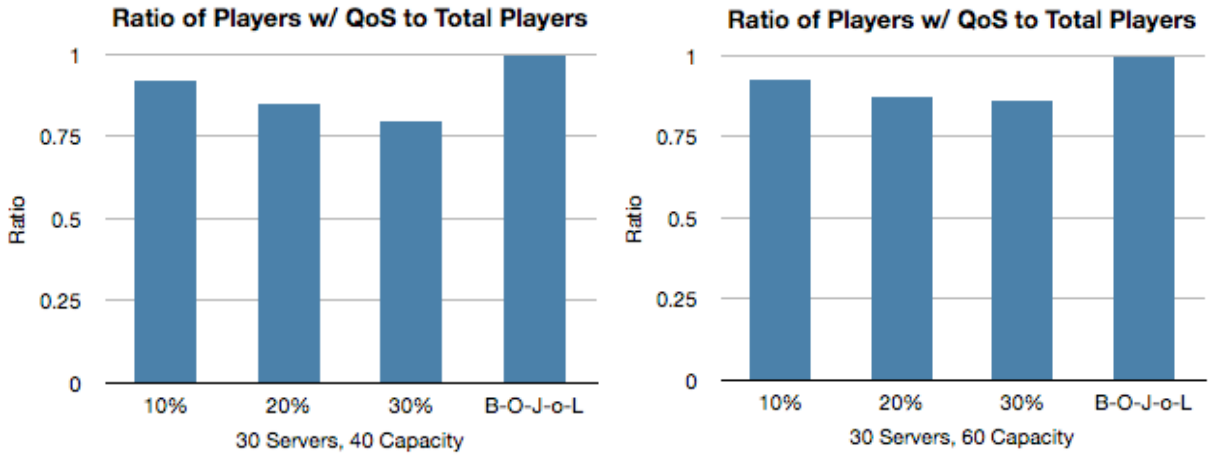
Figure 5.11. Ratio of players leaving w/ QoS (from either initial assignment or moved by algorithm) to total players that have left, for refinement threshold values of ten, twenty and thirty percent using the REASSIGNMENT THRESHOLD algorithm (indicated by the percentages on the graph) and also for B-O-J-o-L using the same join/leave rate as REASSIGNMENT THRESHOLD, for server capacities of (a) forty and (b) sixty, using ten servers.

Figures 5.12, 5.13 and 5.14 show the same comparisons as in 5.11 for an increasing number of servers and the same two values of server capacity (forty and sixty). Note that as the number of servers increase, so does the ratio of players leaving with QoS. For each of these scenarios, B-O-J-o-L assigns each player within QoS by the time he or she leaves.



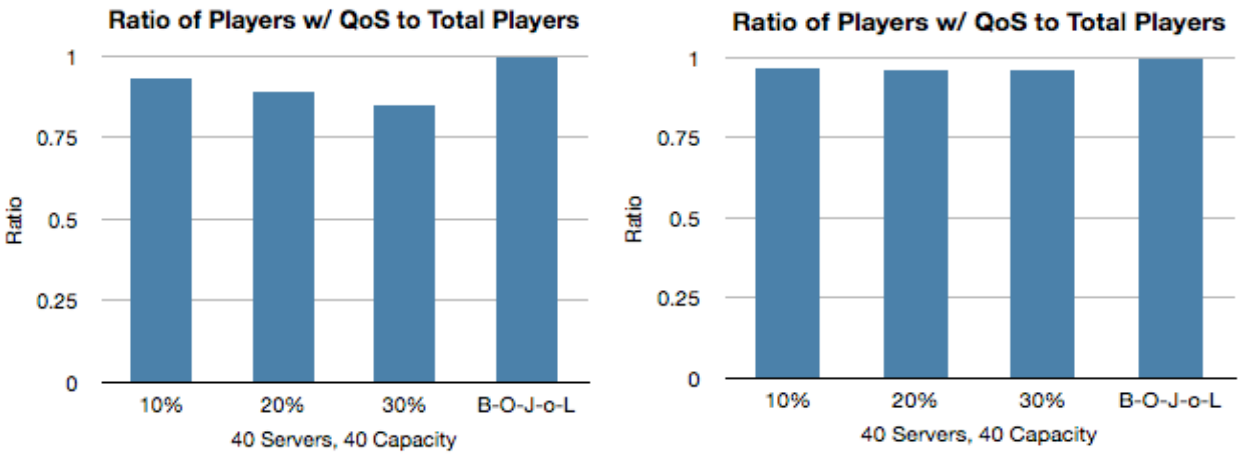
(a) (b)

