6-28-2018

Heuristics for Client Assignment and Load Balancing Problems in Online Games

Shawn Michael Farlow
Louisiana State University and Agricultural and Mechanical College, shawn.m.farlow@gmail.com

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations
Part of the Computer Engineering Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_dissertations/4655

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
HEURISTICS FOR CLIENT ASSIGNMENT AND LOAD BALANCING PROBLEMS IN ONLINE GAMES

A Dissertation

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
Doctor of Philosophy

in

School of Electrical Engineering and Computer Science
Division of Electrical & Computer Engineering

by
Shawn Michael Farlow
B.S., Louisiana State University, 2004
M.S., Louisiana State University, 2011
August 2018
This dissertation is dedicated to my wife Melissa and my daughter Mikaela, who supported me and helped to continually motivate me for its duration.
Acknowledgements

I would like to express my appreciation and thanks to Dr. Jerry L. Trahan for his time, help, and dedication to ensuring my continued success throughout the course of my work on my Master’s and Ph.D degrees at LSU. Without his support, motivation, and accommodation this work would not have been possible. He ensured I was able to complete this work while maintaining a busy family and work life outside of school. I would also like to thank the committee members, Dr. Jianhua Chen, Dr. Suresh Rai, Dr. Ramachandran Vaidyanathan, and Dr. Shannon Walsh for serving on my dissertation committee and their support of my efforts and work as a Ph.D candidate at LSU.
# Table of Contents

Acknowledgements .................................................................................................................. iii

Abstract ................................................................................................................................... v

1. Introduction ............................................................................................................................... 1

2. Offline CAP-Z .......................................................................................................................... 6
   2.1 Introduction ......................................................................................................................... 6
   2.2 Offline Cap-Z Literature Review ......................................................................................... 7
   2.3 Offline Cap-Z Background ................................................................................................. 17
   2.4 Greedy Bin Packing ........................................................................................................... 21
   2.5 Simulation .......................................................................................................................... 39

3. Centralized Load Balancing ................................................................................................... 93
   3.1 Load Balancing Introduction ............................................................................................. 93
   3.2 Centralized/Distributed Load Balancing Literature Review .............................................. 94
   3.3 Centralized Load Balancing Background .......................................................................... 103
   3.4 Centralized Load Balancing Method .................................................................................. 107
   3.5 Centralized Load Balancing Simulation ............................................................................ 132

4. Distributed Load Balancing .................................................................................................. 164
   4.1 Distributed Load Balancing Introduction and Background ................................................. 164
   4.2 Distributed Load Balancing Method .................................................................................. 166
   4.3 Distributed Load Balancing Simulation ............................................................................ 172

5. Future Work ............................................................................................................................ 204

6. Conclusion ............................................................................................................................... 206

Works Cited .................................................................................................................................. 207

Vita .............................................................................................................................................. 212
Abstract

Massively multiplayer online games (MMOGs) have been very popular over the past decade. The infrastructure necessary to support a large number of players simultaneously playing these games raises interesting problems to solve. Since the computations involved in solving those problems need to be done while the game is being played, they should not be so expensive that they cause any noticeable slowdown, as this would lead to a poor player perception of the game. Many of the problems in MMOGs are NP-Hard or NP-Complete, therefore we must develop heuristics for those problems without negatively affecting the player experience as a result of excessive computation. In this dissertation, we focus on a few of the problems encountered in MMOGs – the Client Assignment Problem (CAP) and both centralized and distributed load balancing – and develop heuristics for each. For the CAP we investigate how best to assign players to servers while meeting several conditions for satisfactory play, while in load balancing we investigate how best to distribute load among game servers subject to several criteria. In particular, we develop three heuristics - a heuristic for a variant of the CAP called Offline CAP-Z, a heuristic for centralized load balancing called BreakpointLB, and a heuristic for distributed load balancing called PLGR. We develop a simulator to simulate the operations of an MMOG and implement our heuristics to measure performance against adapted heuristics from the literature. We find that in many cases we are able to produce better results than those adapted heuristics, showing promise for implementation into production environments. Further, we believe that these ideas could also be easily adapted to the numerous other problems to solve in MMOGs, and they merit further consideration and augmentation for future research.
1. Introduction

As the industry of computing has grown since its inception, the number of applications for its use has been abundant over the years. Whereas the computer may have been useful to only hobbyists or those in the industry, it has now turned into an item found in nearly every household and the crux of almost all successful businesses [7]. It is difficult to imagine getting through one’s day without the use of a computing device – we ask our smart home device to turn on the lights and tell us the weather in the morning, use our smartphone to check traffic on the way to work and listen to music while traveling, work on our personal computer to complete our duties at our jobs, and watch TV on our streaming device to relax at the end of the day. As these devices have become more commonplace, so has the use of these devices to play games. At its inception, many computer-based games required a dedicated gaming console to enjoy, such as the Intellivision [11], CalecoVision [12] and Atari [13] systems. Since then, the realms of gaming devices and computing devices have merged, and many of the devices we use primarily for other functions include a gaming facet. Personal computers, smartphones, and streaming devices include the ability to play games, and the aforementioned gaming consoles are still alive and well also. This level of accessibility has lead to a boom in the popularity of gaming as a whole, and the increase of ease to Internet access has made online gaming as much a part of our daily lives as the computing devices on which it takes place.

Online gaming has been particularly popular in the past decade. Whereas the industry was once previously isolated to groups of few players whose machines were
physically connected in the same location, and may have required intimate knowledge of computing and protocols, now people of all ages, backgrounds, and levels of technical expertise can not only play games, but enjoy them with friends and family both near and far. Gaming online can be as simple as competitive scoring on a leaderboard with other players across the globe, involve playing in a live tournament one-on-one, or playing online with dozens or hundreds of other players at once in the comfort of one’s own home. The global media and entertainment market itself has been consistently on the rise, with the worldwide market growing from nearly 1.75 trillion dollars in 2015 to a projected 2.2 trillion dollars by 2021 [41]. Mobile gaming and massively multiplayer online games, or MMOGs, account for nearly 20 billion dollars of that annual total. In creating a front-end experience for users of all backgrounds using multiple devices like computers, gaming consoles, tablets, and smart phones, there exist many interesting challenges and problems behind the scenes to investigate and develop solutions at multiple layers. In particular, we are interested in MMOGs and creating solutions for some of the problems of access and interaction among players.

In MMOGs [48], multiple physical servers host an online game. Each server may be responsible for a unique portion of the game’s virtual world and communicate with other servers only as needed by the game, or alternatively each may contain a full copy of the virtual world and communicate any updates it receives with each of the other servers so that the world is uniform among the servers. A player connects to one of these servers in order to play the game – he or she may have the ability to choose the server to which he or she connects, or the game may choose the server. The implemented method of connecting the player to the server can pose its own unique set of problems and challenges, and the
player should be unaware of the computational issues taking place to connect him or her to the game. Additionally, these choices and computations should not affect the user's ability to play the game. A user’s experience should be as seamless as possible with no visible effect on the game regardless of the choices he or she makes in the game – a game that does not simply “just work” can lead to a frustrating experience and ultimately to the user choosing not to play that game any longer.

Two questions in particular (and problems associated with each) we will investigate in the online gaming realms are:

1) How does a player get connected satisfactorily to a game when joining?

2) How does a player remain satisfactorily connected throughout the course of the game while playing?

The issues we will examine in particular are the Client Assignment Problem and two flavors of load balancing in MMOGs called centralized load balancing and distributed load balancing. We define the Client Assignment Problem (CAP) and develop heuristics for a particular version of the CAP called Offline CAP-Z, in which we assign zones, or partitions of the game world, to servers. We loosely base these heuristics, named GBP, GBP-Skip, and GBP-Skip+Fill, on a greedy approach to the well-known Bin Packing problem [14]. A primary goal of our heuristics is to perform as well or nearly as well as existing heuristics in the literature but with shorter runtime. We show that for many cases our heuristics succeed in this goal.

On load balancing, we propose a periodic approach to load balancing in which we perform load balancing operations during defined breakpoints. This avoids unnecessary computation in response to movement or action by players. For centralized load balancing,
where we have access to global system knowledge, we develop heuristics that balance server load within a defined range called load balancing factor, or LBF. Our basic unit of load is the cost of a zone of players, and that load is not infinitely divisible in our problem as it is in some other load balancing problems. The use of LBF helps define balance in a system where uniform load among all servers is not possible because of this load constraint. Our heuristic, called BreakpointLB, adds and sheds load from the top and bottom ends of the server load gap to reduce the load range to within LBF, and supplements this with proactive and reactive zone splitting operations to improve performance in tough situations. To our knowledge, there do not exist other solutions for this problem, so we adapt an existing heuristic in the literature for a problem close to ours to compare and measure performance. We show that our heuristic brings more breakpoints into balance than this adapted approach.

For distributed load balancing, in which each server has access to its own local knowledge only and not global knowledge, we again balance load within LBF using periodic breakpoint operations. Since servers do not have access to global information to balance load, they perform pairwise operations with neighbor servers in parallel to reduce load gap across the system in an attempt to bring each server within LBF of each of its neighbors. We anticipate that if we bring each server within LBF of each of its neighbors, then this will reduce the global load even though we do not have access to this global information to balance load. Again, to our knowledge, no work exists toward this problem, so we adapt an existing heuristic in the literature for a problem similar to ours. We show that our heuristic brings more breakpoints into balance than this adapted approach.
As the arena of online gaming continues to flourish and migrate into other realms, such as cloud gaming [18], undoubtedly more aspects will open up and introduce interesting problems to research and solve, and we anticipate that this will be a relevant area of research for years to come.

The remainder of this dissertation is outlined as follows: Chapter 2 discusses the Client Assignment Problem, in particular a variant called Offline CAP-Z; Chapter 3 discusses centralized load balancing; Chapter 4 discusses distributed load balancing; Chapter 5 discusses future work; and Chapter 6 concludes the dissertation.
2. Offline CAP-Z

2.1 Introduction

We examine the *Client Assignment Problem* (CAP) in MMOGs in which we are tasked with assigning game zones to servers subject to several criteria so that at least a defined fraction of players in each zone has satisfactory connections to the game. When a player connects to a server, the server incurs some computational load. Each server has a maximum capacity for such a load. If load exceeds this capacity, it degrades system performance, so we must take care to not make assignments that will overload any server. In addition, the time to communicate between player and server, or latency, must not be higher than a prespecified threshold. If latency exceeds this value, then the player experience suffers. The name for this threshold is *quality of service*, or QoS. If we can make an assignment such that the player latency is less than QoS, then we consider that player to be *satisfied*.

The CAP has several variants that make assignments of players to servers subject to criteria like server load and QoS. One variant, CAP-C, requires heuristics that assign individual players (clients) to servers subject to QoS and server capacity. Satisfaction in CAP-C happens at a per-player level. One example of a satisfied CAP-C system is if the fraction of player-to-server delays within QoS lies above a minimum threshold. Another variant of CAP that we investigate in greater detail is CAP-Z. In CAP-Z successful zone-to-server assignments are subject to not only QoS and server load, but also a value called pQoSZ. Each player falls into a game world partition called a *zone*. Instead of assigning players to servers directly, we assign zones of players at once. Value pQoSZ specifies what fraction of players in each zone must be satisfied by a server to consider the zone as
satisfied. The goal in CAP-Z is to find a minimum-sized subset of all servers that satisfies every zone in the system, and if one does not exist, then to find the subset that satisfies the greatest number of zones, all while respecting server capacity constraints. In particular, the CAP-Z problem we investigate is Offline CAP-Z, in which we make all assignments before gameplay begins.

Other work on problems similar to Offline CAP-Z [44] has proposed novel heuristics—however, these heuristics are for an uncapacitated version of the problem, in which servers have infinite capacity. We consider this assumption to be unrealistic, and so develop heuristics that take server capacity into account. We construct three heuristics for offline CAP-Z based on bin packing. The first, GBP, proceeds in stages with more computationally expensive operations in the later stages where fewer zones remain to process. The second, GBP-Fill, adapts GBP by filling servers upon activation. The third, GBP-Skip+Fill, identifies hard-to-satisfy zones and skips them in early stages, waiting until later stages to consider interserver connections. In the next section, we provide a review of existing literature for the landscape of the CAP and related facets. Section 2.3 discusses background for Offline CAP-Z, Section 2.4 outlines our method for this problem and our three heuristics, and Section 2.5 presents simulation results.

2.2 Offline Cap-Z Literature Review

Ta et al. [43] proposed the NP-hard zone mapping problem, assigning zones to servers in order to reduce the number of clients without QoS. They asserted that most research in this area has been formulated as a load balancing problem, seeking to evenly distribute the workload among servers, but not with the goal of reducing latency as they
do. As in our research, they partition the virtual world into zones to allocate load among servers.

The authors proposed new zone mapping heuristics to achieve better results than a greedy approach might obtain. They based these heuristics on local search and genetic algorithms, called LS-Mapping and GA-Mapping, respectively. Unlike our work which randomly generates delays between clients and servers, the authors used real Internet delay data to evaluate their heuristics.

Although some facets of the zone mapping problem are similar to our work, the authors’ approach based on linear programming algorithms is too slow for our purposes. While time is not a constraint offline, we still seek a solution that is faster than a “traditional” offline approach while providing an assignment with nearly as many or as many zones within QoS.

Ta *et al.* [42] continued to work with an extension of their zone mapping problem [43] called the interactivity-constrained server provisioning problem (ICSP). They sought to achieve a pre-specified fraction of zones or players with QoS using the minimum number of servers. The authors formulated two variations of the problem, showing they are both NP-hard by reducing to the set cover problem.

The authors proposed a heuristic they called Greedy for each flavor of their problem – one makes zone-to-server assignments within QoS and the other makes client-to-server assignments within QoS. They compared these heuristics with a randomized assignment, an optimal assignment, and an implementation of a solution to the set cover problem. When QoS requirements are strict, their Greedy heuristic tends to perform best of the four in terms of runtime to fraction of zones or clients within QoS – it achieves assignments with
a QoS fraction close to that of an optimal assignment in faster time, and better QoS fraction than set cover or randomized approaches.

One interesting facet is that the authors simulated “inaccurate” input data, where the exact latencies are unknown during assignment, and instead the latency used is within a prespecified range of the actual latency. It would be an interesting extension of our work to see how the system responds to this data.

This paper and its findings are the basis for much of our research and Section 2.5.1 discusses it in more detail.

Ta et al. [44] worked again on their proposed zone mapping problem, however here they looked at other facets – the cost of assigning a zone to a server and the overhead involved in zone migration. They developed two algorithms, LPLS and HMOEA, which are based on local search and genetic algorithms, respectively, like those they previously developed [43]. Unlike our problem, clients always connect directly to the target server, and there is no intermediate contact server.

In this problem, given an initial assignment of zones to servers, the authors sought to increase the number of clients within QoS, while refraining from moving zones from their initial assigned server if possible. To help choose assignments with few zone-to-server reassignments, they introduced a new parameter - migration ratio - which represents the cost of moving a zone to a new server. They calculated this by normalizing the total number of clients in zones that need to be migrated with total number of clients in the system. An assignment that minimizes this cost is the most desirable.

The first algorithm they implemented, LPLS, utilizes LP relaxation to get an initial solution that is improved upon using local search. The second, HMOEA, uses a genetic
algorithm to solve the problem. A gene in a chromosome represents a zone, and the position of the gene represents the zone index. The algorithm uses standard single point crossover, standard mutation and binary tournament selector operations, and uses local search to improve offspring. They ultimately discovered that HMOEA offers little to no benefit over LPLS. As in their previous work [43], the approaches the authors used have slower runtimes than we seek in our approaches.

Beskow et al. [2] considered dynamically partitioning the game world into zones based on player distribution in an attempt to lower the overall latency of interacting players. There are three key elements to the authors’ approach – zone assignment, a distributed name service, and latency estimation.

Zone assignment consists of choosing a server to which to assign a zone such that all players interacting in the zone can send each other updates within some latency constraint. The algorithm, given the set of players in a zone as an input, will choose the server for which the sum of latencies of shortest paths to those players is the smallest. This assignment tries to ensure that there is not a server that could provide the players in a zone with lower delay. One server can end up hosting all zones, whereas in our work each server has a finite capacity.

Zone assignment saves time and processing by measuring a subset of links and then estimating the latency of the rest of the links based on those measurements. Beskow et al. used two techniques - 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, which uses a set of landmark nodes to
estimate others’ relative network position. Vivaldi uses multidimensional-scaling based estimation, which uses statistical comparisons of data to make its estimates.

Nguyen et al. [38] investigated assignment of audio mixing operations to a geographically distributed set of servers, intending to provide an immersive audio experience by broadcasting voice communications within range of a player’s avatar. Although we do not deal with audio mixing operations, the problem they defined generally corresponds to players in a game world and strategically assigning them to servers. The authors focused on a distributed server architecture, partitioning the game world into zones and assigning computation for all avatars in a zone to a server, much like we do in our problem. Their goal was to find the optimal game world partition into zones and to choose a server assignment to reduce the sum of delays for all players. Their model allows for a server to host multiple zones, but each zone is hosted by a single server. They model network and server topology as one graph with two disjoint sets of vertices representing each of the two. The authors used a greedy heuristic that has comparable performance and faster runtime than an optimal solution. Although the authors acknowledged that an online environment where players move, join and leave throughout would change the optimal assignment, they assumed these assignments to take place offline. They implemented both a uniform distribution of players as well as a clustered distribution (positioned around the center according a normal distribution with mean of zero).

One unique aspect of the authors’ research is the study of avatar group patterns. The authors identified three groups – loners, clans, and crowds. Loners are single players who travel alone, while clans and crowds are groups of 30 or more and 100 or more players, respectively, that are clustered together at a point of interest in the game world. The
authors identified a tendency for people to group together due to language, culture, and/or lifestyle preferences or commonalities. In terms of simulation, they modeled this behavior with a correlation parameter, allowing for adjustment of “closeness” among players. Interestingly, they proposed the use of a hybrid architecture instead of distributed server. This would implement a peer-to-peer architecture for areas of loners and a distributed server architecture for points of interest where crowds tend to congregate.

Although some aspects of the authors’ work falls within our offline CAP-Z problem, reducing the sum of all delays is not one of them – this might be more applicable to a CAP-C problem. Notably, they reported faster runtimes for their algorithm than some other approaches we have observed (the authors’ algorithm runs in 500 ms or less for fewer than 100 zones), which is important for us as we seek faster runtimes in an offline environment as well.

Zhang and Tang [49][50][51][52] performed their work on continuous distributed interactive applications, or DIAs, that change states not only in response to user operations but also to passing of time. This translates to dynamic client joins and leaves and players moving around the game world, as in an MMOG.

Zhang and Tang modeled the Client Assignment Problem as a combinational optimization problem, modeling the network as a graph. The graph has $V$ nodes (representing clients and servers) and the distance function between each pair of nodes represents latency. They defined the maximum length of the path of data between any two players as the interaction path. The path between any two players $P_1$ and $P_2$ is the path from $P_1$ to $P_1$’s server $S_1$, from $S_1$ to $P_2$’s server $S_2$ (if the players connect to different servers), and finally from $S_2$ to $P_2$. (Note that information may need to pass through
intermediary servers between $S_1$ and $S_2$ if they are not directly connected.) Their goal was to minimize the lengths of interaction paths among all client pairs.

The authors presented three heuristic algorithms – nearest-server assignment, greedy assignment, and distributed-modify assignment. These algorithms do not inherently consider that servers have limited capacity, but the authors make proposals and considerations for a problem that is capacitated.

Nearest-server assigns each client to its nearest server, or server with lowest latency. This approach reduces individual client-server latencies, but could potentially increase client-to-client delays depending on the number of servers and server connection speeds on the intermediate path between those. The authors sought to reduce such delays so this outcome is not desirable.