Figure 5.12. Ratio of players leaving w/ QoS (from either initial assignment or moved by algorithm) to total players that have left, for refinement threshold values of ten, twenty and thirty percent using the REASSIGNMENT THRESHOLD algorithm (indicated by the percentages on the graph) and also for B-O-J-o-L using the same join/leave rate as REASSIGNMENT THRESHOLD, for server capacities of (a) forty and (b) sixty, using twenty servers.



(a) (b)

Figure 5.13. Ratio of players leaving w/ QoS (from either initial assignment or moved by algorithm) to total players that have left, for refinement threshold values of ten, twenty and thirty percent using the REASSIGNMENT THRESHOLD algorithm (indicated by the percentages on the graph) and also for B-O-J-o-L using the same join/leave rate as REASSIGNMENT THRESHOLD, for server capacities of (a) forty and (b) sixty, using thirty servers.



(a) **(b)**

Figure 5.14. Ratio of players leaving w/ QoS (from either initial assignment or moved by algorithm) to total players that have left, for refinement threshold values of ten, twenty and thirty percent using the REASSIGNMENT THRESHOLD algorithm (indicated by the percentages on the graph) and also for B-O-J-o-L using the same join/leave rate as REASSIGNMENT THRESHOLD, for server capacities of (a) forty and (b) sixty, using forty servers.

The next figures compare the percentage of players leaving with QoS that are moved there by an algorithm operation – a refinement in the case of the REASSIGNMENT THRESHOLD algorithm, or a join/leave bump in the case of B-O-J-o-L. A graph accompanies each of these, showing the average number of players moved into QoS per algorithm operation, to give context to how many players are moved into QoS total and how efficient each operation is. The number of total operations is also included to further show how efficient (or inefficient) each algorithm is in each scenario. In each graph, B-O-J-o-L moves one player per operation, since it does not bump chains of players.

Figure 5.15 shows these comparisons for an environment of ten servers with server capacities of forty and sixty servers. The number on each bar indicates the total

number of algorithm operations performed, while the y axis of the graphs on the right indicates the number of players moved into QoS for each of these operations. An "algorithm operation" means a refine call for REASSIGNMENT THRESHOLD or a join or leave operation for B-O-J-o-L. For this number of servers, refinement is called a much larger number of times than for a larger number of servers, regardless of threshold value, and does little movement each operation. This large number of refine calls comes from the small overall capacity. Since total capacity over all servers is small, and the refinement threshold is defined as a percentage of total players without QoS, the algorithms calls refinement after only a few non-QoS joins. Fewer players are moved into QoS per operation since there are fewer alternative servers to which to move non-QoS players. From these results, the REASSIGNMENT THRESHOLD algorithm does not appear to be a good option for these values and an environment of ten or fewer servers.

Figure 5.16 shows the same comparisons as 5.15, but for an environment of twenty servers. For this number of servers, the percentage of players put into QoS by an algorithm operation is similar to the percentage in an environment of ten servers, but there is a large difference in the number of operations performed. There are still many refinement calls at a threshold of ten percent, for the same reasons as in the ten-server scenario. This result indicates that even though there is some effort in the algorithm to prevent unnecessary refinement calls, we require more effort to prevent this for a lower refinement threshold value. Apart from that, as the threshold increases, the number of players moved per operation increases (although the total number of players moved into QoS decreases). REASSIGNMENT THRESHOLD performs fewer overall algorithm

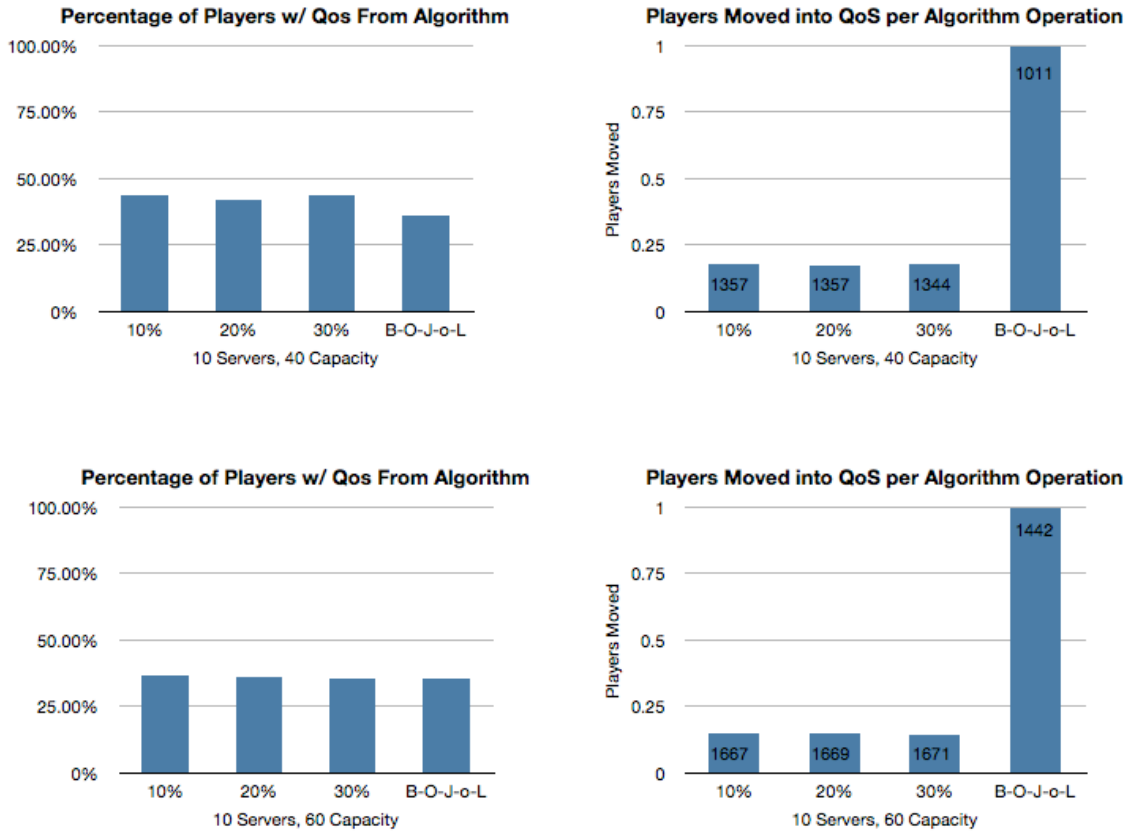


Figure 5.15. Percentage of players leaving with QoS moved there from an algorithm operation (refinement or join/leave), paired with graph of players moved into QoS per algorithm operation, for refinement threshold values of ten, twenty, and thirty percent (REASSIGNMENT THRESHOLD algorithm), and B-O-J-o-L, for server capacities of forty and sixty in an ten server environment. Number in each bar indicates total algorithm operations performed.

operations than B-O-J-o-L, but moves fewer players into QoS. There is a tradeoff between operations performed and number of players moved into QoS, which confirms what we set out to do when designing the REASSIGNMENT THRESHOLD algorithm – reduce operations performed while achieving performance near to the other algorithms.

Figures 5.17 and 5.18 show results for scenarios of thirty servers and forty servers, respectively. For this number of servers, the REASSIGNMENT THRESHOLD algorithm moves fewer overall players into QoS, but increases the number of players