Greedy assignment is iterative, with each step considering all server possibilities. After choosing a server for a particular client, the approach also assigns all other unassigned clients with equal or less latency to that server. This does not increase the maximum delay between any pair of these unassigned clients because it assigns them to the same server. The heuristic calculates for each unassigned client ($a$) the increase in the maximum client-pair delay were it to assign that client (and thus, all other unassigned clients with equal or less latency) to a particular server; ($b$) the number of clients that would be reassigned to that server, that is, the number of unassigned clients with latency less than or equal to the chosen client. It chooses the client to assign by the client-server pair with smallest fraction $a/b$.

The distributed-modify heuristic performs distributed operations without requiring global knowledge at each server. It elects one server among all as a coordinator for
calculating the maximum length interaction path, which it uses to select servers along the way. The coordinator selects a server involved in this path and notifies it to attempt to help reduce the path length by modifying assignment of its client involved in that path.

The authors found experimentally that the greedy and distributed-modify approaches provide shorter maximum interaction paths than nearest-server. They also found that the distributed-modify approach adapts to dynamic client joins and leaves best of the three.

Chertov and Fahmy [10] investigated an adaptive partitioning architecture that partitions virtual space into zones. Each server handles a rectangular zone that can grow or shrink in size to accommodate a varying number of players. Their system 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. The servers attempt to keep the number of players handled by each server balanced as players move to different game regions. Zones 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. Their environment sets capacity limits on servers and can dynamically reassign players during the course of gameplay as needs change due to moving players. The authors’ environment was online CAP-C and so does not apply to our work here, however it gives precedence to partitioning game worlds into zones as we do in CAP-Z.

Bortnikov et al. [5] created two heuristics for two flavors of the Client Assignment Problem they referred to as the Load Distance Balancing Problem (LDB). In the Min-Max LDB, the focus is to minimize the maximum incurred delay by a client on a server. In the
Min-Average LDB, the focus is on minimizing the average delay in the system. They showed some approximation algorithms and optimal algorithms for their problems. Unlike our work in CAP, they reduced either average or maximum delay, while we focus only on achieving delay within QoS. Additionally they fixed the number of servers, and we attempt to reduce the number of servers used in the assignment. The authors’ work gives further merit to working on the Client Assignment Problem and shows other metrics to tailor heuristics and solutions.

Armitage [1] improved 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 probed server to send back a reply is known as the round trip time (RTT). In an online game such as *Counterstrike: Global Offensive* (which boasts over one hundred 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.

While the author’s approach and findings do not directly correspond to an offline CAP-Z environment, they provide some justification for development of server selection methods in multiple problems, including offline CAP-Z.

Briceño et al. [6] developed heuristics for an offline CAP and also developed a mathematical model for MMOGs. Their heuristic minimizes the maximum delay among all users. Their heuristic is different than ours in that it assumes a static assignment is sustainable over time, while we believe that assignments need to change as players move throughout the game. Additionally, they permit their execution time to be five minutes or greater, which is well beyond what we attempt to achieve in our offline work. Despite this, this work still provides additional justification for work on the Client Assignment Problem, including offline variants.

Li and Cai [32] extended their previous work [31] to apply some of the principles they formulated there to a client assignment problem. In this paper, they sought to assign zones and clients to servers that minimizes a time-space inconsistency metric by formulating the problem into a mixed-integer programming problem. They defined the time-space inconsistency metric as the difference between the player’s perceived location in the game and the player’s actual location server-side, subject to delays imposed between the client and server. This is an example of a client assignment problem that minimizes delay rather than assigning subject to QoS.
2.3 Offline Cap-Z Background

2.3.1 Terms

Here we introduce some terms that we will use throughout the paper. A client is a potential player of an MMOG, along with his or her medium for connecting to the game (a PC, typically). “Client” and “player” are interchangeable terms here. An avatar refers to the player’s character in the game. A server is a dedicated resource that hosts the game application and routes information among the various clients connected to it, as well as to other servers that also host clients for the same game application. A zone is a virtual partition of the game “world” and at most one physical server may host a zone.

Clients and servers exist in the “physical world” outside of whatever game application the servers are hosting. Zones exist in the “virtual world”, that is, inside of the aforementioned game application. All clients whose avatars are in a zone in the game connect directly or indirectly in the physical world to the server responsible for hosting that zone. Clients separated by large physical distance in the physical world may be relatively close inside of the virtual world, or vice versa.

We allow for clients to connect either directly or indirectly to a server hosting a zone, so we name the servers involved to distinguish between these types of connections. The server through which a client connects to the game is the contact server. The target server hosts the zone containing the player’s avatar in the virtual world and handles all the processing for the client (such as sending move and action updates by the client to other players). The contact server routes all information sent between the client and the target server. Note that a player’s contact server and target server may be the same physical server.
We define the round-trip connection time between a player and a server to be the delay or latency. Contact delay is the latency from the player to his or her contact server and back, interserver delay is the delay from contact server to target server and back, and total delay is the sum of contact delay and interserver delay. If the contact and target servers are the same, then total delay is the same as contact delay. We will use the term delay to mean total delay throughout this paper. (Delay implicitly accounts for processing time at the target server.)

QoS (short for quality of service) is a prespecified player delay threshold. If a player’s delay is within this threshold, then he or she achieves QoS and is satisfied. For example, if the QoS value is 150 ms and a client’s delay is 125 ms, then that combination of contact and target server achieves QoS for that client, and were that target server to host the client’s zone then the client would be satisfied. In cases where we define QoS at the zone level instead of the player level, another value, $pQoSZ$, indicates what fraction of players in each zone must be satisfied to consider that zone satisfied. For example, if $pQoSZ$ is 0.9, then we must satisfy 90 percent of the players in a zone to consider that zone satisfied. We use the term “satisfied” for both clients and zones, however note that for clients this indicates client-server delay within QoS, while for zones it indicates an acceptable fraction of satisfied clients. Server capacity is the computational load, or cost, a server can accommodate. This load consists primarily of processing updates sent to the server by clients and communication of those updates to players or other servers. The cost is a constant value for each client connected to a server. If the players are in zones, then the cost of each zone is the sum of the costs of players in that zone. If a server exceeds its
capacity, the server becomes *overloaded*, and performance degrades or the server shuts down completely.

### 2.3.2 Zones

Two common schemes exist for servers handling zones. In the first, we partition the game world into static zones, such that we can always handle one zone within the capacity bound of one server. Further, it may be that the load of many zones fits within the capacity of one server. In the second scheme, each active server handles one zone, but the boundaries of each zone expand and contract according to the number of players in the zone. We assume the first scheme. Both schemes may handle a small, densely populated area of the virtual world with many servers, while handling a larger, sparsely populated area with one server.

As players move throughout the virtual world, their assigned target servers change as their zone changes. With one server responsible for hosting a zone, and allowing for more than one zone to be assigned to a single server, as more players enter a zone the computational load required to process game updates among the players in that zone can grow large quickly. Eventually, the server can become overloaded, causing a detrimental effect to player experience.

### 2.3.3 Client Assignment Problem

In the client assignment problem (CAP), we begin with a set of clients, a set of servers with their capacities (server load must always be less than or equal to its capacity), player-server delays and server-server delays, and the value for QoS, then we assign clients to servers subject to constraints. The constraints can be on client parameters and/or organization, the server parameters, the requirements for a successful assignment, or a
combination of any of these. We can also examine the problem offline or online – this indicates whether the application is actively running while we make our assignments. Let us discuss these constraints in further detail.

The constraints we choose to implement and the way we define each can change the basic CAP in many different ways. One constraint is that we can make the assignments in the CAP relative to clients (CAP-C) or zones (CAP-Z). In CAP-C, we attempt to make player-to-server assignments that achieve QoS for each player. We do not consider zones in CAP-C. When measuring performance, the number of players with QoS is what is important – player delays give us a basis for assignment, but we do not seek to make assignments with least delay or improve any player assignments made within QoS. We instead attempt to make assignments such that a minimum fraction of players in the system achieves QoS, given by an input called \( pQoSC \). In CAP-Z, the input also includes sets of zones, the clients in those zones, and instead of \( pQoSC \), a value called \( pQoSZ \). We attempt to make zone-to-server assignments such that a minimum fraction (given by \( pQoSZ \)) of the players in each zone achieves QoS. As long as we exceed this fraction with our assignment, our requirements are met, even if a better assignment exists that will satisfy a larger fraction of players in the zone.

In online client assignment, or online CAP [47], we can either begin with no prior assignments as in offline, or we may start with some given assignment of clients to servers and attempt to improve them. The game is also running during this time, so players may join and leave over time. This requires us to accommodate for dynamic changes in environment, such as players moving among zones if examining CAP-Z. The computations performed by a client-to-server assignment happen during the course of the game and
incur some load on the servers doing the processing, which can affect player performance. Therefore heuristics that perform fast, adequate client-to-server assignments are preferable to thorough or exhaustive methods that provide slower, better assignments. In an offline client assignment, or offline CAP, we assume the MMOG is not running. We do not begin the game until the assignments are complete, so heuristics that require more computation time to make better assignments are acceptable. Once we have finished making these assignments, the game begins.

In addition to providing constraints on the organization of our clients (individual or in zones) and the operating environment (offline or online), we can change the requirements for successful assignment of a player or zone to a server. For example, Bortnikov et al. [5] attempted to minimize the maximum delay of clients and minimize the average delay among all clients. This differs from the CAP-C described above in that it makes assignments to improve the delay directly, not the number of clients within a delay threshold. We present this as an example of how adding or changing a few parameters can create an entirely new problem that will call for new heuristics.

We next present our motivation and method for three heuristics for approximating a solution to offline CAP-Z named GBP, GBP-Skip, and GBP-Skip+Fill.

2.4 Greedy Bin Packing

2.4.1 Motivation

We propose a new offline CAP-Z heuristic that we will refer to as Greedy Bin Packing (GBP). This heuristic operates much like a greedy approach to the Bin Packing Problem [10]. In the Bin Packing problem, given a collection of objects each with varying cost and bins with finite capacity, we “pack” the objects into the bins with a goal of using the
minimum number of bins. A greedy first-fit decreasing approach sorts and processes the objects in decreasing order of cost, places each object into the first bin that can accommodate it, and opens a new bin if no opened bin can accommodate the object.

Adapting this to our heuristic, we consider the bins to be servers with finite capacity. The objects are the game zones and their costs are the load incurred on the server that will host each. Our general approach processes the zones in decreasing order of load and, for each zone, finds the first server that can satisfy and accommodate it, activating a new server if necessary. The goal of GBP is to produce a server assignment in offline MCAP-Z in less time than Ta et al.’s Greedy-Z [44], which we describe in more detail in Section 2.5.2, with the same number or close to the same number of servers that Greedy-Z does.

GBP operates in stages of increasing complexity, each on a smaller subset of zones. Only the final stage has the same complexity as Greedy-Z, but over fewer zones in GBP than in Greedy-Z.

In an offline CAP, heuristics perform assignments and calculations before the start of the application and players do not move from zone to zone during the course of assignment (as they might in an online version). All information is known at the beginning of the heuristic – server locations, player locations, player assignments to zones, player delays to each server, etc. If assignments use direct connections – the contact and target servers are the same – then we can take advantage of this knowledge and use it to pre-calculate for which zones each server satisfies pQoSZ. Greedy-Z does not perform or take advantage of this pre-calculation because it also considers indirect connections - its zone-to-server assignments can use separate contact and target servers and take into account interserver delays among servers in SL’, the in-progress list of selected servers. This list SL’
cannot be known until the heuristic begins its work and starts to choose servers to add to $SL'$. The first stage of GBP considers only direct connections, and so it performs this pre-calculation for efficiency – the calculations are performed once and then the results are reused as necessary. We do not include the runtime of this phase in the reported runtime of our heuristic during simulations, but have found experimentally that it ranges between 3 ms and 10ms depending on the number of servers. While this dominates the execution time of the simulation, the total runtime is still much less than Greedy-Z, as we will show in Section 2.5.4. The next section will go into the operations of GBP in more detail.

2.4.2 GBP

2.4.2.1 Overview

GBP consists of three stages of zone-to-server assignment. The first stage operates on the full set of zones, and each subsequent stage operates on only the zones that the previous stages could not satisfy. We order the stages by increasing computational complexity so that the more computationally complex stages operate on fewer zones and only those zones not satisfied by the simpler stages.

Before Stage 1 begins, a phase we refer to as pre-calculation determines which servers satisfy each zone were we to disregard capacity constraints. In an offline environment, players will not move and zones will not change, so we can take advantage of this information when making assignments.

Stage 1 of GBP utilizes the information from pre-calculation to assign zones to servers considering only direct connections. This significantly reduces the amount of time needed to produce a final zone-to-server assignment list, since the heuristic needs only to look up whether a server satisfies a zone, then check for capacity constraints before making
zone-to-server assignments. Also, if Stage 1 assigns a zone to a server, then later stages do not consider this zone, avoiding the issue of SetCover-Z described later in Section 2.5.2.

Stages 2 and 3 use interserver connections to make assignments that were not possible using direction connections in Stage 1. Although the use of interserver connections adds computational complexity to our heuristic, our simulations also support Ta et al.’s [44] finding that interserver connections improve zone-to-server assignments (lower failure rates and more zones satisfied with fewer servers). To be less computationally expensive than Greedy-Z, we use interserver connections conditionally – only when Stage 1 fails and only on a limited combination of servers.

In Section 2.5.4, we will compare the failure fraction of GBP to the failure fraction of Stage 1 only to quantify the improvement gained with interserver connections. We will also compare the failure fraction and running time of GBP to two of Ta et al.’s heuristics. We do this to ensure that the benefit brought by Stages 2 and 3 does not come at the cost of GBP becoming as computationally expensive as Ta et al.’s heuristics.

2.4.2.2 Detailed Approach

The inputs to the GBP heuristic are an array of zones Z, an array of servers S (and the capacities of those servers), pQoS, and QoS. We sort Z in descending order of cost, and each zone $z \in Z$ contains $z.PL$, a list of players in $z$ along with their delays to each server in $S$. Each server in $S$ has the same total capacity (the heuristic can be readily adapted to handle servers of differing capacities, however).

Stage 1 assigns zones to servers using direct connections.

*Stage 1*- For each server, pre-calculate the zones for which it satisfies $pQoS$, storing this in an array of flags $ZL$ for each server. Iterate through the zones in $Z$ to find a
server for each that will satisfy and accommodate it. Let $SL$ denote the set of servers selected this way. For a given zone, first check servers in $SL$, and, if none of these can satisfy and accommodate the zone, then check servers outside $SL$. If we cannot satisfactorily assign a zone to a server directly, then mark that zone to be addressed by Stage 2, set it aside, and continue with the next zone in $Z$.

If Stage 1 has assigned each zone to a server, then GBP finishes and $SL$ contains the list of active servers. Otherwise, Stage 2 and Stage 3 implement alternative, more computationally expensive steps for any zones not yet satisfied.

Stage 2 assigns zones to servers using interserver connections among servers in $SL$.

**Stage 2** - For unsatisfied zones, attempt to satisfy them using interserver connections among the servers in $SL$ (that is, make assignments utilizing separate target and contact servers). If we cannot satisfy a zone this way, then we set it aside for Stage 3 and continue to the next marked zone. If we satisfy all zones in this stage, then the heuristic is complete.

Stage 3 assigns zones to servers using interserver connections while growing $SL$ as necessary.

**Stage 3** - For each remaining unsatisfied zone, we test one at a time each server $q$ not in $SL$ as to whether $SL \cup \{q\}$ can satisfy the zone using interserver connections. We do not consider adding more than one server at a time to $SL$ - if we cannot satisfy a zone by adding a single server to $SL$ and using interserver connections, then fail the assignment.
Each iteration of Greedy-Z performs the same steps as Stage 3, but for all zones, not just unsatisfied ones. Typically, this makes it more computationally expensive than our heuristics, as we show in section 2.4.2.4.

### 2.4.2.3 GBP Pseudocode

The pseudocode for GBP is as follows.

**Heuristic: GBP**

**Inputs**

- $S$ – array of $m$ servers = \{ $s_0, s_1, ..., s_{m-1}$\}, along with their loads (initialized to 0).
- $Z$ – array of $k$ zones = \{ $z_0, z_1, ..., z_{k-1}$\}, sorted in descending order of cost. Each zone contains $PL$, an array of all players in the zone along with their delays to each server. Assume that every zone contains at least one player.
- $pQoSZ$ – the fraction of players in a zone that must be with QoS for the zone to be considered with QoS
- $QoS$ – QoS threshold value
- $servCap$ - maximum server capacity

**Variables**

- $R = \{ r_0, r_1, ..., r_{m-1} \}$ – a copy of $S$, as a list, used for iteration purposes so we may remove servers and add them to $SL$ while maintaining the integrity of $S$.
- $ZB$ – an array of marked zones passed to Stages 2 and/or 3 (if necessary)
- $ZL$ – for each server in $S$ we calculate $ZL$, an array tracking which zones each server satisfies, and store those in $S$

**Outputs**

- $SL$ – sublist of $S$ chosen to host all zones in $Z$ and provide QoS for each. If such a sublist does not exist, the algorithm returns NULL. Newest element added is always at the head of the list.