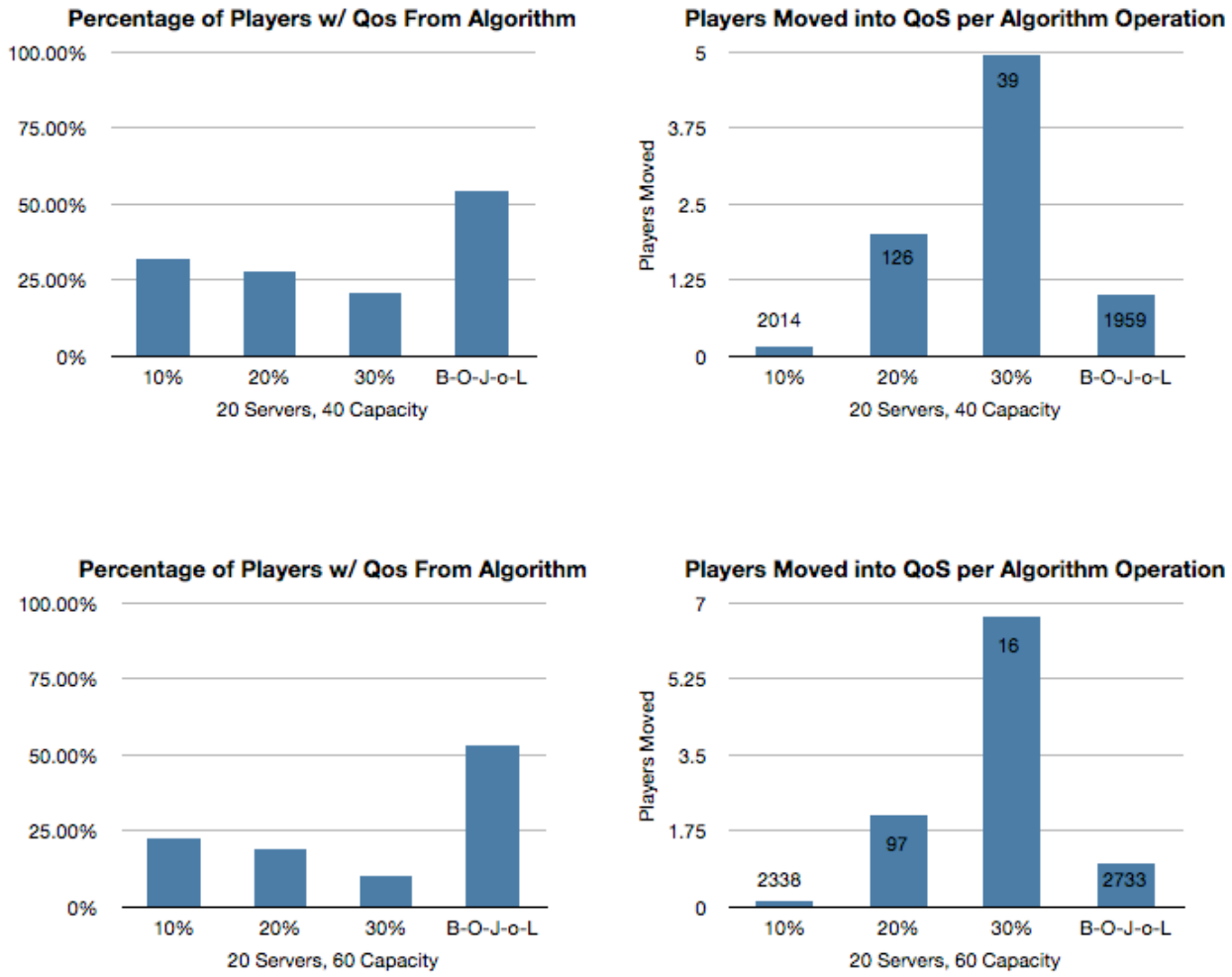


Figure 5.16. Percentage of players leaving with QoS moved there from an algorithm operation (refinement or join/leave), paired with graph of players moved into QoS per algorithm operation, for refinement threshold values of ten, twenty, and thirty percent (REASSIGNMENT THRESHOLD algorithm), and B-O-J-o-L, for server capacities of forty and sixty in an twenty server environment. Number in each bar indicates total algorithm operations performed.

moved per operation. B-O-J-o-L performs similarly to the previous environments. Once we reach a thirty percent threshold for thirty servers with sixty capacity, refinement is not called during the course of simulation. The forty server, sixty capacity scenario is not shown here at all because refinement is never called regardless of refinement threshold value. With so many choices of server to connect to there are few players

without QoS to begin with, let alone enough to exceed the refinement threshold value (except for very low values).

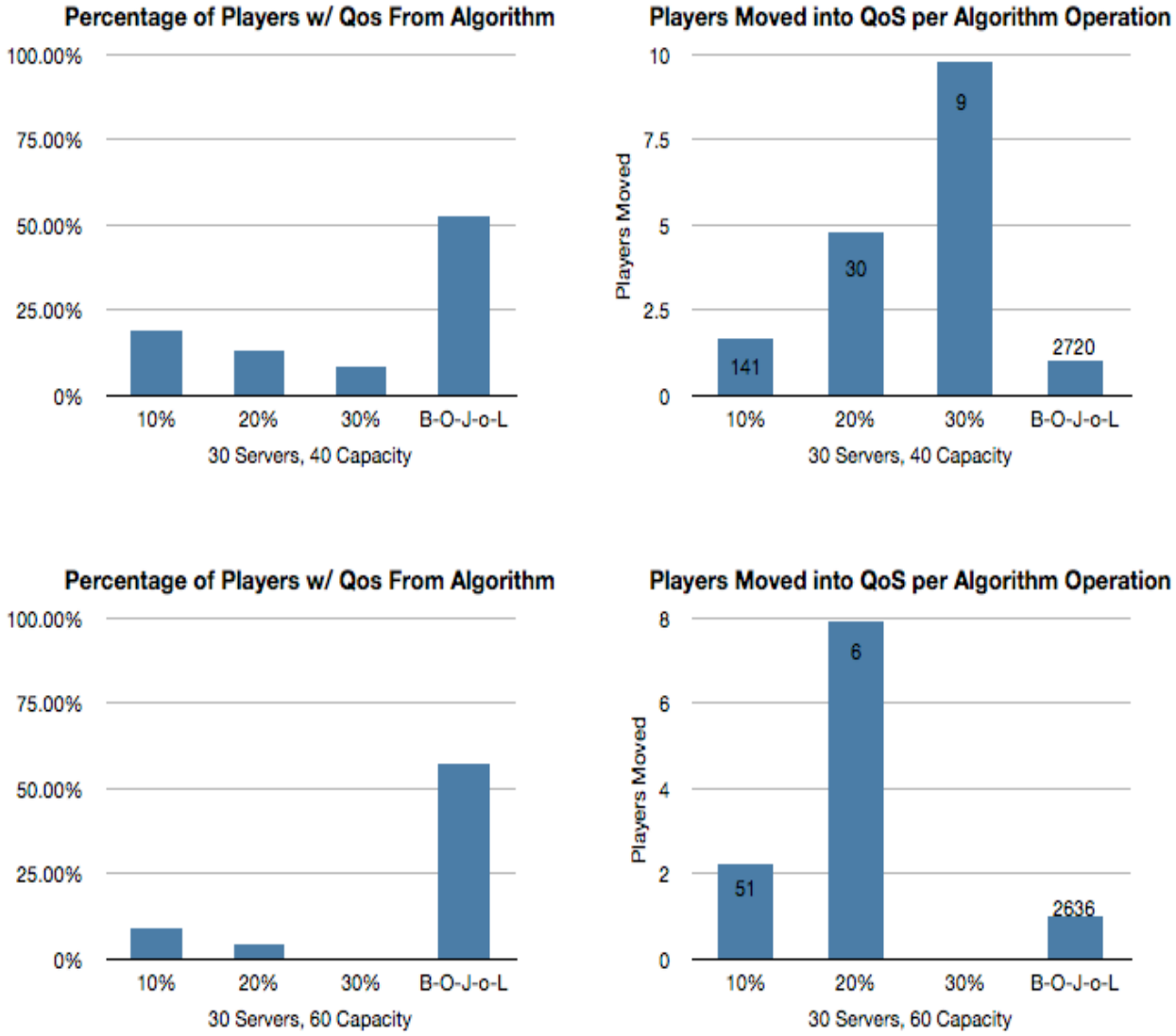


Figure 5.17. Percentage of players leaving with QoS moved there from an algorithm operation (refinement or join/leave), paired with graph of players moved into QoS per algorithm operation, for refinement threshold values of ten, twenty, and thirty percent (REASSIGNMENT THRESHOLD algorithm), and B-O-J-o-L, for server capacities of forty and sixty in a thirty server environment. Number in each bar indicates total algorithm operations performed.