**Method**

```plaintext
//Note that for this approach to be useful, we must perform additional maintenance
//steps, such as keeping track of which zones are assigned to which servers. However,
//since the problem definition seeks only to find a subset of all servers that will satisfy
//every zone in $Z$, we provide only those steps to streamline the pseudocode presentation.
```

Heuristic: GBP (cont.)
/begin Stage 1
//perform precalculation of satisfactory zone-server assignments
for a<0 to sizeof(Z) - 1
  for b<0 to sizeof(S) - 1
    if(checkQoS(z_a, s_b, pQoSZ, QoS) == 1)
      s_b.ZL[a] <- 1
    else s_b.ZL[a] <- 0

//copy S into R as a list
SL <- NULL
for j<0 to sizeof(Z) - 1 //iterate over all zones in zone array
  c <- 0
  x <- sizeof(R) - 1
  z_j.sat <- 0
  d <- 0
  //this while loop will be skipped if SL is empty, and the following if condition
  //will handle initialization
  while(d < sizeof(SL))
    //check to see if any server added to SL has capacity and will satisfy
    //QoS
    if(sl_d.load + z_j.cost <= servCap && sl_d.ZL[j] == 1)
      z_j.sat <- 1 //indicate zone satisfied by SL
      sl_d.load += z_j.cost
      break
    d++
    if (z_j.sat == 0) //if zone not satisfied by SL attempt to add a new server
      while(z_j.sat == 0 && c < x) //iterate over remaining servers
        if( r_c.load + z_j.cost <= servCap) && r_c.ZL[j] == 1)
          z_j.load += z_j.cost
          SL += r_c //add server to SL
          R -= r_c //remove server from future consideration
          break
        c++
    if z_j.sat == 0 //if zone is not able to be satisfied
      ZB += z_j
  if(sizeof(ZB) == 0) return SL

//end Stage 1
//Begin Stages 2 and 3
else
  ZB <- backup_op_1(SL, ZB, pQoSZ, QoS) //Stage 2 call
  if (ZB == NULL) return SL //if NULL all zones were satisfied

Heuristic: GBP (cont.)
else
    \( SL = \text{backup\_op\_2}(SL, ZB, pQoS, QoS) \) //Stage 3 call
    return \( SL \)    //will return either \( SL \) or NULL

Function: checkQoS

Inputs

\( z \) – A single zone to determine QoS for. Contains \( PL \), an array of all players in the zone along with their delays to each server.
\( s \) – A single server to check QoS for
\( pQoS \) – fraction of players for which we need to achieve QoS to satisfy a zone
\( QoS \) – QoS threshold value

Variables

\( numQoS \) – number of players in zone with QoS

Output

1 if \( s \) will achieve QoS for \( z \) based on \( pQoS \), 0 otherwise

Method

if (sizeof(\( z.PL \)) == 0) return 0

\( numQoS \leq 0 \)
for \( i \leq 0 \) to sizeof(\( z.PL \)) - 1
    if delay(\( z.PL_i \), \( s \)) \( \leq \) QoS
        \( numQoS++ \)
if(\( numQoS / \text{sizeof}(z.PL) \) \( \geq pQoS \))
    return 1
else return 0

Function: backup\_op\_1 (Stage 2)

Inputs

\( ZB = \{ zb_0, zb_1, …, zb_{n-1} \} \) – an array of zones not satisfiable by direct zone-to-server assignments. Maximum size is sizeof(\( Z \)). Each zone in \( ZB \) contains \( PL \), an array of all players in the zone along with their delays to each server.
\( SL \) – a subset of all servers \( S \) that satisfies all zones not in \( ZB \)

Heuristic: GBP (cont.)
\( pQoSZ \) – fraction of players for which we need to achieve QoS to satisfy a zone
\( QoS \) – QoS threshold value
\( servCap \) - maximum server capacity

**Variables**

\( numQoS \) – number of players in zone with QoS
\( ZB' \) - used for iteration through \( ZB \) and removal of satisfied zones

**Output**

\( ZB \) – remaining unsatisfied zones (if any); NULL otherwise

**Method**

```java
// copy ZB to ZB'
// iterate over all unsatisfied zones
for i<-0 to sizeof(ZB) - 1
  // check interserver connections among selected servers
  for b<-0 to sizeof(SL) - 1
    numQoS <- 0
    // only proceed if there is enough capacity for the zone
    if(sl.b.load + zb'.cost <= servCap)
      for a<-0 to sizeof(zb'.PL) - 1
        // first check for direct satisfaction
        if delay(zb'.pl_a, sl_b) <= QoS
          numQoS++
        else
          // only if not directly satisfiable by target
          // server, check delay from separate contact
          // server to target server
          for c<-0 to sizeof(SL) - 1
            if (delay(zb'.pl_a, sl_c) + delay(sl_c, sl_b) <= QoS)
              numQoS++
              break for loop c
      // if pQoSZ is satisfied, remove the zone from stage 2 zones
      // and assign the zone
      if((numQoS / sizeof(zb'.PL)) >= pQoSZ)
        ZB -= zb'
        sl_b.load += zb'.cost
        break for loop b
```

**Heuristic:** GBP (cont.)
if sizeof(ZB) != 0 return ZB
else return NULL

Function: backup_op_2 (Stage 3)

Inputs

ZB = {zb₀, zb₁, ..., zbₙ₋₁} – an array of zones not satisfiable by direct zone-to-server assignments or by the first backup operation. Maximum size is sizeof(Z). Each zone in ZB contains PL, an array of all players in the zone along with their delays to each server.
SL – a subset of all servers S that satisfies all zones not in ZB
pQoS – fraction of players for which we need to achieve QoS to satisfy a zone
QoS – QoS threshold value
servCap – maximum server capacity

Variables

numQoS – number of players in zone with QoS
ZB´ - used for iteration through ZB and removal of satisfied zones
IS – all inactive servers {S – SL}
exitFlag – used to help determine when we had a successful assignment

Output

SL if all zones satisfied; NULL otherwise

Method

//We first will perform the same steps as in backup_op_1. We do this here because
//during the course of backup_op_2, SL can change as new servers are added to it for other
//unsatisfied zones, so we want to ensure that the new SL does not satisfy the next
//unsatisfied zone before adding another new server.
for i =< 0 to sizeof(ZB´) – 1
  exitFlag =< 0
  for b =< 0 to sizeof(SL) – 1
    numQoS =< 0
    if(slₜ.load + zb´.cost ≤ servCap)
      for a =< 0 to sizeof(zb´.PL) - 1
        if delay(zb´.plₚ, slₜ) ≤ QoS
          numQoS++
        else
          for c =< 0 to sizeof(SL) - 1
      Heuristic: GBP (cont.)
if (delay(zb\text{'},pl_{a}, s_{l_{c}}) + delay(s_{l_{c}}, s_{l_{b}}) \leq QoS)
numQoS++
break for loop c

if((numQoS / sizeof(zb\text{'},PL)) \geq pQoSZ)
    ZB := zb\text{'}
    s_{l_{b}.load} += zb\text{'},cost
    exitFlag <- 1
    break for loop b

//if we cannot use the existing SL to satisfy the zone, now check if we can add
//a server to use interserver connections with to satisfy the zone
if(exitFlag == 0)
    //iterate over all unselected servers
    for d<- 0 to sizeof(IS) - 1
        numQoS <- 0
        SL += IS_{d} //add the server temporarily to SL
        //perform identical steps to Stage 2
        for b<-0 to sizeof(SL) - 1ZZ
            numQoS <- 0
            if(s_{l_{b}.load} + zb\text{'},cost \leq servCap)
                for a<-0 to sizeof(zb\text{'},PL) - 1
                    if delay(zb\text{'},pl_{a}, s_{l_{b}}) \leq QoS
                        numQoS++
                    else
                        for c<-0 to sizeof(SL) - 1
                            if (delay(zb\text{'},pl_{a}, s_{l_{c}}) +
                                delay(s_{l_{c}}, s_{l_{b}}) \leq QoS)
                                numQoS++
                    if((numQoS / sizeof(zb\text{'},PL)) \geq pQoSZ)
                        ZB := zb\text{'}
                        s_{l_{b}.load} += zb\text{'},cost
                        exitFlag <- 1
                        break for loop b
        if flag is set then the added server should be kept in SL
        //permanently and the zone is satified
        if(exitFlag == 1)
            IS := IS_{d}
            break for loop d
        //if the flag was not set, then we need to remove the
        //temporary addition
else SL := IS_{d}

if sizeof(ZB) == 0 return SL
else return NULL
2.4.2.4 Time Complexity Analysis

We derive the time complexity for GBP as follows.

- Stage 1: Stage 1 consists of two parts, the precalculation of zone satisfaction and the main loop of GBP. Let $m$ denote the total number of servers in the system and let $p$ denote the total number of players in the system, i.e., in all zones in $Z$. For each zone-server pair we call $\text{checkQoS}(\cdot)$ to check whether the server satisfies $pQoS_Z$ for the zone. Function $\text{checkQoS}()$ performs a QoS check on each player within the zone for the server. Across all calls to $\text{checkQoS}()$, precalculation tests QoS for each player in the game on each server, therefore the complexity for precalculation is $O(mp)$.

Before the main loop of GBP, we copy $S$ into $R$, which will take $O(m)$ time.

During the main loop of GBP, in the worst case we add no servers to $SL$ for any zone. This makes $\text{sizeof}(R) = \text{sizeof}(S) = m$ throughout the main loop. Let $k$ denote the number of zones in $Z$. Since we iterate through $Z$ checking for zone satisfaction by every server in $R$, the time complexity for the main loop is $O(mk)$. The total time complexity for Stage 1 is then $O(mp + m + mk)$. Since $p >> k$, we can conclude that Stage 1’s time complexity is $O(mp)$.

- Stage 2: Let $q$ denote the number of servers in $SL$. Let $n$ denote the number of zones in $ZB$, the zones being operated on by Stage 2. Let $r$ denote the total number of players in $ZB$, that is,

$$r = \sum_{s=0}^{n-1} \text{sizeof}(zb_s \cdot PL).$$

\[\text{1 Although pre-calculation time dominates Stage 1, if we did not perform these operations the worst-case time complexity would not change since we would still need to check each player-server pair during the main loop.}\]
In the worst case, for each player in ZB we check each server pair in SL (i.e.,
interserver connection) for QoS satisfaction. This makes the time complexity
$O(q^2r)$. We also copy ZB to ZB' which takes $O(n)$ time. The total time
complexity of Stage 2 is therefore $O(q^2r + n)$, or $O(q^2r)$ as $n < r$.

- **Stage 3:** Let $n$, $q$ and $r$ denote the same values as in Stage 2. Let $t$ denote the
  number of servers in $IS = S - SL$. The first loop in Stage 3 repeats the
  operations of Stage 2, which is shown above to be $O(q^2r)$. The second loop
  performs these operations again after adding a server from IS into SL. In the
  worst case, we perform this loop for every server in IS. Therefore in the
  worst case the time complexity for the second loop is $O(q^2rt)$. This makes the
time complexity for Stage 3 $O(q^2r + q^2rt)$, or simply $O(q^2rt)$.

Now that we have derived the time complexity for each piece of GBP, let us
determine the total time complexity. In the worst case we perform Stages 1, 2, and 3.
During Stage 2, in the asymptotic worst case, $SL$ contains a constant fraction of all servers
in $S$ and $ZB$ contains a constant fraction of all players in the system. Therefore $q = O(|SL|) =
O(|S|) = O(m)$, and $r = O(p)$, so the time complexity of Stage 2 is $O(m^2p)$. In Stage 3, $IS$ grows
inversely with $SL$ since $IS = S - SL$, so in the worst case $t = O(sizeof(IS)) = O(sizeof(SL)) =
O(q) = O(m)$. This makes the time complexity of Stage 3 $O(m^3p)$. The total time complexity
for GBP is therefore $O(mp + m^2p + m^3p)$, or $O(m^3p)$.

We will be measuring our heuristic performance against a heuristic by Ta et al. [44]
called Greedy-Z, whose time complexity is also $O(m^3p)$. (We go into more detail about their
approach in Section 2.5.2.) Although the time complexity for GBP and Greedy-Z are the
same in the worst case, we expect our performance to be better than Greedy-Z. GBP
contains three distinct stages, with the slowest stage (highest time complexity) being a last resort for those zones not satisfied by the prior, more efficient methods, as opposed to Greedy-Z which contains computationally expensive operations only. We expect our heuristic to make most of its assignments during Stage 1 and Stage 2, which we have shown to have smaller time complexities than Greedy-Z. This should result in lower overall runtimes than Greedy-Z in the average case.

2.4.3 GBP-Fill

2.4.3.1 Overview

We next identify an area of improvement in GBP. Recall that GBP processes zones strictly in descending order of cost. During Stage 1, if a server in SL cannot satisfy the next zone in the zone list, then we activate a new server and add it to SL to accommodate that zone. We must check each server in SL for each zone, because although a server in SL may not satisfy the current zone or may lack capacity to accommodate it, that does not disqualify the server from satisfying or accommodating a later zone.

We can instead assign zones in a different order, taking advantage of the information gathered during precalculation. Once we add a server to SL, instead of proceeding to the next zone, we add to that server in descending order of cost as many zones as the server satisfies and can accommodate. Processing items this way ensures that once a server is selected for SL, we need not check that server again in future iterations, as we assign all satisfiable zones when the server is selected (barring capacity limitations). This does not change the worst-case time complexity because in the worst case we still check each zone-server pair, but the elimination of redundant checks of SL can decrease the
number of operations in most cases and lead to a faster runtime for the heuristic. We refer
to this modified GBP as GBP-Fill.

2.4.3.2 GBP-Fill Pseudocode

The pseudocode for GBP-Fill is as follows.

<table>
<thead>
<tr>
<th>Heuristic: GBP-Fill</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
</tr>
<tr>
<td>$S$ – array of $m$ servers = { $s_0, s_1, ..., s_{m-1}$ }, along with their capacities.</td>
</tr>
<tr>
<td>$Z$ – array of $k$ zones = { $z_0, z_1, ..., z_{k-1}$ }, sorted in descending order of cost. Each zone contains $PL$, an array of all players in the zone along with their delays to each server.</td>
</tr>
<tr>
<td>$pQoSZ$ – the fraction of players in a zone that must be with QoS for the zone to be considered with QoS</td>
</tr>
<tr>
<td>$QoS$ – QoS threshold value</td>
</tr>
<tr>
<td><strong>Variables</strong></td>
</tr>
<tr>
<td>$R = { r_0, r_1, ..., r_{m-1} }$ – a copy of $S$, as a list, used for iteration purposes so we may remove servers and add them to $SL$ while maintaining the integrity of $S$.</td>
</tr>
<tr>
<td>$ZB$ – an array of marked zones passed to backup operations (if necessary)</td>
</tr>
<tr>
<td>$Y = { y_0, y_1, ..., y_{k-1} }$ – A copy of $Z$ as a list used for iteration purposes so we may remove zones and add them to $ZB$ while maintaining the integrity of $Z$. $y.server$ is used to store the identity of a zone's assigned server for flagging purposes.</td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
</tr>
<tr>
<td>$SL$ – sublist of $S$ chosen to host all zones in $Z$ and provide QoS for each. If such a sublist does not exist, the algorithm returns NULL. Newest element added is always at the head of the list.</td>
</tr>
<tr>
<td>$S$ – for each server in $S$ we calculate $ZL$, an array tracking which zones each server satisfies, and store those in $S$</td>
</tr>
<tr>
<td><strong>Method</strong></td>
</tr>
<tr>
<td>//Begin Stage 1</td>
</tr>
<tr>
<td>//perform precalculation of satisfactory zone-server assignments</td>
</tr>
<tr>
<td>for $a&lt;0$ to sizeof($Z$) - 1</td>
</tr>
<tr>
<td>for $b&lt;0$ to sizeof($S$) - 1</td>
</tr>
<tr>
<td>if(checkQoS($z_a, s_b, pQoSZ, QoS) == 1)</td>
</tr>
<tr>
<td>$s_b.ZL[a] &lt;- 1$</td>
</tr>
</tbody>
</table>

Heuristic: GBP-Fill (cont.)
else \( s_b.ZL[a] \leftarrow 0 \)

copy \( S \) into \( R \) as a list

\( \text{copy Z into Y as a list} \)

\( SL \leftarrow \text{NULL} \) //initialize

//main loop – go through all remaining zones and add servers/make assignments until
//they are all satisfied or we have added them to ZB, the list of zones for backup
//operations. Once we have done so, Y will be empty
while(sizeof(Y) > 0)

//check all servers not in SL to find one that satisfies the next zone in Y
//add the server to SL and remove the zone from Y and the server from R
\( server\_flag \leftarrow 0 \)
for(\( b \leftarrow 0 \) to sizeof(R) - 1)

if(\( r_b.load + y_0.cost \leq servCap \&\& \ r_b.ZL[\text{ID}(y_0)] == 1 \))

\( server\_flag \leftarrow 1 \)
\( r_b.load += y_0.cost \)
\( SL += r_b \)
\( R -= r_b \)
\( Y -= y_0 \)

break for loop b

//if we do not find a server to satisfy the zone, add the zone to ZB for backup
//operations
if(\( server\_flag == 0 \))

\( ZB += y_0 \)
\( Y -= y_0 \)

else

//we now check all remaining unsatisfied zones to see if the newly
//activated server will satisfy any of them
//sl is new SL server at the list head
\( \text{copy Y to } Y^\prime \)
for(\( c < 0 \) to sizeof(Y') - 1)

if(\( sl_0.load + y'_c.cost \leq servCap \&\& \ sl_0.ZL[\text{ID}(y'_c)] == 1 \))

\( sl_0.load += y'_c.cost \)
\( Y^- = y'_c \)

if(sizeof(ZB) == 0) return SL //if nothing in ZB, we are done

//end Stage 1
//Begin Stages 2 and 3
else

\( ZB \leftarrow \text{backup\_op\_1(SL, ZB, pQoSZ, QoS)} \) //Stage 2 call

if(\( ZB == \text{NULL} \)) return SL //if NULL all zones were satisfied
else

\( SL = \text{backup\_op\_2(SL, ZB, pQoSZ, QoS)} \) //Stage 3 Call

return SL //will return either SL or NULL

Heuristic: GBP-Fill (cont.)
2.4.4 GBP-Skip+Fill

2.4.4.1 Overview

During experiments under several different sets of simulation parameters, we encountered zones that are "difficult" to satisfy during Stage 1 of GBP. In multiple cases, zones exist that are satisfiable by very few servers (as little as one or zero) using direct connections, even in cases where there are many servers (100 or more) from which to choose. It is detrimental to overall runtime to perform Stage 1’s operations on these zones. This leads us to further modify GBP in a way we call GBP-Skip+Fill. In GBP-Skip+Fill, we improve upon GBP-Fill by adding the ability to "skip" over zones that are difficult to satisfy (directly satisfiable by one or zero servers) during Stage 1 and instead add those zones immediately to the list of zones to be satisfied during Stages 2 and 3. We can identify these zones during precalculation and set them aside before Stage 1 begins. This way we do not tie up execution time or make special case additions of servers to SL to satisfy these tough zones – we instead leave them for backup operations to satisfy using interserver connections.

2.4.4.2 GBP-Skip+Fill Pseudocode

The pseudocode for the precalculation phase of GBP-Skip+Fill is as follows. All other steps are identical to GBP-Fill.

<table>
<thead>
<tr>
<th>Heuristic: GBP-Skip+Fill</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
</tr>
<tr>
<td>( S ) – array of ( m ) servers = { s_0, s_1, ..., s_{m-1} }, along with their capacities.</td>
</tr>
<tr>
<td>( Z ) – array of ( k ) zones = { z_0, z_1, ..., z_{k-1} }, sorted in descending order of cost. Each zone contains ( PL ), an array of all players in the zone along with their delays to each server.</td>
</tr>
<tr>
<td>( pQoSZ ) – the fraction of players in a zone that must be with QoS for the zone to be satisfied.</td>
</tr>
</tbody>
</table>

Heuristic: GBP-Skip+Fill (cont.)
considered with QoS
QoS – QoS threshold value

Variables

$Z_B$ – an array of marked zones passed to backup operations (if necessary)
$Y = \{y_0, y_1, \ldots, y_{k-1}\}$ – A copy of $Z$ as a list used for iteration purposes so we may remove zones and add them to $Z_B$ while maintaining the integrity of $Z$.

Outputs

$SL$ – sublist of $S$ chosen to host all zones in $Z$ and provide QoS for each. If such a sublist does not exist, the algorithm returns NULL. Newest element added is always at the head of the list.
$S$ – for each server in $S$ we calculate $Z_L$, an array tracking which zones each server satisfies, and store those in $S$

Method

//perform precalculation of satisfactory zone-server assignments
for $a<-0$ to sizeof($Z$) – 1
  $z_a.numSat<-0$
  for $b<-0$ to sizeof($S$) - 1
    if(checkQoS($z_a, s_b, pQoS, QoS) == 1)
      $s_b.ZL[a]<-1$
      $z_a.numSat++$
    else $s_b.ZL[a]<-0$

copy $Z$ into $Y$ as a list
for $a<-0$ to sizeof($Y$) – 1
  if $y_a.numSat \leq 1$
    $ZB += y_a$
    $Z -= y_a$

continue with operations of GBP-Fill

Now that we have established the motivation and detail of our heuristics, in the next section we will discuss our simulation techniques and results, along with the work of Ta et al. [44], whose problem and heuristics serve as comparison for our performance.
2.5 Simulation

2.5.1 Ta et al.’s Interactivity-Constrained Server Provisioning Problem

Ta et al. [44] proposed the *interactivity-constrained server provisioning problem*, which is a modified offline CAP that removes server capacity constraints. The authors developed heuristics for both an offline modified CAP-Z (MCAP-Z) and an offline modified CAP-C (MCAP-C).

For offline MCAP-Z, the inputs are the same as in offline CAP-Z, minus the server capacities. The primary goal is to assign all zones to servers, leaving no unsatisfied zones. A secondary goal is to minimize the number of servers used in the assignment. The offline MCAP-C is similar to the offline MCAP-Z except instead of $pQoS_Z$ the inputs contain $pQoS_C$, which specifies the minimum fraction of the overall number of players that must be with QoS, and the primary goal is to make assignments that meet $pQoS_C$. Table 2.1 summarizes the variations of the CAP.

The authors tested the performance of four heuristics for offline MCAP-Z, two of which we describe here. The first is SetCover-Z, their implementation of an algorithm for

**Table 2.1.** CAP variations. Players, servers, their delays, and QoS are assumed inputs to each variation.

<table>
<thead>
<tr>
<th>CAP type</th>
<th>MMOG running?</th>
<th>Capacity constraints?</th>
<th>Satisfy clients or zones?</th>
<th>Additional Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline CAP-C</td>
<td>No</td>
<td>Yes</td>
<td>Clients</td>
<td>Server capacities</td>
</tr>
<tr>
<td>Online CAP-C</td>
<td>Yes</td>
<td>Yes</td>
<td>Clients</td>
<td>Server capacities</td>
</tr>
<tr>
<td>Offline CAP-Z</td>
<td>No</td>
<td>Yes</td>
<td>Zones</td>
<td>Server capacities, zones, client-zone mapping, $pQoS_Z$</td>
</tr>
</tbody>
</table>

(table cont’d.)
Online CAP-Z | Yes | Yes | Z | Zones | Server capacities, zones, client-zone mapping, pQoSZ
Offline MCAP-Z | No | No | Z | Zones | Zones, client-zone mapping, pQoSZ
Offline MCAP-C | No | No | C | Clients | pQoS

the NP-hard Set Cover problem [14]. The authors showed that the offline MCAP-Z generalizes to this NP-hard problem. In this heuristic, the inputs are a set of servers, set of zones, and set of players (with delays). The output is \( SL' \), a subset of the given set of servers that will satisfy all zones. The heuristic builds \( SL' \) iteratively. Each iteration temporarily adds to \( SL' \) one server not in \( SL' \). The heuristic tests \( SL' \) to see how many zones it can satisfy (and as previously specified, the \( pQoS \) threshold must be met for a server to satisfy a zone), then removes the server from \( SL' \). After performing this operation using each server not in \( SL' \), it permanently adds to \( SL' \) the server that (along with \( SL' \)) satisfied the highest number of zones. These iterations occur until \( SL' \) satisfies all zones or no combination of servers can satisfy all zones (at which point the heuristic fails).

The second algorithm, Greedy-Z, performs the same operations as SetCover-Z, with one change. When considering whether the final list of servers satisfies a zone, Greedy-Z attempts to satisfy clients in a zone through interserver connections by setting his or her contact server and target server as two different servers in \( SL' \). For example, say the potential final server list consists of two servers, \( S_1 \) and \( S_2 \). Two clients, \( C_1 \) and \( C_2 \), exist in a zone \( Z \) to be assigned. Figure 2.1 shows the clients, servers, and their respective delays. Let \( pQoS \) be 0.8, so 80% of the players in \( Z \) must achieve QoS for \( Z \) to be within QoS. In our case, this means both players must achieve QoS for \( Z \) to achieve QoS, because were only one client to be within QoS, our ratio of satisfied players would be 0.5, less than \( pQoS \). Assume
the value for QoS is 150. In SetCover, this list of servers would not satisfy Z, because were we to assign Z to S₁, we would satisfy C₁ but not C₂, and if we were to assign Z to S₂, we would satisfy C₂ but not C₁. However, using Greedy-Z, if we assign Z to S₁ (target server), we satisfy C₁ directly (a delay of 100) with its contact server as S₁, and we satisfy C₂ indirectly by setting its contact server to S₂. The total delay for C₂ will be 125, which is less than QoS, and we thus satisfy C₂. Therefore we can satisfy Z with the given servers using the Greedy-Z algorithm, but not the SetCover-Z algorithm.

2.5.2 Analysis of Offline MCAP-Z, Extension to Offline CAP-Z and Parameters

One assumption Ta et al. made in the offline MCAP-Z is that each server may accommodate an unlimited number of clients. They considered each server to be a data center containing a large number of physical servers and other hardware, and that the cost of scaling the capacity of the data center is negligible in comparison to deploying new “servers” (data centers). This assumption corresponds to unlimited capacity for any offline MCAP-Z server. This can lead to some results that may be difficult or impossible to implement in a practical environment – if a server is centrally located in the physical world and is generally a good option for most players, then in the offline MCAP-Z that server can

![Figure 2.1. Set of clients and servers used to contrast Greedy-Z and SetCover-Z.](image)
and likely will host all or most of the zones in the virtual world with no consideration given for the strain or load placed on that server.

We will bring Ta et al.'s heuristics into a standard offline CAP-Z environment. Servers do have limits, and since a server can be both a target and contact server, it is responsible for routing a large amount of network traffic (as a contact server), in addition to computation for the zones it is hosting (as a target server). If a server becomes overloaded, it could hinder game performance or cease gameplay completely. We must consider this when choosing to which servers we will assign zones. In offline CAP-Z, our goal is the same as in offline MCAP-Z – to assign zones to servers such that the threshold of satisfied players per zone specified by $pQoS Z$ is met and to do so with a minimum number of servers. In addition, each server has some capacity constraint, so this assignment must limit the number of players assigned to each server. Ta et al. assumed interserver connections to be well-provisioned, that is, to have near negligible delay. In the physical world, servers may be located large distances apart, and while their connections to one another may be fast, they likely will not be negligible. We do not consider interserver connections to be negligible, though we do consider them to be better than player-to-server connections. To accommodate this, we introduce a scaling factor that allows us to adjust how much better server-to-server connections are than player-to-server connections.

### 2.5.3 Introduction to Simulator

In order to simulate servers, clients, and delays among all of them, we generate an $n \times n$ grid as an $n \times n$ two-dimensional array, where $n$ is a parameter passed to the simulator. Each adjacent index in the array corresponds to 1 ms of delay, and the Euclidean distance
between any two points (players or servers) determines the delay between them. For example, given a client position of [2][3], and a server position of [2][4], the client would have a 1 ms delay to that server.

The simulator randomly places players and servers throughout the grid, partitions the game world into zones, and maps players to zones based on the respective parameters passed to it. The inputs given to each run of the simulator consist of the following:

- Number of players
- Number of servers
- Number of zones
- Capacity of each server
- Grid size
- Value for QoS
- Value for pQoS
- Hotspot zone fraction (HSF)
- Fraction of players in hotspots (HSD)
- Interserver delay scaling factor

The hotspot zone fraction parameter (HSF) determines the fraction of zones that are considered to be “hot spots”, or zones where players in an MMO might gather more densely than others (such as in towns or other centralized locations in the game). A second hotspot parameter (HSD) dictates the fraction of overall players located in hotspot zones. Using these values, the simulator assigns players to random zones.

Example: Suppose HSF is 0.2, HSD is 0.4 and there are ten zones. Then, two of the ten zones will be hotspots and those two hotspot zones will contain 40% of all players. The remaining 60% of players will be distributed among the remaining eight “cool” zones.

Although in their offline MCAP-Z experiments Ta et al. considered the connections between servers to be well-provisioned (that is, with near negligible delay), we do not make this assumption. In offline MCAP-Z, Ta et al. set each interserver delay to be 10% of
the delay value calculated from their Euclidean distance. An equivalent assumption in our simulator, setting the interserver delay scaling factor to 0.1, would generate a delay of 20 ms between two servers whose Euclidean distance would give a 200 ms delay without this assumption. While interserver delays can be faster than player-to-server delays, they are not negligible. We find a more reasonable value for this parameter to be 0.8 (80% of the calculated delay value), which makes interserver delays faster than player-to-server delays with equivalent Euclidean distance, but not so much faster that they potentially skew simulation results.

Each run of the simulator begins with an initialization phase that sets up the environment under which each heuristic operates. This phase starts by generating a grid (size given by the grid size parameter) and pseudorandomly placing players and servers in the grid with a uniform distribution. It then pseudorandomly maps players to zones with the distribution calculated by the hotspot parameters previously described. After initialization, using the simulated heuristic’s method, the simulator assigns zones to servers. If we assign all zones to servers successfully, then the simulator reports the number of servers used to make the assignment and the runtime for that heuristic. Otherwise it simply reports that the assignment failed. We use these statistics to compare the heuristics’ performance in terms of failure fraction (the fraction of failed simulations to total number of simulations run for that heuristic for the indicated set of parameters), total runtime, and number of active servers used to satisfy all zones. The results include only successful assignments, so if a heuristic A has a much higher failure fraction than a heuristic B, then the number of active servers is over a smaller, and easier, number of runs in A than in B.
2.5.4 Simulation Results

2.5.4.1 Overview

In this section, we experimentally verify our motivation for Stages 2 and 3 of GBP, and then we perform some targeted probes of the simulator parameter space, altering the values of some parameters to observe the effects on servers used, runtime and failure rate for GBP, GBP-Fill, GBP-Skip+Fill, Greedy-Z, and SetCover-Z. The complete parameter space is vast, as we have ten different input parameters to our simulations and each parameter has a large range of plausible values. In order to display a subset of the parameter space that gives a sample of heuristic performance without exploring the entire parameter space (which would take a very long time and be very difficult to display), we fix some parameters to static values and limit the range and interval of the remaining parameter values.

In order to help narrow the parameter space and help us determine where to focus our simulations, we performed many simulations using a multitude of input parameter values to gauge each environment’s assignment potential, that is, the likelihood that an environment created with those input parameters would allow all of our simulated heuristics to perform non-trivial zone-server assignments. We gauged this potential by generating a virtual environment as we would in a typical simulation with the same input parameters and distributing players among zones. We then determined the number of zones that were unsatisfiable by direct connections and the number of servers that could not satisfy any zone directly. Parameter sets that contained too many unsatisfiable zones or too few servers that could satisfy zones for all simulated heuristics (both ours and Ta et al.’s) were pruned – all heuristics simulated have very high failure fractions for those
parameter choices. We also pruned sets that were too “easy” for all simulated heuristics (both ours and Ta et al.’s) – sets that contained too few directly unsatisfiable zones or too many servers that could provide direct zone satisfaction. The particular thresholds for too “easy” or “hard” were made relative to the full set of parameters simulated – for example, for a particular parameter set, as we increased the number of total servers the percentage of directly unsatisfiable servers remained between 5-15%. Once we reached a value for number of servers where percentage of unsatisfiable zones suddenly increased to 30-40%, those were pruned as too difficult. This allows us to explore a space that is non-trivial and has potential for interesting results without fully exploring every value for every parameter.

Each reported result is the average of 25 separate runs of the simulator for the given heuristic under the specified parameters. We report the average number of servers used, the average runtime, and the average failure fraction for each heuristic. The following parameters remain static among all runs reported here:

- Number of players - 5000
- Grid Size – 250x250
- Value for QoS - 150
- Interserver delay scaling factor – 0.8

2.5.4.2 GBP Approach Verification

One motivating factor behind Stages 2 and 3 of GBP was to reduce the failure fraction of the heuristic while still running as fast or faster than Greedy-Z. To validate this we compare performance in terms of active servers used, runtime and failure fraction between GBP and GBP Stage 1. Figures 2.3, 2.4 and 2.5 show a sample of simulation results.
comparing Ta et al.’s Greedy-Z, their implementation of SetCover-Z, GBP, GBP-Fill, GBP-Skip+Fill, and GBP Stage 1.

Figures 2.2 and 2.3 show the average number of active servers and failure fraction among all the heuristics for a few select parameter choices. These parameters display example cases that exhibit some of the same behaviors as the broader set of parameters shown in later simulations. GBP Stage 1 uses fewer active servers than GBP, GBP-Fill and GBP-Skip+Fill, however, its failure rate is very high – in some cases, even higher than SetCover-Z. Adding Stages 2 and 3 reduces the failure fraction from as high as 0.9 to less than 0.05. A successful assignment is preferable to an unsuccessful one, so the large reduction in failure fraction is worth the tradeoff of activating more servers. As we can see in Figure 2.4, the total runtime of each of GBP, GBP-Fill and GBP-Skip+Fill is still very small as opposed to the runtimes of SetCover-Z and Greedy-Z, which is in line with our goals for GBP.

2.5.4.3 Targeted Probes of Parameter Space

2.5.4.3.1 Targeted Probe Overview

Now that we have validated our motivation behind GBP, we perform targeted probes of the parameter space to investigate how these heuristics perform under various
Figure 2.2. Average number of active servers for combinations of 20,40 zones and 50,100 servers.

Figure 2.3. Average failure fraction for combinations of 20,40 zones and 50, 100 servers.
Figure 2.4. Average runtime for combinations of 20,40 zones and 50,100 servers.

combinations of parameters. We will modify the number of zones, the number of servers, the value of pQoSZ, the maximum server capacity, the hotspot factor, and the hotspot density to a few different experimentally chosen values.

Table 2.2 displays the parameter choices we will use among the simulation results displayed here.

We explain the justification for these choices below.

- **Number of zones** – When the number of zones is set fewer than 20, each zone contains many players and satisfying pQoSZ is very difficult for all heuristics unless pQoSZ is set to a very low value (0.5 or less), which then makes the case trivial for all heuristics. When the number of zones is greater than 80, each zone contains few players and again satisfying pQoSZ becomes too difficult unless pQoSZ is set to a low value.
Table 2.2. Parameter choices among all reported simulations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of zones</td>
<td>20, 40, 60, 80</td>
</tr>
<tr>
<td>Number of servers</td>
<td>50, 100, 150, 200</td>
</tr>
<tr>
<td>pQoSZ</td>
<td>0.85, 0.9</td>
</tr>
<tr>
<td>Server capacity</td>
<td>1500, 3000</td>
</tr>
<tr>
<td>HSF</td>
<td>0.1, 0.4</td>
</tr>
<tr>
<td>HSD</td>
<td>25%, 50%</td>
</tr>
</tbody>
</table>

- **Number of servers** – With fewer than 50 servers, there do not exist enough choices for zone assignments. There end up being only a few servers that will satisfy any zone, and once we fill those servers to capacity we cannot assign any other zones to servers. The failure fraction for all heuristics is near or at 1.0 in these cases. For greater than 200 servers, we reach a saturation point where increasing the number of available servers has no effect on number of active servers used.

- **pQoSZ** – For values of pQoSZ greater than 0.9, for our other choices of parameters solving the cases become too difficult for all of the simulated heuristics – they rarely, if ever, succeed. Likewise, for values of pQoSZ less than 0.85, the cases become too easy for the heuristics to solve – lowering the value past this point has no effect on the number of active servers.

- **Server capacity** – This parameter can be tricky in that it can be a bottleneck in the simulation for particular choices of HSF, HSD and number of zones (i.e., in cases where the number of players in the largest zone is greater than maximum server
capacity), or it can make the problem too easy to solve (such as in the case when all zones can be assigned to a single server that has capacity for every player in the system). We found that based on the choices we made for the other parameters in the simulation, 1500 and 3000 capacity were two values that provided non-trivial results for a majority of cases. It is a somewhat arbitrary choice based on the total number of players, but as we will see it can have a significant effect on results.

- **HSF, HSD** – We use these parameters only to help create a non-uniform distribution of players among all zones and to simulate the hotspots that occur in virtual environments. As such we are not as interested in exploring their space as we are the other parameters. We found the values we used to be neither trivial nor a performance bottleneck, but they are not as strongly justified as the other parameters.

Table 2.3 displays the combinations of parameter values we use in our simulations. The first group shown, group A, establishes the case to which each other group will be compared. The far right column shows which parameter in the group we alter from the

<table>
<thead>
<tr>
<th>Group</th>
<th>pQoSZ</th>
<th>Capacity</th>
<th>HSF</th>
<th>HSD</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.9</td>
<td>3000</td>
<td>0.1</td>
<td>25%</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>0.9</td>
<td>1500</td>
<td>0.1</td>
<td>25%</td>
<td>Capacity</td>
</tr>
<tr>
<td>C</td>
<td>0.85</td>
<td>3000</td>
<td>0.1</td>
<td>25%</td>
<td>pQoSZ</td>
</tr>
<tr>
<td>D</td>
<td>0.9</td>
<td>3000</td>
<td>0.4</td>
<td>50%</td>
<td>HSF/HSD</td>
</tr>
</tbody>
</table>

Table 2.3. Simulation parameter groupings.
initial case and observe. Note that each group contains four sets of results – one set for each choice of number of zones (20, 40, 60, 80).

**2.5.4.3.2 Targeted Probes – Initial Case (Group A)**

In our initial case we choose pQoSZ to be 0.9. We choose initial values of 0.1 and 25% for HSF and HSD, respectively. Since number of players is always 5000, then for \( f \) zones we can expect to have an average of \( \frac{5000 \times 0.25}{0.1 + f} \) players per hotspot zone (or \( \frac{5000 \times 0.75}{0.9 + f} \) players per non-hotspot, or “cool” zone). Table 2.4 shows the average number of players per hotspot zone and cool zone for each choice of number of zones. In this group we choose server capacity to be 3000 – this means that the fewest possible number of active servers for these parameter choices is 2.

Figures 2.5, 2.6 and 2.7 show the average servers used, average failure fraction, and average runtime of the 20 zone case for parameter group A. For 50 servers we see that GBP uses nearly twice the number of active servers as Greedy-Z and the same failure fraction. GBP-Fill and GBP-Skip+Fill use fewer servers than GBP and also have no failures. GBP-Fill uses the fewest active servers of the GBP variants.

**Table 2.4.** Average number of players per hotspot and cool zone for given input parameters.

<table>
<thead>
<tr>
<th>Number of zones</th>
<th>Number of hotspot zones</th>
<th>Average players per hotspot zone</th>
<th>Number of cool zones</th>
<th>Average players per cool zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>2</td>
<td>625</td>
<td>18</td>
<td>208</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>312</td>
<td>36</td>
<td>104</td>
</tr>
<tr>
<td>60</td>
<td>6</td>
<td>240</td>
<td>54</td>
<td>69</td>
</tr>
<tr>
<td>80</td>
<td>8</td>
<td>156</td>
<td>72</td>
<td>52</td>
</tr>
</tbody>
</table>
Average Active Servers (25 Sims)

![Graph showing average active servers across different numbers of total servers.](image)

**Figure 2.5.** Average number of active servers, group A, 20 zones.

Increasing the total number of servers to 100 or more significantly reduces the number of active servers used by GBP, fewer than the other GBP variants. In this case for parameter group A, adding more available servers appears to mitigate the benefits of GBP-Fill and GBP-Skip+Fill. Greedy-Z and SetCover-Z use the minimum number of active servers for 100 or greater servers.

SetCover-Z has a high failure rate for the 50 server case and is the only heuristic to fail in the 100 server case, so we cannot draw many conclusions from its results – it fails much more often than any other heuristic, which is a trait we will see occur throughout the rest of our simulations.

SetCover-Z and Greedy-Z increase in runtime as total number of servers increases, but GBP does not. We expect this because for a greater number of servers, we have more direct connections to satisfy zones, which keeps us from needing to proceed to Stage 2 and
Figure 2.6. Average failure fraction, group A, 20 zones.

Figure 2.7. Average runtime, group A, 20 zones.
3 as often. However, for SetCover-Z and Greedy-Z this means more connections to consider while determining which SL will satisfy all zones.

Table 2.5 shows the runtimes for the 20 zone case in parameter group A to provide an example of the differences among the very small GBP runtimes. GBP and GBP-Fill are faster than GBP-Skip+Fill here – we expect this in the cases where there exists a zone satisfiable by only one server and GBP or GBP-Fill are able to assign that server during Stage 1. GBP-Skip+Fill will not attempt to satisfy the zone during this stage and will use more computationally complex operations to make the assignment, increasing total runtime over the assignment GBP or GBP-Fill would produce.

The 40 zone case in Figures 2.8, 2.9 and 2.10 shows that the average number of servers used increases over the 20 zone case for all heuristics in the 50 server case. GBP uses the greatest number again, but now GBP-Skip+Fill uses the fewest among the GBP variants. GBP-Skip+Fill is also the only heuristic not to fail. Although SetCover-Z appears to use the fewest number of servers, its failure fraction is 0.88 (only 3 successful assignments in 25 runs).