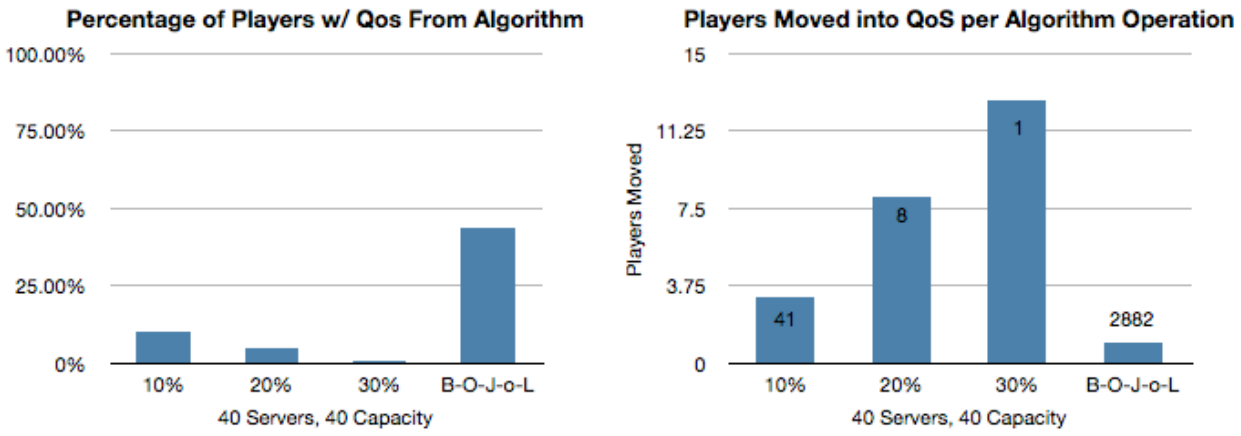


Figure 5.18. Percentage of players leaving with QoS moved there from an algorithm operation (refinement or join/leave), paired with graph of players moved into QoS per algorithm operation, for refinement threshold values of ten, twenty, and thirty percent (REASSIGNMENT THRESHOLD algorithm), and B-O-J-o-L, for a server capacity of forty in a forty server environment. Number in each bar indicates total algorithm operations performed. Capacity of sixty is not shown because refinement is never called.

Overall, there is less of a clear choice of which algorithm should be used – one does not consistently outperform the other one in all areas. We must make the decision whether to reduce the overall number of algorithm operations or increase the number of players within QoS. For the purposes of this paper, our goal is to increase the number of players within QoS, so our choice is made for us, but in other operating environments there may be some merit to reducing the number of algorithm operations.

6. Conclusion

We have shown, over the course of this document, that the solution to the CAP proposed by Ta *et al.* has some areas where player assignment can be improved, specifically after the initial assignment. Opportunities exist to move players into QoS after the initial assignment and we have proposed three distinct algorithms to take advantage of these opportunities. We have shown that for certain situations, two of these algorithms assign more players within QoS than Ta *et al.*'s. Although the third algorithm could not be directly compared to Ta *et al.*'s in simulation, it accomplishes its goal of reducing the number of overall algorithm operations compared to the other two algorithms and displays merit for use in other environments where improving the overall number of players within QoS is not the primary motive.

The research here brings up several considerations for future work. Simulation shows that in many cases, the chain of bumps performed by the B-O-L algorithm gives it performance near or better than the single bump method used by the B-O-J-o-L algorithm, even though the B-O-L algorithm does not bump on joins. The next natural step is to apply the chain of bumps to the B-O-J-o-L algorithm. Next, the REASSIGNMENT THRESHOLD algorithm needs further improvement so that the refinement function is not called repeatedly for low refinement threshold values. We would also like to measure the execution time of the algorithm to determine if the added time cost of algorithm processing is worth the improvement in player assignment, as well as other, more detailed performance measures (such as measuring the number of players within QoS at any particular moment of simulation rather than on leave only) to further prove or

disprove the merit of each algorithm. Lastly, we would like to simulate the algorithms in a real environment of clients and servers to see how they perform and modify them as needed to work beyond the limits of the simulator. With these improvements, we could not only better judge the value of the algorithms, but use those findings to make them more generally applicable and better performing for a wider variety of operating environments.

References

[ESRB] *How Much Do You Know About Video Games?.* Retrieved October 2011 from http://www.esrb.org/about/video-game-industry-statistics.jsp

[ARM2] G. Armitage. “Optimising Online FPS Game Server Discovery through Clustering Servers by Origin Autonomous System.” *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2008, pp. 3-8.