Table 2.5. GBP variant runtimes for 20 zone case in parameter group A.

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>50 servers</th>
<th>100 servers</th>
<th>150 servers</th>
<th>200 servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>GBP</td>
<td>1.48 ms</td>
<td>0.15 ms</td>
<td>0.13 ms</td>
<td>0.12 ms</td>
</tr>
<tr>
<td>GBP-Fill</td>
<td>1.21 ms</td>
<td>0.54 ms</td>
<td>0.51 ms</td>
<td>0.51 ms</td>
</tr>
<tr>
<td>GBP-Skip+Fill</td>
<td>1.56 ms</td>
<td>0.70 ms</td>
<td>0.54 ms</td>
<td>0.51 ms</td>
</tr>
</tbody>
</table>
Figure 2.8. Average number of active servers, group A, 40 zones.

For 100 servers, all heuristics use fewer active servers than the 50 server case, with GBP using nearly half the number of active servers. GBP and GBP-Fill show small fluctuations in average active servers as number of servers increases, while GBP-Skip+Fill and Greedy-Z show consistent decreases in active servers.

As in the 20 zone case for parameter group A, Greedy-Z and SetCover-Z increase in runtime as the number of servers increase, and the runtime for each server choice is greater for 40 zones than 20 zones. This will continue to increase as number of zones increases for every parameter group we simulate.

Figures 2.11, 2.12 and 2.13 show the results for the 60 zone case of parameter group A. For 50 total servers, again the number of active servers used increases as number of zones increases. In the 20 and 40 zone sets, GBP clearly used more servers than the other two GBP variants, but now the difference between GBP and GBP-Fill is much less
Figure 2.9. Average failure fraction, group A, 40 zones.

Figure 2.10. Average runtime, group A, 40 zones.
Figure 2.11. Average number of active servers, group A, 60 zones.

pronounced, and GBP-Skip+Fill uses the same number of active servers on average as GBP. GBP-Fill is the only GBP variant to fail here. Greedy-Z uses half the number of active servers as any GBP variant.

Again we see that increasing the total number of servers benefits GBP greatly, and it uses the fewest active servers of the GBP variants in the 100 server case. However, as the total number of servers is further increased, GBP-Fill and GBP-Skip+Fill use fewer than GBP.

We can see that this is a more demanding choice of zones than 20 or 40 zones were in parameter group A, as the more computationally complex operations of Greedy-Z and SetCover-Z were able to produce assignments using the minimum number of servers in those cases, but here use an average of 3 or more servers.
Figure 2.12. Average failure fraction, group A, 60 zones.

Figure 2.13. Average runtime, group A, 60 zones.
Increasing the number of zones to 80 in Figures 2.14, 2.15 and 2.16 shows that the number of active servers increases in the 50 server case over the 50 server case of Figure 2.11 for GBP, Greedy-Z and GBP-Skip+Fill. GBP-Skip+Fill shows the largest such increase and uses the most active servers, however it is the only heuristic with no failures for this choice of servers. This is an example of the situation for which it was designed– trading a potentially increased number of servers for a reduced failure fraction.

For 100 total servers we reduce the number of active servers across the board. SetCover-Z is the only heuristic to fail for 100 or more servers. Although GBP and GBP-Fill use fewer active servers than GBP-Skip+Fill in the 100 server case, as we saw in the 40 zone and 60 zone cases (and the 20 zone case, to a lesser extent) the number of active servers used by GBP-Skip+Fill steadily decreases as the total number of servers increases.

![Average Active Servers (25 Sims)](image)

**Figure 2.14.** Average number of active servers, group A, 80 zones.
Figure 2.15. Average failure fraction, group A, 80 zones.

Figure 2.16. Average runtime, group A, 80 zones.
Overall in our initial parameter set we observe that increasing the number of zones and increasing the number of servers visibly increases the runtime of Greedy-Z and SetCover-Z, while adding comparatively little runtime for the GBP variants. Greedy-Z always uses fewer active servers than the GBP variants but with a much greater runtime.

GBP-Skip+Fill does not always produce the assignment with the fewest active servers, but it is the only heuristic to never fail in parameter group A. It also steadily decreases in servers used as total number of servers increases, often using the fewest servers among GBP variants in the 150 and 200 server cases.

50 servers is a demanding choice for the server parameter and will continue to be for the rest of the parameter groups. Although the gap between the GBP variants and Greedy-Z widens as the number of zones increases, increasing the total number of servers narrows that gap significantly.

There is no clear “best” GBP variant here, but we have observed some behaviors of each that we will see in the other parameter groups showing each variant lends itself to situational use.

2.5.4.3.3 Parameter Group B (Server Capacity Effect Probes)

In parameter group B we alter the server capacity in parameter group A from 3000 to 1500 and observe the effects on heuristic performance. Note that for 5000 players, this now increases the minimum possible number of active servers from 2 to 4. We thus expect to see an increase in number of active servers used for all heuristics.

Figures 2.17, 2.18 and 2.19 display the 20 zone case for parameter group B. For 50 servers we see in accordance with our expectations that the number of active servers
increases for all heuristics over the 20 zone, 50 server case of parameter group A in Figure 2.5. Although Greedy-Z uses the fewest active servers, it has a higher failure fraction than all of the GBP variants. All heuristics have some failure fraction, and SetCover-Z continues to have the highest. GBP-Fill uses the fewest active servers of the GBP variants.

For 100 servers and greater, Greedy-Z achieves the minimum active servers and does not fail. No GBP variants fail in these cases either, and the number of active servers used among them is very similar, hovering around the 5 server mark. This is approximately 2 more servers on average than the corresponding parameter group A case, which follows from the minimum possible number increasing by 2. Runtimes for Greedy-Z and SetCover-Z increase significantly (more than double) over the corresponding group A set. GBP variants run in 2 milliseconds or less in this set.

Figure 2.17. Average number of active servers, group B, 20 zones.
Figures 2.20, 2.21 and 2.22 show parameter group B’s 40 zone case. We see an increase of approximately 2 additional active servers used on average by GBP-Fill and GBP-Skip+Fill over the corresponding group A case, but an increase of approximately 4
Figure 2.20. Average number of active servers, group B, 40 zones.

Figure 2.21. Average failure fraction, group B, 40 zones.
additional active servers used on average by GBP. Greedy-Z also increases in number of active servers used over both the 20 zone group B case shown in Figure 2.17 and significantly over its corresponding group A case in Figure 2.5. Failure fraction is less than or equal to the 20 zone group B case displayed in Figure 2.18 for all heuristics except SetCover-Z, which rises to over 0.9.

As we have seen before, increasing the total number of servers benefits GBP variants by reducing the average number of active servers used. GBP still uses the most servers of the variants in the 100 server case but significantly fewer than it does in the 50 server case. GBP-Skip+Fill steadily uses fewer active servers as total number of servers increases, as seen in group A. Greedy-Z registers failures in the 100 and 150 server cases, but no GBP variant does.
Figures 2.23, 2.24 and 2.25 display the 60 zone case for group B. For 50 servers GBP uses more active servers than it did for 40 zones and continues to use more average active servers than any other heuristic, however it has the lowest failure fraction of any heuristic, which is uncommon to this point. The other variants increase in average active servers over their 40 zone results as well, however Greedy-Z does not. We see the highest failure fractions thus far here, with every heuristic failing at least 20% of the time, and SetCover-Z failing 100% of the time. In the 150 and 200 server cases, although GBP uses the largest number of average active servers, it shows the smallest increase in servers used over its corresponding group A case, using only 1.5 additional servers on average, as opposed to approximately 2 additional servers for the other heuristics.

Finally for group B, Figures 2.26, 2.27 and 2.28 show the 80 zone case. GBP actually uses fewer servers here than the 60 zone case, while GBP-Fill and GBP-Skip+Fill increase.
Figure 2.24. Average failure fraction, group B, 60 zones.

Figure 2.25. Average runtime, group B, 60 zones.
**Figure 2.26.** Average number of active servers, group B, 80 zones.

**Figure 2.27.** Average failure fraction, group B, 80 zones.
We attribute this to a reduced failure fraction for GBP-Fill and GBP-Skip+Fill, while the failure fraction for GBP increases. GBP-Fill is the only heuristic with no failures here. SetCover-Z still fails 100% of the time. Runtimes for Greedy-Z and SetCover-Z are the highest seen to this point - nearly 300 ms for Greedy-Z in the 200 server case.

Overall for parameter group B, we can see that decreasing server capacity has a visible effect on average active servers, as we expected since the minimum number possible increased from 2 to 4. The most significant effect decreasing capacity has, however, is a visible increase in simulation runtime for Greedy-Z and SetCover-Z. Since the average size of SL increased for these heuristics, their computations took significantly longer, especially Greedy-Z which continually checks interserver connections for SL during each iteration. The demanding 50 server parameter choice was even more demanding in

**Figure 2.28.** Average runtime, group B, 80 zones.
group B than in A, causing the first failures we have observed for GBP-Skip+Fill and the highest failure fractions we have observed for any heuristic other than SetCover-Z.

2.5.4.3.4 Parameter Group C (pQoS Effect Probes)

For our next set of parameter probes we alter the value for pQoS in parameter group A to 0.85 to observe the effect on the heuristic performance. We expect all heuristics to reduce the average number of active servers used, as a larger number of servers should satisfy pQoS for a larger number of zones. All other parameter choices are the same in group C as in group A.

Figures 2.29, 2.30 and 2.31 show the 20 zone case for group C. We can immediately observe a drastic reduction in servers used in the 50 server case. This was the most demanding choice for the number of servers in the two previous groups, but now the difference in active servers used among all heuristics and among the other choices

![Average Active Servers (25 Sims)](image)

**Figure 2.29.** Average number of active servers, group C, 20 zones.
for number of servers is much smaller. Increasing the total number of servers does not have as obvious of an effect, if any, on active servers used by the GBP variants as it did in

![Average Failure Fraction (25 Sims)](image)

**Figure 2.30.** Average failure fraction, group C, 20 zones.

![Average Simulation Runtime (25 Sims)](image)

**Figure 2.31.** Average runtime, group C, 20 zones.
Groups A and B. We can also see that there is a visible effect on failure fraction as there are no heuristic failures – not even SetCover-Z, which failed frequently in the previous two groups. As we will see in the upcoming figures, no GBP variant fails at all in parameter group C. Runtime is approximately the same as in the corresponding set of parameter group A.

In Figures 2.32, 2.33 and 2.34 we see the 40 zone case for group C. For 50 total servers, the number of active servers used increases for the GBP variants, and GBP uses the fewest of the three for not only this case, but all choices of numbers of servers. Greedy-Z and SetCover-Z continue to produce assignments with the minimum number of active servers. SetCover-Z registers a few failures in the 50 server case, but not for any other choice of number of servers. For choices of 20 and 40 zones and this relaxed pQoSZ parameter, the benefits provided by GBP-Fill and GBP-Skip+Fill are mostly unnecessary to

![Average Active Servers (25 Sims)](chart.png)

**Figure 2.32.** Average number of active servers, group C, 40 zones.
Figure 2.33. Average failure fraction, group C, 40 zones.

Figure 2.34. Average runtime, group C, 40 zones.

produce better assignments, so they show no improvements over the results obtained by GBP.
In the 60 zone case shown in Figures 2.35, 2.36 and 2.37, we observe that active servers used increases again for the GBP variants in the 50 server case. Although GBP uses the fewest servers of the GBP variants in the 50 server case, we again observe that GBP-Skip+Fill and GBP-Fill reduce the number of active servers as total number of servers increases, and they both use fewer servers than GBP for choices of 100, 150 and 200 servers. SetCover-Z registers failures for all choices of servers other than 200, and Greedy-Z does in the 50 server case only. Runtime in this set is the same or lower than in the corresponding set of group A for Greedy-Z and SetCover-Z, and GBP variant runtimes continue to be very low.

In the 80 zone case of Figures 2.38, 2.39 and 2.40 we see an increase in active servers for all heuristics in the 50 server case over the 60 zone case. Greedy-Z is able to produce assignments using the minimum number of servers for numbers of total servers.

![Average Active Servers (25 Sims)](image)

**Figure 2.35.** Average number of active servers, group C, 60 zones.
Figure 2.36. Average failure fraction, group C, 60 zones.

Figure 2.37. Average runtime, group C, 60 zones.
Figure 2.38. Average number of active servers, group C, 80 zones.

Figure 2.39. Average failure fraction, group C, 80 zones.
Figure 2.40. Average runtime, group C, 80 zones.

greater than 50. GBP is the best of the variants in the 50 server case, and along with GBP-Skip+Fill uses fewer than half the number of servers it required in parameter group A to achieve a successful zone-server assignment. For greater than 50 servers, GBP-Fill and GBP-Skip+Fill reduce active servers used and use fewer than GBP.

Overall for parameter group C we see that lowering the pQoSZ requirement allows us to produce assignments with fewer active servers, in less or the same time, and with fewer failures. SetCover-Z experienced a drastic reduction in failure fraction, Greedy-Z rarely failed, and no GBP variant failed. The performance benefits offered by GBP-Fill and GBP-Skip+Fill over GBP that we observed in groups A and B were not seen here until number of zones was increased to 60, where we observed that increasing total number of servers past 100 caused GBP-Skip+Fill, and sometimes GBP-Fill, to use fewer active servers
than GBP. The gap in active servers used between Greedy-Z and the GBP variants was much smaller here than in previous groups.

2.5.4.3.5 Parameter Group D (Hotspot Parameter Effect Probes)

Under the final parameter group, we alter the hotspot parameters from parameter group A to observe their effects on heuristic performance. We change the HSD from 25% to 50%, so there are an equal number of players in hotspot zones and cool zones, and we change HSF from 0.1 to 0.4. Table 2.6 shows the average number of players per hotspot zone and cool zone for each choice of number of zones. Capacity is the same as group A, so the minimum number of servers possible remains 2. These choices of HSF and HSD are more relaxed – the difference in average players per hotspot zone and cool zone is smaller than group A. We expect this to lead to assignments using fewer active servers and lower failure fractions. Fewer players per zone should make pQoSZ easier to satisfy so a larger number of servers will satisfy a larger number of zones.

Table 2.6. Average number of players per hotspot and cool zone for parameter group D.

<table>
<thead>
<tr>
<th>Number of zones</th>
<th>Number of hotspot zones</th>
<th>Average players per hotspot zone</th>
<th>Number of cool zones</th>
<th>Average players per cool zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>8</td>
<td>312</td>
<td>12</td>
<td>208</td>
</tr>
<tr>
<td>40</td>
<td>16</td>
<td>156</td>
<td>24</td>
<td>104</td>
</tr>
<tr>
<td>60</td>
<td>24</td>
<td>104</td>
<td>36</td>
<td>69</td>
</tr>
<tr>
<td>80</td>
<td>32</td>
<td>78</td>
<td>48</td>
<td>52</td>
</tr>
</tbody>
</table>

We begin with the 20 zone case for parameter group D displayed in Figures 2.41, 2.42 and 2.43. Comparing Figure 2.42 with Figure 2.6, we see in the 50 server case that
Figure 2.41. Average number of active servers, group D, 20 zones.

Figure 2.42. Average failure fraction, group D, 20 zones.
each of the GBP variants in group D uses fewer active servers. Greedy-Z and SetCover-Z experience no reduction in active servers used. SetCover-Z’s failure fraction increases over group A, while Greedy-Z’s is reduced and GBP’s remains the same. No other heuristic fails.

When we increase the number of servers to 100 or more, we see reductions in active servers for all three GBP variants as we have seen in previous cases. We also see that just as in group A, for 150 and 200 servers GBP uses fewer active servers than GBP-Fill or GBP-Skip+Fill. The runtimes do not change significantly between groups A and D.

Figures 2.44, 2.45 and 2.46 show the 40 zone case for parameter group D. In the 50 server case we observe a reduction in active servers used by GBP and GBP-Skip+Fill against the 50 server case of parameter group A in Figure 2.9. GBP-Fill uses a larger number of servers, however. All GBP heuristics use a larger number of servers in this set as opposed
Figure 2.44. Average number of active servers, group D, 40 zones.

Figure 2.45. Average failure fraction, group D, 40 zones.
Figure 2.46. Average runtime, group D, 40 zones.

to the 20 zone case of group D. Two of the three GBP variants, Greedy-Z and SetCover-Z experience failures in this case. SetCover-Z's failure fraction is smaller than its corresponding group A results, however it is larger than the 20 zone group D case.

Increasing servers to 100 decreases the number of servers used by the GBP variants, and GBP uses the fewest servers of the three variants in this case as well as the 150 and 200 server cases. The servers used by each heuristic in these cases is approximately the same or slightly fewer than their corresponding group A cases. Runtime is approximately the same for all heuristics compared to their group A counterparts, with some minor fluctuations.

Figures 2.47, 2.48 and 2.49 represent the 60 zone case for group D. The 50 server case shows a decrease in servers used as opposed to their group A counterparts in Figure
Figure 2.47. Average number of active servers, group D, 60 zones.

Figure 2.48. Average failure fraction, group D, 60 zones.
Figure 2.49. Average runtime, group D, 60 zones.

2.11, apart from GBP-Skip+Fill. GBP and GBP-Fill use approximately the same number of servers as the 40 zone case of group D, but GBP-Skip+Fill sees a large increase in number of active servers used. SetCover-Z, Greedy-Z and GBP-Fill experience an increase in failure fraction over their group A counterparts.

Increasing servers to 100 and beyond decreases servers used in general, as we have seen throughout these simulations. GBP’s servers used increases in the 200 server case, but still uses slightly fewer servers than its group A counterpart. While GBP is the best of the variants in the 50 and 100 server cases, GBP-Fill and GBP-Skip+Fill outperform it in the 150 and 200 server cases, as we have seen before.

In our final set of results for the chapter, Figures 2.50, 2.51 and 2.52 show the 80 zone case for group D. For 50 servers, the heuristics generally use fewer active servers than their group A counterparts, but a larger number than the 60 zone, group D case. As we have
**Figure 2.50.** Average number of active servers, group D, 80 zones.

**Figure 2.51.** Average failure fraction, group D, 80 zones.
Figure 2.52. Average runtime, group D, 80 zones.

seen in the previous parameter groups, increasing the servers beyond 100 causes GBP-Fill and GBP-Skip+Fill to use fewer servers than GBP. GBP-Skip+Fill uses a larger number of servers in the 200 server case over the 150 server case, which is uncommon, but still uses fewer than GBP.

For group D as a whole we observe that although in a few cases the number of active servers increased over their group A counterparts, in general they decreased as we expected. Runtime appears generally unaffected by altering the hotspot parameters. Of the GBP variants, GBP is a good general choice for the 20 and 40 zone cases for most choices of number of servers. It is also generally the best of the three in the 50 and 100 server cases for 60 or 80 zones, but once we increase the total number of servers to 150 or 200, GBP-Fill and GBP-Skip+Fill become the best choices. We speculate this is because for a greater number of zones and a greater number of servers, the benefits of filling a server before
choosing a new one becomes more apparent because it is more likely that multiple unsatisfied zones exist that can be added to a server once chosen by the heuristic.

### 2.5.4.4 Confidence Intervals

Figure 2.53 shows example data to display the statistical margin of error for the data we report in our offline CAP-Z simulations. The graphs display the 95% confidence interval for the 40 zone, 3000 capacity, 0.9 pQoSZ, 0.1 HSF, 25% HSD case. We show the margin of error for the average active servers measure for GBP, GBP-Fill, and GBP-Skip+Fill. The margin of error for this example case is relatively small, save for the 50 server case, particularly for GBP-Fill.

![Average Active Servers](image)

**Figure 2.53.** 95% Confidence Interval for 40 server, 3000 capacity, 0.9 pQoSZ, 0.1 HSF, 25% HSD case.

In the observed simulations, there were a few outliers where the number of servers was very high – nearly 30 or more – leading to this relatively large margin of error. The 50
server case was also a tougher case in general, as fewer server options lead to more server activations to satisfy zones, so we expect a wider variety of results.

2.5.4.5 Final Observations

Greedy-Z and SetCover-Z, when they succeed, use fewer average active servers than all GBP variants. However, SetCover-Z has an unacceptable failure fraction in most cases simulated here, and its active server results occur over a smaller, easier number of runs than the other heuristics, so it is difficult to draw any conclusions about their usefulness or performance. The performance of Greedy-Z comes with a tradeoff for runtime, which is largely dependent on the number of servers and their capacity. There is a clear correlation between increasing number of servers and an increased runtime for Greedy-Z. Decreasing server capacity also leads to an increased Greedy-Z runtime since it drives average active servers up. When server capacity was halved, average active servers was nearly doubled, and so doubled Greedy-Z’s runtime.

This observation helps to show why server capacity should not be considered limitless, as it is in Ta et al.’s simulations. With limitless capacity, the number of active servers needed is reduced, perhaps even as low as one server hosting all zones. In this case the runtime of Greedy-Z would be artificially low. When we introduce capacity, the lower the value we choose, the larger the runtime for Greedy-Z will be because more servers will be required to host all zones. Lower capacity will not affect GBP as largely – the precalculation phase will take longer, but it does not consider interserver connections until later stages so its runtime will not be affected to the degree of Greedy-Z. Zone to server assignment runtime will only be affected if Stages 2 and 3 are required. We verify this in
our group B comparison to group A – GBP variant runtime is not significantly affected by decreasing server capacity.

When the pQoSZ parameter is relaxed, it reduces the difference in active servers used between Greedy-Z and the GBP variants (though the gap does widen as the number of zones is increased). Simply reducing pQoSZ from 0.9 to 0.85 greatly reduces failure fraction. Under this relaxed choice, for fewer zones (20, 40) the adjustments made to GBP for GBP-Fill and GBP-Skip+Fill are almost unneeded – GBP uses fewer active servers than they do. We speculate this to be because there is a large enough number of servers satisfying a large enough number of zones that extra work is unneeded to improve the assignments. Supporting this, we observe that for some parameter sets SetCover-Z produces assignments using the minimum number of active servers just as Greedy-Z does, and in less time because it does not consider interserver connections, which ends up being unnecessary work. We can conclude that for situations where runtime is not a concern and with less demanding parameters (rare chance of failure), the generic SetCover-Z approach is adequate. If runtime is a concern, as in our environment, then for these low demand parameters any of the GBP variants can provide a result close to that of SetCover-Z or Greedy-Z in much less time.

Increasing the total number of servers generally reduces the average active servers used by all heuristics. This is most noticeable between the 50 and 100 server cases of each parameter set shown in this chapter. In demanding environments with few servers from which to choose, a more computationally complex approach such as Greedy-Z will clearly use fewer servers than an approach such as GBP, and the margin between them widens as the parameters become more demanding. However, increasing the number of available
servers narrows the difference in servers used, and the runtime for Greedy-Z increases greatly. The return of decreased number of servers versus time invested obtaining this result becomes very small. In these cases, GBP provides a nice alternative to Ta et al.’s computationally expensive approach.

There is no GBP variant consistently better than any other. However, we have observed some behavioral trends that lead us to use the variants situationally. When servers are few in number and server capacity is small, it is likely we will need to proceed to Stages 2 and 3, so the adjustments made to GBP for GBP-Skip+Fill are beneficial in these situations. GBP-Skip+Fill is also the variant of choice when we have a larger number of zones (60, 80) and a larger number of servers (150, 200). GBP-Fill does well in these situations as well and generally uses fewer active servers than GBP. Under less demanding parameter choices, GBP can produce an adequate assignment and GBP-Fill or GBP-Skip+Fill are not needed, and it generally lends itself to environments with a small number of servers and small number of zones.

We have shown that, in practice, solutions to an offline CAP-Z problem should consider server capacities, and that the runtime of solutions to an uncapacitated version of the problem does not scale well in the capacitated version. We have developed three heuristics to improve upon the results of Ta et al.’s heuristics and while there is not one clear “best” heuristic, they each perform well under specific criteria, and could be chosen for an application based on the pre-specified known criteria for best performance. We seek in future work to adapt these heuristics to an online environment in which we would reassign zones and players as they move around the game world so that the game remains
in QoS throughout the course of gameplay. We will also adapt the principles and learnings from this work to the area of load balancing.
3. Centralized Load Balancing

3.1 Load Balancing Introduction

As players move around the game world in MMOGs, the processing and computation time of players’ actions incurs a load on each server hosting the zones within which the players move. Depending on which servers are responsible for which areas of the game, one server may have a higher load than another, and can even become overloaded if the server is responsible for enough players performing enough actions. Additionally, an overloaded server slows response time to players. Therefore, it is important to keep the load of the system balanced so that no one server becomes overloaded. An overloaded server can hinder performance of the game or cause the game to fail entirely. In load balancing, the active servers of the system shuffle around computation of player actions or other game processes so that no one server has significantly more load than another. They do these load balancing operations incrementally, checking after each load-changing action, or periodically, checking for imbalance at regular intervals. Additionally, they do this balancing in a centralized manner or in a distributed manner. When balancing load in a centralized manner, one controller has knowledge of the loads of all servers in the system, so it can balance loads relative to the entire system. In a distributed load balancing approach, each server acts with incomplete knowledge of the system, balancing load relative only to local knowledge, such as servers adjacent to it in a graph representation of the system. The specific local load balancing problem and approach dictates the amount of knowledge held by a server.

As load balancing a system is NP-Complete [46], we develop novel heuristics that monitor the balance of loads and bring an unbalanced system back into balance. We will
describe both a centralized load balancing heuristic and a distributed load balancing heuristic that each balance load subject to many of the same constraints as our offline CAP-Z problem, such as pQoSZ and server capacity. Additionally, we create some additional constraints that allow us to perform these actions periodically rather than incrementally, and measure performance of each against adapted approaches in existing literature.

3.2 Centralized/Distributed Load Balancing Literature Review

Chen et al. [9] proposed a locality-aware dynamic load management scheme for load balancing. Their approach proposes to balance load in a system while maintaining zone contiguity, monitoring servers for QoS violations and performing load shedding and/or aggregation as necessary. The preservation of zone contiguity among zones on a server aims to reduce communication overhead necessitated by passing data between servers hosting adjacent zones. Preservation of zone contiguity is an interesting concept that we researched during conception of methods for load balancing, but ultimately abandoned because the limitations imposed by pQoSZ were too strict to maintain zone contiguity. However the other methods of load shedding proposed by the authors served as a good basis for comparison for our load balancing heuristics and we ultimately adapted their approach to a heuristic against which to compare our work. We go into more detail about their work and this adapted heuristic in Section 3.5.3.

Deng and Lau [16][17] proposed both a distributed and a centralized approach to load balancing for online games utilizing principles of heat diffusion. They sought to develop an approach that suffered neither from the slow execution times of exhaustive methods nor the lower effectiveness of faster methods. They determined their approach to be both effective in reducing servers from overloading while also being efficient, i.e., having
faster runtimes. Their basic approach used heat diffusion principles to develop a diffusion coefficient that would help to determine the maximum amount of load that could be shed from an overloaded or highly-loaded server to its neighbor servers, without negatively affecting the shedding servers' load such that it become out of balance with those neighbors. Their particular method of distributed load balancing proved to be adaptable into our problem investigating a distributed load balancing method, and we used it as a comparison heuristic against which to measure our performance. We go into more detail on their approach in Section 4.3.1.

Bezerra et al. [3] propose the use of a kd-tree for partitioning a game environment into zones and dynamically adjusting those zones based on player distribution. The authors built a kd-tree corresponding to the initial partitioning of the game world into zones. After initial partitioning and creation of the kd-tree, their heuristic distributes players among zones based on the load incurred by each player on his or her respective server. This incurred load is estimated by the number of other players that will need to be updated about the actions of that player. When a server hosting one of these zones becomes overloaded, the authors' algorithm adjusts the split coordinates of the zones to bring the system back into balance. It does so by starting at the node in the kd-tree associated with the server, then traversing up the kd-tree until it finds a node whose server has capacity greater than or equal to the load, or the root of the tree if no such node exists, collecting load and player data along the way. Once this is found, it adjusts the split coordinates of that node to distribute the players according to the capacity of the node’s children. In simulating this heuristic, the authors use players moving throughout a game world and choose hotspots to which the players move with some percentage of probability. They
attributed the performance of it against other heuristics to the “necessary contiguity” of regions perpetuated by the kd-tree structure, reducing the number of migrations required as well as rebalancing operations. Although this approach deals strictly with balancing load and not with satisfying other factors such as pQoSZ, there are several similarities to our work – the partitioning of the game world into regions using kd-trees, as well as using hotspots to simulate areas of the game world with increasing player activity.

Ta and Zhou [45] proposed a load sharing algorithm to transfer load from an overloaded server to other non-overloaded servers. They performed this load sharing in a distributed manner. When an overloaded server is identified, it will attempt to shed load down to a safe threshold. It will first check its immediate servers as destinations for this load. If it cannot shed enough load to its immediate neighbors to get it to the safe threshold, it will then check its neighbors’ neighbors as destinations for load. It continues to do this until all of the load has been shed or all neighbors have been checked. As this heuristic is load sharing and not load balancing, it is not directly related with our research, but it adds credibility to heuristics that maintain loads within or under thresholds rather than viewing them as minimizations.

Bezerra et al. [4] proposed four load balancing algorithms to allocate load on servers proportionally and to reduce interserver communication overhead. Their heuristic investigates load balancing relative to local information, and it balances bandwidth rather than processing power. Each server checks regularly for imbalance and triggers the algorithm when needed. The system is heterogeneous, with servers not each having the same capacity or computation speed. The algorithms can split zones into smaller subdivisions for finer tuning of balance.
The first algorithm, proportional greedy region growing algorithm (ProGReGA), takes a group of zones as its input and balances load by beginning with the highest cost zones to be balanced and assigning them to the servers with the largest capacity, proceeding in descending order of zone cost. The fraction of load moved to each server should be equivalent. The second algorithm, ProGReGA-KH (ProGReGA keeping heaviest cell), is an extension of the first that attempts to mitigate player disconnects caused by the large number of migrations in ProGReGA. When the original rebalances load, it removes all zones from their assigned servers and reassigns them, which can cause an interruption in gameplay for players. The modification implemented in ProGReGA-KH keeps the highest cost cell within its original region to reduce reassignments. While this modification can reduce reassignments, it also might prevent a good balancing result since the number of migration options is reduced. Additionally, after multiple splits, zones can become very fragmented in this way. The third heuristic, ProGReGA-KF, allows zones to split off their lowest cost subdivisions until the zone’s cost relative to the cost of the other zones to balance is less than or equal to the load on the server relative to the load on the other servers being balanced. Proceeding in ascending order of cost allows for the heaviest subdivisions to remain on their same servers and thus reduces migrations. This also suffers from the same pitfall as ProGReGA-KH in that zones can become fragmented. The final heuristic proposed is an alternative to the ProGReGA method called best-fit based cell transference, or BFBCT. This looks for opportunities to move subdivisions of zones to other servers that have capacity as close to the cost of the zone as possible, without intentionally overloading any server. The most ubiquitous heuristic ended up being ProGReGA-KF, as it
migrated players the second fewest number of times among the algorithms and distributed load generally well.

The authors’ work shares a few similarities with ours. Like the authors, we approach zone reassignment in descending order of cost in portions of our approach. We also allow for zones to be subdivided, however, we do place a limit on the number of divisions in order to reduce the overhead problem mentioned by the authors. The authors do not balance relative to QoS or pQoSZ, and they allow for a subdivision of load that is too fine for our approach, so their method is not applicable to our problem. However, many of the general ideas and motivations behind their heuristics match up with our own.

Chertov and Fahmy [10] investigated a load balancing method that 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 zones. Zones 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 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. This provides existing motivation for implementation of server capacity in a system and dynamic reallocation of players in zones during the course of gameplay. However, while we do some repartitioning, i.e., zone splitting, during our load balancing work, generally we work with static zone partitions throughout the course of our research.
Kim and Park [29] developed a scheme to assign users to servers in order to reduce interserver communication. Players nearer to one another will need to update one another about their current status and actions, and so it is beneficial to place them each on the same server to reduce update messages sent between servers. They used a centralized load balancing scheme that redistributes players to servers based on their proximity to the server handling the zone within which they are located, both at the beginning of the game and then periodically throughout. Although they do not assign subject to QoS or propose a method of redefining zones if a server becomes overloaded by the number of players in the zone it’s handling, the periodic update and action approach is one that we implement in our own methods.

Lau [30] proposed a hybrid load balancing approach that implemented a distributed load balancing heuristic augmented by seldom-accessed global information to improve its performance. It used global information to estimate the direction in which lightly loaded servers were located in order to shed load from overloaded servers in the direction of those lightly loaded servers, instead of using local information only. When choosing a neighbor to which to shed load, an overloaded server would also consider how close that neighbor was located to the area of server load availability to choose to whom to send the load. The author hypothesized that over time the load would make its way to the underloaded area and improve load balance. This heuristic deals primarily with load shedding and not load balance so it is not applicable to either of our load balancing problems, but their distributed method sought to make local changes in the interest of having a global effect, and that fundamental idea lies at the core of our own distributed approach.
Denault et al. [15] balanced load by separating the game world into triangles that can grow or shrink depending on the load incurred on the server hosting them. This process uses two sets of calculations. The first is interest management, which determines the objects or players in which each player is interested depending on his or her state, i.e., the things on which the player needs to be updated when his or her state changes and the things that need to be updated when the player’s state changes. The second is update dissemination, which informs the objects to which a player subscribes during interest management about the player’s state change. Update dissemination occurs at the client level, but servers perform interest management, and that is where the authors’ approach executes. Although their particular method is unrelated to our work, it provides justification for an approach that dynamically allocates load as players move throughout the game world.

Dietrich and Banik [20] proposed an infrastructure in which a group of virtual servers, or pieces of software responsible for processing zone operations, is mapped to physical servers and is responsible for balancing the load among those servers. The authors developed a heuristic to create a multicast tree of these virtual servers for each zone that will guarantee QoS for users within the zones. Their environment is different than ours in a few ways – it allows for more than one contact server, it always connects clients to the closest geographical server, and it activates and deactivates servers as needed. They assume load to be automatically redistributed to compensate for the addition or loss of servers as well. One very important commonality is that the authors establish a QoS value that must be respected, and reiterate the value of 150 ms – a value that we use throughout the course of our work. Ultimately, although the authors’ work is primarily concerned with
construction of the framework of an online game and creation of a tree, which is unrelated to our work, it provides additional evidence for our QoS value assumption, separating contact and target servers, and additional justification for partitioning a game world into regions.

Farooq and Glauert [25] proposed a zone partitioning scheme called Joint Hierarchical Nodes based User Management (JoHNUM) for performing zone splits on overloaded servers. Additionally, they aggregated zones on servers that are underloaded. Although this deals more primarily with load shedding than load balancing and it is unclear what the execution times of these repartitioning schemes are, which would determine whether they are realistic to use in online games in response to crowding or hotspots, we utilize the idea of reactive zone partitioning and aggregation on servers that are overloaded in our load balancing work. This existing approach provides justification to our approach. They extend this work [23] to perform more exhaustive region aggregation, called Aggregate Region Assignment (ARA), for improved load balancing. They further extended the work [24] and applied it to extend the architecture of an open source framework for virtual worlds called OpenSim [26].

Hu et al. [27] investigated a problem they referred to as a joint optimization of view consistency and load balancing. They partitioned a game world into zones such that the load deviation among those partitions was within a specified threshold and then assigned those zones to servers such that the client-server delays were minimized. Although their work is similar to ours in that it assigns zones to servers subject to a view consistency restriction that is similar to our QoS, they restricted zone-to-server assignments such that a server may host only one zone, and they sought to minimize client-server delays where our
work does not. Additionally they repartitioned the world subject to a load deviation threshold, while we move static zones based on player satisfaction within those zones. Their approach is fast in comparison to the heuristics against which they evaluated, and gives some comparison values for parameter choices for our load balancing simulations.

Morillo et al. [34][35] proposed a load balancing scheme that responds to overloaded servers by spreading the overload amount among non-overloaded servers. An interesting addition to their work is the observation that response time in a system increase non-linearly as players join, that is to say, that as a system becomes more loaded the response time increases non-linearly, and so just one overloaded server, or several highly loaded servers, can cause performance degradation beyond what might be expected. This is a consideration that could be an interesting extension for future work. Additionally, their load balance occurs within a threshold of values, such that there is a maximum and minimum threshold between which server loads should lie. This concept is one that we've adapted and applied in our own work (LBF).

Ng et al. [37] developed a virtual walkthrough system called Cyberwalk to model the fundamentals of an online game (users moving around zones hosted by servers). They proposed an adaptive zone partitioning technique to partition zones in response to changes in load. If a server becomes overloaded, it can partition a zone it hosts and transfer some of the load to another server. Additionally, underloaded servers can aggregate zones to increase their loads. We use this basic idea in our centralized load balancing work to conditionally split and combine zones.

Chan et al. [8] proposed k-way spectral graph-based partitioning methods that generalized the ratio-cut cost metric of existing ratio-cut partitioning methods and
provided a lower bound on this cost metric. Although our research is not in the area of graph partitioning methods, when researching methods of partitioning the game world graph into zones during our load balancing work, we considered this paper as a potential method for the partitioning. Ultimately its costly operations and implementation difficulty made it a poor candidate. Partitioning schemes were also considered by [19][21][28][31][33][39].

3.3 Centralized Load Balancing Background

In a massively multiplayer game, several servers work together to host the game world, and as players move around it, the processing and computation time of players’ actions incurs a load on each. Depending on which servers are responsible for which areas of the game, one server may have a higher load than another and can even become overloaded if the server is responsible for enough players performing enough actions. Additionally, an overloaded server slows response time to players. Therefore, it is important to keep the load of the system balanced so that no one server becomes overloaded. An overloaded server can cause a performance hindrance in the game or cause the game to fail entirely.

In load balancing, the active servers of the system shuffle around computation of player actions or other game processes so that no one server has significantly more load than another. One can do these load balancing operations incrementally, checking after each load-changing action, or periodically, checking for imbalance at regular intervals. Additionally, one can do this balancing in a centralized manner or in a distributed manner. When balancing load in a centralized manner, one controller has knowledge of the loads of all servers in the system, so it can balance loads relative to the entire system. In a
distributed load balancing approach, each server acts with incomplete knowledge of the system, balancing load relative only to local knowledge, such as servers adjacent to it in a graph representation of the system. The specific local load balancing problem and approach dictates the amount of knowledge held by a server.

As load balancing a system is NP-Complete [46], we develop novel heuristics that monitor the balance of loads and bring an unbalanced system back into balance.

Load balancing is related to our work in CAP-Z, in that it has a set of clients, set of servers, set of zones, value for QoS, and value for pQoSZ. We assign the zones containing the clients to servers subject to pQoSZ satisfaction and server capacity. One change from CAP-Z, however, is that instead of selecting active servers from a larger pool of servers to satisfy the zones, in load balancing the set of active servers is an input and does not change. We also perform operations online while the game is running.

Likewise, we define the zones in the game world when the game begins and they do not change unless load balancing operations split or merge them (this will be discussed in further detail later). As players move throughout the game world, their assigned zone changes when they cross a zone boundary. If a player crosses into a zone that is hosted by a different server, then his or her assigned server will change, and thus his or her incurred load will transfer to that server. As gameplay ensues and players move around the game world, the load on the servers responsible for hosting each zone will increase or decrease according to the player distribution. This shift in player distribution, and thus in server load across the system, can cause the load on an individual server to become disproportionate with respect to the average load per server. A system in this state of disproportionate load among active servers is called out of balance. Two extreme cases in
an out of balance system are server overloading and server underloading. An overloaded server is one that has taken on more computational load than its resources allow for it to handle. An underloaded server is one that has too little (even zero) computational load to process to warrant the upkeep required to keep the server active. Disproportionate balance among active servers requires action to shift load around and bring the system back into balance, and as part of this we should take measures to reduce load on overloaded servers or add load to underloaded servers.