[BES5] P. Beskow, K. Vik, P. Halvorsen, C. Griwodz. “The Partial Migration of Game State and Dynamic Server Selection to Reduce Latency.” *Multimedia Tools and Applications*, Volume 45, Issue 1-3, October 2009, pp. 83-107.

[BRI3] L. Briceño, H. Siegel, A. Maciejewski, Y. Hong, B. Lock, M. Teli, F. Wedyan, C. Panaccione, C. Klumph, K. Willman, C. Zhang. “Robust Resource Allocation in a Massive Multiplayer Online Gaming Environment.” *Proceedings of the 4th International Conference on Foundations of Digital Games*, 2009, pp. 232-239.

[CHE6] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, C. Amza. “Locality Aware Dynamic Load Management for Massively Multiplayer Games.” *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005, pp. 289-300.

[CHE8] Y. Chen, S. Radhakrishnan, S. Dhall, S.Karabuk. “Server Selection with Delay Constraints for Online Games.” *Proceedings of GLOBECOM Workshops*, 2010, pp. 882-887.

[CHE9] Y. Chen, S. Radhakrishnan, S. Dhall, S.Karabuk. “On the Game Server Network Selection with Delay and Delay Variation Constraints.” *Proceedings of the Third International Conference on Communication Systems and Networks (COMSNETS)*, 2011, pp. 1-10.

[CF7] R. Chertov, S. Fahmy. “Optimistic Load Balancing in a Distributed Virtual Environment.” *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2006, pp. 1-6.

[CLAY2] M. Claypool, K. Claypool. “Latency and Players Actions in Online Games.” *Communications of the ACM*, Volume 45, Number 11, November 2006, pp. 40-45.

[MIRRSERV] E. Cronin, A. R. Kurc, B. Filstrup, and S. Jamin, “An Efficient Synchronization Mechanism for Mirrored Game Architectures.” Multimedia Tools and Applications, vol. 23 (1), 2004, pp. 7-30.

[GAR10] S. Gargolinski, C. St. Pierre, M. Claypool. “Game Server Selection for Multiple Players.” *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, 2005, pp. 1-6.

[LEE11] K. Lee, B. Ko, S. Calo. “Adaptive Server Selection for Large Scale Interactive Online Games.” *Computer Networks*, Volume 49, Issue 1, September 2005, pp. 84-102.

[TAZHOU1] D. Ta, S. Zhou. “A Network-centric Approach to Enhancing the Interactivity for Large-Scale Distributed Virtual Environments.” *Computer Communications Journal*, vol. 29, no. 17, 2006, pp. 3553-3566.

[TAZHOU2] D. Ta, S. Zhou. “A Two-phase Approach to Interactivity Enhancement for Large-Scale Distributed Virtual Environments.” *Computer Networks*, vol. 51, no. 14, 2007, pp. 4131-4152.

[TAETAL] D. Ta, S. Zhou, H. Shen. “Greedy Algorithms for Client Assignment in Large-Scale Distributed Virtual Environments.” *Simulation*, Volume 84, 2008, pp. 521-533.

[WEBBETAL2] S. D. Webb and S. Soh. “Adaptive Client to Mirrored-Server Assignment for Massively Multiplayer Online Games." *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN)*, 2008, paper 68180I.

[WEBBETAL] S. Webb, S. Soh, J. Trahan. “A Mirrored Referee Scheme for Anti-Cheat P2P Network Games”. Manuscript.

[ZHANGTANG] L. Zhang, X. Tang. “Client Assignment for Improving Interactivity in Distributed Interactive Applications.” *Proceedings of IEEE INFOCOM*, 2011, 9 pp.

Vita

Shawn Farlow was born in Metairie, Louisiana. He moved to Baton Rouge in 2000 to attend Louisiana State University, where he earned a Bachelor of Science degree in electrical engineering in 2004. After leaving Baton Rouge in 2005, he returned to Louisiana State University to pursue a Master of Science degree in electrical engineering in 2009. After obtaining the Master of Science degree, Shawn plans to continue his research and pursue a Doctor of Philosophy in electrical engineering at Louisiana State University.