When defining a balanced system, one common criterion is to keep all servers’ load at the same value (say, all servers 30% loaded). In this problem, all load balancing operations center around moving units of load, sometimes very small or fractional, among the existing servers to maintain a specific load on each server. While this system can provide a very precise result, it is not applicable to our problem because of the very fine details of load movement. In our problem, we move load around by zones at a time, and while we can divide zones to bring load closer to a value, they are not infinitely divisible – thus achieving an equal load value may be impossible.

We propose a method of balance based on the delta of load between the highest loaded server and the lightest loaded server, called the load gap. We define a maximum threshold for load gap, called Load Balancing Factor (LBF), and adjust loads among servers when the load gap exceeds the threshold. Thus, load balancing operations hinge on this LBF. If the load gap is less than the LBF, then we consider the system to be in balance. Only if the load gap has exceeded the LBF do we perform load balancing operations.

In our load balancing operations, we reduce the load on the highest loaded server and increase the load on the lightest loaded server, all while being mindful of the load gap –
movements of load in the system should serve to decrease the load gap only, never to increase or maintain it. We will refer to the movement of zones to the lightest loaded server as *zone adding* and we will refer to the movement from the highest loaded server as *zone shedding*. If at any point we can no longer make such a move, then we cease load balancing operations, even if the load gap is still larger than the LBF. We wait until the next interval check to measure the two values again and determine whether further load balancing operations are needed. Players moving and forming different hotspots bring the system back into balance naturally. Additionally, as player distribution within zones changes, servers that may not have satisfied pQoSZ for a zone may later satisfy that zone (and vice versa), so the options can change for moving zones among servers at the next interval check.

In these systems there also may exist zones that cannot be satisfied by a single server for pQoSZ reasons, or that are simply too large to be hosted by a single server due to capacity constraints. We implement a method of zone splitting that allows for a server that is overloaded to split its largest zone so that the subsequent zones can be moved off to other servers. We also use this method to split the largest zone on the highest loaded server in the cases where initial load balancing operations cannot move any load from that server. We can later recombine the zones into their original zones when able to be accommodated by a single server again.

Through methods such as zone adding/shedding and zone splitting we seek to reduce the load gap in out of balance systems to within LBF. In order to reduce runtime, however, we cease load gap reduction attempts when load gap is less than or equal to LBF or after a prespecified number of tries per breakpoint if the load gap has not reduced to
LBF. We will discuss these methods in detail in the next section, justify a periodic approach to their use, and then simulate them to determine their effectiveness in load gap reduction.

3.4 Centralized Load Balancing Method

While an incremental approach to load balancing can provide a greater level of control over the response to changes in the system, the overhead required to maintain such a system is large. We must measure the effects of each movement by each player, and the position of each player is important – one player crossing one zone boundary could cause that zone’s current server to no longer satisfy it, requiring migration of that zone to a new server. In addition to zone satisfaction, we would need to closely monitor load on the servers to ensure load gap remained within LBF. Small changes in player location could affect this and again prompt action that may not be needed after a few more cycles of movement back and forth across a zone boundary.

To counteract this potential problem, we follow a periodic approach to load balancing. We do not make changes based on each incremental movement, instead defining periodic breakpoints during which load balancing operations take place. During these breakpoints, we check for any zones whose assigned server causes it to violate pQoSZ and migrate that zone if necessary, then we measure load gap. If the load gap is greater than LBF, then we will attempt to move zones to reduce the load gap to within LBF; if the load gap is less than or equal to LBF, then we make no changes. Investigating these conditions only during breakpoints eliminates the need for constant oversight of each change in load that occurs from natural player movement during gameplay. It also helps to reduce potential overcorrection (moving large amounts of load for little or no benefit) that might result from small changes in player distribution/composition or load gap. For example, one
player crossing a zone boundary could cause the zone it enters or the zone it leaves to no
longer satisfy pQoSZ on the server hosting it, which would warrant transferring the zone to
a different server. During the time that it takes to transfer the zone, the player could move
back into his or her original zone, making the zone transfer ultimately unnecessary.
Although performing these checks only during breakpoints does not eliminate this issue
entirely, it will reduce the number of times each individual zone can be migrated.

Load balancing is a better approach to respond to load changes in a breakpoint-
based system such as this, rather than other methods that redistribute load in response to
server overloading. Load distribution can change significantly between breakpoints, so
methods that only avoid overloading would be more susceptible to servers becoming
overloaded between breakpoints as they take no proactive action to redistribute load.
Conversely, load balancing methods would distribute load more evenly among servers
during breakpoints and as far from overload as possible, so it would be less likely for a
server to become overloaded between breakpoints.

During load balancing breakpoints, we calculate the current load data for each
server. With this data, we can easily identify the highest loaded server and the lightest
loaded server and calculate the load gap difference of their loads. If we find that the load
gap has exceeded LBF, we then begin load balancing operations to try to bring load gap
back within LBF. We propose three sets of actions to help balance load in our system: zone
adding/shedding, proactive zone splitting, and reactive zone splitting. Additionally, as
partner operation to zone splitting we employ split zone rejoining. We refer to these
collective operations as BreakpointLB.
3.4.1 Zone Adding/Shedding

Zone adding/shedding (ZAS) will check for opportunities to shed zones from the highest loaded server to other servers that can accommodate them. Correspondingly, it will also check the lightest loaded server for opportunities to move zones from other servers onto it. We assess the zones of the respective servers for movement possibilities, where cost is the total number of players positioned in the zone. We proceed in descending order of cost when attempting to move zones from the highest loaded server. ZAS will alternate between these two operations, one zone at a time, until one of two conditions are met:

- Load gap is within LBF, or
- Ten attempts to move a zone from the highest loaded server and move a zone to the lightest loaded server.

If the latter condition causes ZAS operations to cease, we accept that load gap might still be greater than LBF in order to refrain from negatively affecting gameplay as a result of excessive computation.

The checks and operations required to add zones or shed zones must maintain two conditions.

1) A ZAS move must never increase the load gap

It is possible to shed load from the highest loaded server and increase the load gap accidentally. Consider the situation when the highest loaded server and the second highest loaded server have loads that are close together in value, say 40% for the former and 38% for the latter. If the highest loaded server sheds a zone that reduces its load by 5%, moving that zone to the second highest loaded server, then the second highest loaded server would
become the new highest loaded server at 43% load, thus increasing the load gap. We consider this to be an illegal move.

Similarly, load added to the lightest loaded server can increase the load gap. For example, consider the case where the lightest loaded server is 10% loaded and the second lightest server is 13% loaded. The lightest loaded server takes a zone with 5% of the load from the second lightest server. This results in the second lightest server becoming the new lightest loaded server with 8% load, thus increasing the load gap of the system.

2) pQoSZ must be respected

Not only must assignments be made respective to reducing the load gap to within LBF, but any reassignment must also comply with pQoSZ. We do not add load to any server if it will violate pQoSZ – that requirement supersedes proportionate load among servers. This is because while disproportionate load does not fall in line with our requirements for a balanced system, unless the server is overloaded it will not directly affect performance like a pQoSZ violation would. Thus, meeting pQoSZ is more important.

3.4.2 Proactive Zone Splitting

Proactive zone splitting is a measure taken only when ZAS can move no zone. In these cases, presumably the highest loaded server has one or more large zones for which satisfying pQoSZ is difficult or the zone has grown too large to be satisfied by a single server due to capacity constraints, or the server hosts a zone or zones satisfiable by few or no other servers. In these cases our system would normally fail to provide any balance, since in its existing state it would be impossible to move zones that would reduce load gap significantly enough to bring it back within LBF. In these cases, we take proactive measures to create more opportunities to move load around in later iterations. Proactive zone
splitting (PZS) will split the most costly zone on the highest loaded server in half by area but leave the two resulting zones assigned to that server. This allows ZAS operations in later iterations to consider these additional options for shedding. Note that we may split an individual zone at most twice. We institute this bound to prevent excessive zone splitting.

While zone splitting is helpful in the short term, it can cause long-term problems if performed too many times. Smaller zones with fewer players can be more difficult to satisfy pQoSZ for – if pQoSZ is 0.9 and a zone has five players in it, then all five players must be satisfied by a server or pQoSZ will not be satisfied for the zone – satisfying only four players would be 0.8 satisfaction and thus not satisfy pQoSZ. Also, as the number of zones increases, any operation performed on all zones will require more iterations, such as the sorting performed in some of the ZAS operations we implement. This hinders performance of those operations and can thus negatively affect gameplay. We should make a conscious effort to control the overall number of zones, only splitting when necessary and rejoining as soon as it makes sense to do so (see Split Zone Rejoining).

When splitting a zone, PZS will separate the zone at its boundary midpoint (the direction of which is an implementation detail to be explained later) and create two new zones, assigning each player to one of the two new zones based on his or her position within the original zone. There are rare circumstances in which pQoSZ will not be satisfied by the hosting server for one of the new zones. Consider the case where pQoS is 0.9 and a server S’s most costly zone, Z, contains 100 players. Let QoS be satisfied by S for 95 of the players within the zone, and not satisfied by S for the remaining 5 players. pQoS is satisfied, since the satisfaction fraction is 0.95, which is greater than pQoS (0.9). Let 80 of the satisfied players reside to the left of Z’s midpoint, and the remaining 15 satisfied
players, along with the 5 unsatisfied players, reside to the right of Z's midpoint. Now, let us assume PZS is performed on Z. Let the new zone to the left of the midpoint of Z be referred to as Z_a and the new zone to the right of its midpoint be referred to as Z_b. For Z_a, the satisfaction fraction will be 1.0, as the original server satisfies all players within the zone. However, for Z_b, the satisfaction fraction is 0.75, as the original server does not satisfy 5 of the 20 players within the zone. In this case, the original server will not satisfy Z_b. PZS will search for a server to satisfy this new zone in that case.

Instead of choosing the split point as the boundary midpoint, we could choose it as the population midpoint, in which half of the players in the zone would be contained in each segment. However, determining this split point would be computationally expensive. We would need to find each player’s position in the zone and then sort all players by x (or y) coordinate. As this split operation is always performed on the largest zone on the highest loaded server, we will generally have a large number of players on which to perform this operation, causing this sort to be costly. Additionally, separating the zone in this way could still result in the circumstance of dissatisfaction described above, so the additional computation would not alleviate that issue. For these reasons, we avoid splitting the zone by population.

3.4.3 Reactive Zone Splitting

We perform reactive zone splitting (RZS) during breakpoints in response to a server becoming overloaded. (Note that the only way for a server to become overloaded is by a server’s assigned zones naturally increasing in cost as players move throughout the game world and into those zones.) In order to prevent performance degradation for players in zones belonging to these servers, the system responds by splitting the most costly zone in
the same manner as PZS. However, in these cases we should avoid leaving the newly created zones assigned to the original server, as that server would remain overloaded. RZS will attempt to assign these new zones to servers, ensuring that the load gap is not increased as a result of the movement. (Note that the original hosting server is still a potential landing spot for one of the new zones.) This operation also can decrease the load gap, as the set of overloaded servers will always contain the highest loaded server, and the reassignment of the new zones created from its most costly zone will bring its overall load down. If RZS cannot move these new zones off of the overloaded server, then they will remain on the overloaded server and can be split further by later RZS operations or moved off by subsequent ZAS operations.

3.4.4 Split Zone Rejoining

As proactive and reactive zone splitting are both temporary measures made to help decrease a recalcitrant load gap, we seek opportunities later to rejoin those zones back together. There are two conditions to qualify a previously split zone for rejoin:

- rejoining two zones must never increase the load gap or overload a server, and
- a zone may only be joined back to its original partner zone in order to recreate the original parent zone.

We check for these opportunities at each breakpoint, performing them unless the zones were the product of a reactive zone split at that breakpoint. This is so a split performed on an overloaded server gets at least one full breakpoint cycle for changes to propagate before attempting recombination. We search for split zones and check to see if any can be recombined with its original partner zone, assigning it to the former’s server. A recombined zone is eligible for future splitting if necessary.
3.4.5 Breakpoint Pseudocode

We define periodic breakpoints during gameplay in which load balancing operations take place. We may attempt each of ZAS, RZS, and PZS during these breakpoints. Once we attempt these, we then attempt to recombine a zone split during a previous breakpoint. The following pseudocode shows the framework of the operations that occur during each such breakpoint. The pseudocode for each of ZAS, RZS, and PZS follows, along with functions required by each.

<table>
<thead>
<tr>
<th>Breakpoint Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
</tr>
<tr>
<td>( PL ) – list of all ( p ) players = {pl_0, pl_1, \ldots, pl_{p-1}}. Each player element ( pl_i ) contains zone, the identity of the player’s assigned zone, and coords, their location in the game world.</td>
</tr>
</tbody>
</table>
| \( Z \) – list of all \( n \) zones = \{z_0, z_1, \ldots, z_{n-1}\}. Each zone element \( z_i \) contains:
| - server, the identity of the zone’s assigned server;
| - cost, the load incurred by the zone on a potential server;
| - players, a list of all players assigned to the zone. The size of this list varies by zone and can be at minimum 0 and at most \( p \) for a single zone. The total size of all such lists across all elements of \( Z \) will be \( p \). |
| \( SL \) – list of all \( m \) active servers = \{sl_0, sl_1, \ldots, sl_{m-1}\}. Each server element \( sl_i \) contains:
| - zones, a list of all of the zones assigned to the server. The size of this list varies by server and can be at minimum 0 and at most \( n \) for a single server. The total size of all such lists across all elements of \( S \) will be \( n \);
| - load, the quantity of load incurred by all zones assigned to the server. |
| \( servCap \) – maximum capacity of load a server can take on before becoming overloaded. This value is the same for every server. |
| \( QoS \) – value for QoS |
| \( pQoSZ \) – value for pQoS |
| \( LBF \) – value for LBF |

<table>
<thead>
<tr>
<th><strong>Variables</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( flag ) – holds result of ZAS to trigger PZS if necessary</td>
</tr>
<tr>
<td>( highSrv ) – highest loaded server. output from ZAS and input to PZS if called (does not change between these two functions)</td>
</tr>
<tr>
<td>( tmpZones, tmpLoad ) – used during update of server loads as temp variables</td>
</tr>
</tbody>
</table>

Breakpoint Pseudocode, (cont.)
tmpSrv – temporary variable used during satisfaction of unsatisfied servers

Outpus

SL – servers updated with new zone-to-server assignments
Z – zones updated with splits and new server assignments
PL – updated with new player-to-zone assignments

Method

//rediscovers each player’s zone – zone changes are not tracked between breakpoints
for a< 0 to sizeof(PL) – 1
    updateZone(pla) //update the player’s assigned zone based on coords –
        //remove from previous zone.players and add to new if necessary

//update loads on servers after zone updates
for d<0 to sizeof(SL) – 1
    sla.load <- 0
    for e<-0 to sizeof(sla.zones) – 1
        sla.load <- sla.load + sla.zones[e].cost

//if a zone is no longer satisfied by its server, then reassign it to a new server
for b< 0 to sizeof(Z) – 1
    if(checkQoS(zb, zb.server, pQoS, QoS) == 0)
        for c< 0 to sizeof(SL) - 1
            if(slc.load + zb.cost <= servCap)
                if(checkQoS(zb, slc, pQoS, QoS) == 1)
                    tmpSrv <- zb.server
                    removeZone(zb, tmpSrv)
                    tmpSrv.load <- tmpSrv.load – zb.cost
                    addZone(zb, slc)
                    slc.load <- slc.load + zb.cost
                    break inner loop

RZS(Z, SL, QoS, pQoS, servCap) //check for overloaded servers and split zones as
    //necessary
    //check for opportunities to move load off highest loaded server and to move load
    //to the lightest loaded server, decreasing load gap

flag, highSrv <- ZAS(SL, LBF, QoS, pQoS)
if (flag == 0) PZS(Z, SL, QoS, pQoS, highSrv) //if ZAS moved no load, proactively split zone
    //on highest loaded server to increase movement opportunities in the future

attempt to recombine zones split in previous breakpoints
3.4.6 Function: RZS

RZS checks overloaded servers for opportunities to reduce loads by splitting the most costly zone assigned to them and then reassigning the newly created zones. We can assign those zones to either a new server or the original server, but must conform to typical assignment standards such as server capacity, pQoSZ, etc. While RZS can reduce overloading and the number of overloaded servers, it does not guarantee elimination of overloading.

RZS performs these splits on each overloaded server at each breakpoint. We may not recombine any zone split as a result of RZS during the same breakpoint in which we split that zone.

RZS assigns these zones subject to load gap constraints. We cannot assign these subzones such that they will intentionally overload a different server, so assuming successful reassignment, the result will be a positive for the server and system as a previously overloaded server will be no longer overloaded, or at the very least, less loaded. These reassignments may also result in a load gap reduction if the overloaded server was also the highest loaded server.

We cannot split a zone more than two levels, or in other words, further than into “sub-subzones”. For example, consider a zone to be split called zts. RZS splits zone zts and form two subzones, zts_a and zts_b. Later, it is possible that RZS splits zts_a, forming “sub-subzones” zts_c and zts_d. Neither zts_c nor zts_d can be split further. However, RZS can later split zts_b into zts_e and zts_f. We allow no further subdivisions.
RZS

Inputs

$Z$ – list of all $n$ zones $= \{z_0, z_1, ..., z_{n-1}\}$. Each zone element $z_i$ contains:
- server, the identity of the zone’s assigned server;
- cost, the load incurred by the zone on a potential server;
- players, a list of all players assigned to the zone. The size of this list varies by zone and can be at minimum 0 and at most $p$ for a single zone. The total size of all such lists across all elements of $Z$ will be $p$.

$SL$ – list of all $m$ active servers $= \{sl_0, sl_1, ..., sl_{m-1}\}$. Each server element $sl_i$ contains:
- zones, a list of all of the zones assigned to the server. The size of this list varies by server and can be at minimum 0 and at most $n$ for a single server. The total size of all such lists across all elements of $S$ will be $n$;
- load, the quantity of load incurred by all zones assigned to the server.

QoS – the value for QoS

$pQoSZ$ – the value for $pQoSZ$

$servCap$ – maximum capacity of load a server can take on before becoming overloaded

Variables

$OLzones$ – contains zones assigned to an overloaded server

$highZone$ – highest cost zone on overloaded server

$highZone_a, highZone_b$ – zones created as result of zone split

$haflag, hzbflag$ – is set when the respective zone is assigned

$succesFlag$ – set if subzones added successfully to new servers

Outputs

$Z$ – updated with new server assignment and splits if applicable

$SL$ – updated with new zone-to-server assignments

$PL$ – updated with player zone identities after split

Method

for $i < 0$ to sizeof($SL$) – 1
  if($sl_i$.load > $servCap$) //if server load is greater than maximum capacity
    $OLzones$ <- getZones($sl_i$) //retrieve zones assigned to overloaded server
    $highZone$ <- MaxCost($OLzones$) //retrieve highest cost zone
    //split only if zone has been split fewer than two times. Do nothing
    //otherwise
    if ($highZone$.numSplits < 2)
split highZone in half by area, forming highZone_a and highZone_b
update pl.zone for all players in highZone
remove highZone from sl_i
remove highZone from Z
add highZone_a, highZone_b to Z
sl_i.load <- sl_i.load - highZone.cost
hzaflag <- 0
hzbflag <- 0
successFlag <- 0
//iterate through servers and find one that satisfied one or
//both new zones. Assignments are subject to pQoSZ and
//capacity constraints.
//exit loop once both are reassigned
for j<-0 to sizeof(SL) – 1
  //if new load on sl_b will be less than server capacity,
  //check QoS and assign. Cannot increase load gap since
  //either another overloaded server exists and this
  //assignment will not cause the destination server to
  //become overloaded, or sl_i is the final overloaded
  //server and the reassignment will reduce its load,
  //decreasing load gap
    if(hzaflag == 0 && highZone_a.cost + sl_i.load < servCap)
      if (checkQoS(sl_j, highZone_a, pQosZ, QoS) == 1)
        addZone(highZone_a, sl_i)
        hzaflag <- 1
        sl_i.load <- sl_i.load + highZone_a.cost

    if(hzbflag == 0 && highZone_b.cost + sl_i.load < servCap)
      if(checkQoS(sl_j, highZone_b, pQosZ, QoS) == 1)
        addZone(highZone_b, sl_i)
        hzbflag <- 1
        sl_i.load <- sl_i.load + highZone_b.cost

  //stop once both have been reassigned
  if(hzaflag == 1 && hzbflag == 1)
    successFlag <- 1
    break inner loop

if(successFlag == 0)
  //if we reach here we have not been able to add one or both
  //subzones to another server. add back to sl_i
    if(hzaflag == 0)
      addZone(highZone_a, sl_i)
      sl_i.load <- sl_i.load + highZone_a.cost
    if(hzbflag == 0)

addZone(highZone, sl)

\[ sl_{load} <- sl_{load} + highZone\_b.cost \]

3.4.7 Function: ZAS

ZAS will attempt to shed load from the highest loaded server and take on load to the lightest loaded server in order to decrease the load gap until it is within LBF. It does so by alternating between the aforementioned shed/add operations until the load gap is within LBF, we can shed no load from the high server or add no load to the low server, or ZAS has attempted each of the operations 10 times. ZAS utilizes functions \( lbCheck \) and \( update\_load \), and we describe each in detail later.

**ZAS**

**Inputs**

\( SL \) – list of all \( m \) active servers = \( \{ sl_0, sl_1, ..., sl_{m-1} \} \). Each server element \( s_i \) contains:
- zones, a list of all of the zones assigned to the server. The size of this list varies by server and can be at minimum 0 and at most \( n \) for a single server. The total size of all such lists across all elements of \( S \) will be \( n \);
- load, the quantity of load incurred by all zones assigned to the server.

\( LBF \) – value for LBF (load balancing factor)
\( QoS \) – value for QoS
\( pQoS \) – value for pQoS

**Variables**

\( HS \) – highest loaded server
\( LS \) – lightest loaded server
\( hiFlag \) – flag to signify successful shed from high server
\( loFlag \) – flag to signify successful add to low server
\( numChk \) – counts number of add/shed attempts
\( exitFlag \) – flag to signify end of shed/add attempts
\( maxLoad, minLoad \) – load on highest/lightest loaded servers, respectively
\( servCap \) – maximum capacity of load a server can take on before becoming overloaded
\( secHigh, secLow \) – second highest and second lowest loaded servers, respectively. Used as input to lbcheck
\( HS\_change \) – returned by updateLoad to indicate if we have a new high server, to trigger a sort on \( HS \)'s zones – necessary for high server checks only

ZAS (cont.)
Outputs

SL - updated with any new zone-to-server assignments
returnFlag - indicates if any moves happened
HS – returned to be used as input into PZS if called

Method

exitFlag <- FALSE
returnFlag <- 0
HSchange <- 0
numChk <- 0
HS, LS, secHigh, secLow <- updateLoad(SL)  //initialize these values
maxLoad <- HS.load
minLoad <- LS.load
sort HS.zones in descending order of cost

//iterate until load gap is within LBF or we have checked enough times for moves
while (maxLoad - minLoad > LBF && exitFlag == FALSE)
    hiFlag, loFlag <- FALSE
    //this for loop checks for opportunities to move zones off of the high server
    for a <- 0 to sizeof(HS.zones) - 1
        highZone <- HS.zones_a
        for b <- 0 to sizeof(SL) - 1
            tempSrv <- sl_b
            //if we find a server that's different than the highest loaded
            //server and has capacity, continue
            if(tempSrv != HS && tempSrv.load + highZone.cost <= servCap)
                res <- lbcheck(highZone, tempSrv, HS, QoS, pQoSZ,
                                HS.load, secHigh.load, LS.load, secLow.load, HIGH)
                //if lbcheck passes, remove zone from old server and
                //add to new server
                if(res == 1)
                    removeZone(highZone, HS)
                    addZone(highZone, tempSrv)
                    HS.load <- HS.load - highZone.cost
                    tempSrv.load <- tempSrv.load + highZone.cost
                    HS, LS, secHigh, secLow, HSchange <-
                    updateLoad(SL, HS)
                    if(HSchange == 1) sort HS.zones in descending
                    order by cost
                    maxLoad <- HS.load
                    minLoad <- LS.load
                    hiFlag <- TRUE
returnFlag <- 1
HSchange <- 0
break both for loops

//this for loop checks for opportunities to move zones to the low server
for c <- 0 to sizeof(SL) - 1
  //if the active server to be checked is not the low server, continue
  if(slc != LS)
    tempSrv <- slc
    tempZones <- getZones(tempSrv)
    //iterate over all zones on the potential server to move from
    for d <- 0 to sizeof(tempZones) - 1
      moveZone <- tempZones[d]
      if(LS.load + moveZone.cost <= servCap)
        res <- lbcheck(moveZone, LS, HS, QoS, pQoSZ, HS.load, secHigh.load, LS.load, secLow.load, LOW)
        //if lbcheck pass, remove zone from old server
        //and add to low server
        if(res == 1)
          removeZone(moveZone, tempSrv)
          addZone(moveZone, LS)
          tempSrv.load <- tempSrv.load - moveZone.cost
          LS.load <- LS.load + moveZone.cost
          HS, LS, secHigh, secLow, HSchange <- updateLoad(SL, HS)
          //LS could have become new high while
          //reducing load gap
          if(HSchange == 1) sort HS.zones in
            descending order by cost
          maxLoad <- HS.load
          minLoad <- LS.load
          loFlag <- TRUE
          returnFlag <- 1
          HSchange <- 0
          break both for loops
numChk++;
//exit if we cannot make a move on either end of the load gap or we have
//done ten consecutive iterations with movement to limit the number of
//incremental improvements
if(hiFlag == FALSE && loFlag == FALSE) exitFlag <- TRUE
else if(numChk >= 10) exitFlag <- TRUE
return returnFlag, HS
3.4.8 Function: PZS

PZS will proactively split the largest zone on the highest loaded server if ZAS performs no adds or sheds during a breakpoint. The intention is to help future breakpoints by creating more opportunities for sheds from the highest loaded server to decrease the load gap.

Although in theory the split would be “in-place”, in implementation PZS creates two new zones to replace the original split zone. As a result, we need to add these new zones back to the original hosting server and they are subject to the same requirements as any zone-to-server add. The distribution of players in that zone could potentially cause a scenario in which the original hosting server cannot accommodate one of the new zones because of pQoSZ not being satisfied. In this case, PZS will reassign the zone to a new server.

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
</table>

\( Z \) – list of all \( n \) zones = \( \{z_0, z_1, ..., z_{n-1}\} \). Each zone element \( z_i \) contains:
- \( server \), the identity of the zone’s assigned server;
- \( cost \), the load incurred by the zone on a potential server;
- \( players \), a list of all players assigned to the zone. The size of this list varies by zone and can be at minimum 0 and at most \( p \) for a single zone. The total size of all such lists across all elements of \( Z \) will be \( p \).

\( SL \) – list of all \( m \) active servers = \( \{sl_0, sl_1, ..., sl_{m-1}\} \). Each server element \( s_i \) contains:
- \( zones \), a list of all of the zones assigned to the server. The size of this list varies by server and can be at minimum 0 and at most \( n \) for a single server. The total size of all such lists across all elements of \( S \) will be \( n \);
- \( load \), the quantity of load incurred by all zones assigned to the server.

\( QoS \) – the value for QoS
\( pQoSZ \) – the value for pQoSZ
\( highSrv \) – highest loaded server
**Variables**

- **highSrvZones** – contains all zones assigned to **highSrv**
- **splitZone** – zone to be split – highest cost zone on **highSrv**
- **splitZone_a, splitZone_b** – zones created as a result of zone split
- **addRes** – result of addZone function to determine if adding new zones back to original server was successful
- **res** – result of lbcheck (success or failure)
- **successFlag** – set if zones successfully reassigned

**Outputs**

- **SL** – updated with new zone-to-server assignments if applicable
- **Z** – updated with zone splits if applicable

**Method**

```plaintext
highSrvZones <- getZones(highSrv) //retrieve the zones assigned to the highest loaded server
splitZone <- MaxCost(highSrvZones) //retrieve highest cost zone
//only split if zone has been split fewer than two times. Otherwise do nothing
if (splitZone.numSplits < 2)
    split splitZone in half by area, forming splitZone_a and splitZone_b
    update pl.zone for all players pl in splitZone
    add splitZone_a to Z
    add splitZone_b to Z
    remove splitZone from Z and highSrv //remove old zone from Z and its server
highSrv.load <- highSrv.load - splitZone.cost
//If the server does not have capacity for the zone or does not satisfy QoS for it, find //another server to assign to that will satisfy pQoSZ. We do not restrict //assignments to within load gap to reduce the possibility of not finding a place to //assign the zone
if(highSrv.load + splitZone_a.cost > servCap || checkQoS(splitZone_a, highSrv, pQoSZ, QoS) == 0)
    for i<-0 to sizeof(SL) - 1
        if(checkQoS(splitZone_a, sl, pQoSZ, QoS) == 1)
            addzone(splitZone_a, sl)
            sl.load <- sl.load + splitZone_a.cost
            successFlag <- 1
            break loop
//if we arrive here we could not find a place for the zone, so simply //add it anywhere to be handled by future iterations
if(successFlag == 0) add splitZone_a without satisfaction to any server with capacity
```

PZS (cont.)
else
    addZone(splitZone_a, highSrv)
    highSrv.load <- highSrv.load + splitZone_a.cost
    successFlag <- 0
    if(highSrv.load + splitZone_b.cost > servCap || checkQoS(splitZone_b, highSrv, pQoSZ, QoS) == 0)
        for j<-0 to sizeof(SL) – 1
            if(checkQoS(splitZone_b, sl, pQoSZ, QoS) == 1)
                addzone(splitZone_b, sl)
                sl.load <- sl.load + splitZone_b.cost
                successFlag <- 1
                break loop
            else
                if(successFlag == 0) add splitZone_b without satisfaction to any server with capacity

        else
            addZone(splitZone_b, highSrv)
            highSrv.load <- highSrv.load + splitZone_b.cost

3.4.9 Function: lbCheck

This function checks if a server will satisfy QoS for a zone and ensures that adding the zone will not increase the load gap. Note: when calling this function when cur_check is HIGH, the server input to this function will be a server other than the highest loaded server (looking for a server to which we shed one of the highest loaded server’s zones). The zone input will be one of the highest loaded server’s zones. Alternatively, when calling this function when cur_check is LOW, the server input to this function will be the lightest loaded server. The zone input will be a zone belonging to a server other than the lightest load server (looking for a zone to add to the lightest loaded server).

Function: lbCheck

Inputs
input_zone – a single zone to check
input_server – a single server to check
high_server – highest loaded server
QoS – the value for QoS

Function: lbCheck (cont.)
pQoSZ - the value for pQoSZ

high_server.load, sec_high.load - the loads on the highest loaded (and second highest loaded) servers, respectively, before this potential move is made
low_server.load, sec_low.load – the loads on the least loaded (and second least loaded) servers, respectively, before this potential move is made
cur_check – specifies if we are looking at shedding load from the highest loaded server or moving load onto the lightest loaded server

Outputs

retval - 1 if the server will satisfy the zone. 0 otherwise

Variables

source_server, dest_server – variables used for readability/understandability when calculating current and potential load differences.
keepGoing – tells us whether to continue or quit during checks throughout the function

Method

retVal <- 0

//server is a server other than highest loaded server
//zone is a zone from highest loaded server
if (cur_check == HIGH)
  source_server <- high_server
  dest_server <- input_server

  //if input_server == sec_high, input_server.load + input_zone.cost will always
  //be higher than sec_high.load (since input_server.load == sec_high.load) so that
  //will not break this operation. Further down, we check whether this would
  //increase our boundary, so no need to redundantly check here.
  new_high_load <- max(input_server.load + input_zone.cost, high_server.load -
                      input_zone.cost, sec_high.load)

//server is lightest loaded server
//zone is zone from server other than lightest loaded server
else if (cur_check == LOW)
  source_server <- input_zone.server
  dest_server <- input_server //input_server == low_server
  //similar to above, if source_server is the same as sec_low, then
  //source_server.load – input-zone.cost will always be less than sec_low.load.

Function: lbCheck (cont.)
new_low_load <- min (source_server.load - input_zone.cost, low_server.load + input_zone.cost, sec_low.load)

if(dest_server.load + input_zone.cost < high_server.load && source_server.load - input_zone.cost > low_server.load)
    keepGoing <- TRUE
else keepGoing <- FALSE

if(keepGoing == TRUE)
    retVal <- checkQoS(input_zone, input_server, pQoSZ, QoS)

return retVal

3.4.10 Function: update_load

This function takes SL as an input and outputs the values of high_server, low_server, sec_high and sec_low. The lbcheck function utilizes sec_high and sec_low.

Function: update_load

Inputs

SL – list of all m active servers = {sl0, sl1, ..., slm-1}. Each server element sl contains:
- zones, a list of all of the zones assigned to the server. The size of this list varies by server and can be at minimum 0 and at most n for a single server. The total size of all such lists across all elements of S will be n;
- load, the quantity of load incurred by all zones assigned to the server.
oldHighServ – taken as input to compare against final result. If it changes, will help ZAS trigger a new zone sort

Outputs

high_server, low_server, sec_high, sec_low – highest loaded server, lowest loaded server, second highest loaded server and second lowest loaded server, respectively
res – 0 if high server did not change, 1 otherwise

Method

res <- 0

Function: update_load (cont.)
```c
//initialize tmpHigh, tmpSecHigh
if sl0.load > sl1.load
    tmpHigh <- sl0
    tmpSecHigh <- sl1
else
    tmpHigh <- sl1
    tmpSecHigh <- sl0

//iterate through the remainder of the servers. If a server's load is greater than both the
//highest and second highest loaded server, it becomes the new highest and the former
//highest becomes second highest. Otherwise if it is higher than only the second highest, it
//becomes the new second highest.
for i <- 2 to sizeof(SL) - 1
    if sl.load > tmpSecHigh.load
        if sl.load > tmpHigh.load
            tmpSecHigh <- tmpHigh
            tmpHigh <- sl
        else
            tmpSecHigh <- sl
    else
        tmpLow <- sl
        tmpSecLow <- tmpLow

//initialize tmpLow, tmpSecLow
if sl0.load < sl1.load
    tmpLow <- sl0
    tmpSecLow <- sl1
else
    tmpLow <- sl1
    tmpSecLow <- sl0

//iterate through the remainder of the servers. If a server's load is lower than both the
//lowest and second lowest loaded servers' respective loads, it becomes the new lowest
//and the former lowest becomes second lowest. Otherwise if it is lower than only the
//second lowest, it becomes the new second lowest.
for i <- 2 to sizeof(SL) - 1
    if sl.load < tmpSecLow.load
        if sl.load < tmpLow.load
            tmpSecLow <- tmpLow
            tmpLow <- sl
        else
            tmpSecLow <- sl
    else
        tmpSecLow <- sl

//return all of the final values
high_server <- tmpHigh
sec_high <- tmpSecHigh
low_server <- tmpLow
sec_low <- tmpSecLow
```

Function: update_load (cont.)
if (oldHighSrv != high_server)

    res<-1

return high_server, sec_high, low_server, sec_low, res

3.4.11 Function: checkQoS

Function checkQoS takes a single zone and a single server as input and determines if that server will satisfy pQoSZ for that zone.

Function: checkQoS

Inputs

z – A single zone to determine QoS for. Contains PL, an array of all players in the zone along with their delays to each server.
s – A single server to check QoS for
pQoSZ – fraction of players for which we need to achieve QoS to satisfy a zone QoS – QoS threshold value

Variables

numQoS – number of players in zone with QoS

Output

1 if s will achieve QoS for z based on pQoSZ, 0 otherwise

Method

if (sizeof(z.PL) == 0) return 0
numQoS <- 0
for i<-0 to sizeof (z.PL) - 1
    if delay(z.PLi, s) ≤ QoS
        numQoS++
if((numQoS / sizeof(z.PL)) ≥ pQoSZ)
    return 1
else return 0

In the next section, we will describe the work of Chen et al. [9], whose work will serve as comparison for these heuristics, as well as our simulator and simulation results.
3.4.12 Time Complexity Analysis

We derive the time complexity for BreakpointLB as follows. We summarize the critical path of operations for each portion of BreakpointLB, derive the time complexity for each, and then determine the total time complexity for the heuristic.

- **RZS:** Let \( m \) denote the total number of active servers in the system and let \( p \) denote the total number of players in the system. For each overloaded server \( q \), split the highest cost zone \( r \) assigned to \( q \) into \( r_1 \) and \( r_2 \), then update the assigned zone for each player in the original zone. Finally, for each active server, call \( \text{checkQoS}() \) on \( r_1 \) and \( r_2 \) to check for satisfaction and assign each subzone to a server. The time complexities for these individual operations follow.

  - The time complexity of the split of the original zone into subzones is \( O(1) \).
  - Define \( s \subseteq p \) where \( s \) contains the players whose assigned zone is \( r \). The update operation for the assigned zone for each player in \( s \) takes \( O(p) \) time, as in the worst case \( s = p \).
  - As shown previously, \( \text{checkQoS}() \) runs in \( O(p) \) time for a single server.

In the worst case we call \( \text{checkQoS}() \) on each other active server in the system after splitting the zone. This means that across all \( \text{checkQoS}() \) calls in RZS we test QoS for each player in the game on each server, so its total time complexity is \( O(mp) \). The time complexity for RZS on a single server is thus \( O(1 + p + mp) \), or \( O(mp) \). In the worst case, every server in the system is overloaded, so we perform an RZS split on one zone for each active server \( m \). Therefore, the time complexity for RZS is \( O(m^2p) \).

- **ZAS:** For each zone \( k \) on each of the highest and lightest loaded server, ZAS iterates through the active servers and calls \( \text{lbCheck}() \) on each other active server for zone
satisfaction. It then reassigns \( k \) if it finds a server that satisfies the zone. If ZAS reassigns the zone successfully, it calls \( \text{updateLoad}() \), which updates the identities of the highest loaded, second highest loaded, etc. active server in the system. ZAS then sorts the zones in descending order by cost. The time complexities for the individual operations follows.

- \( \text{lbCheck}() \) calls \( \text{checkQoS}() \) if the server it is checking meets the other requirements for zone satisfaction. These \( \text{checkQoS}() \) calls dominate its time complexity, which is \( O(p) \) for a single server.

- \( \text{updateLoad}() \) iterates through each active server to update the identities of the highest, second highest, etc. servers and its time complexity is \( O(m) \).

- The zone sort is a modified mergesort that takes \( O(k \log(k)) \) time. \[40\]

In the worst case, for each zone, ZAS calls \( \text{lbCheck}() \) on each active server, then performs \( \text{updateLoad}() \) and the sorting operation once each. The time complexity for the ZAS operations for one zone is \( O(mp + m + k\log(k)) \). \( p = m \times \text{server capacity} \times \text{average system load} \), therefore \( p >> m \). We can also safely assume \( p >> k \), otherwise the average number of players per zone would be very small and satisfying each zone would be impossible or extremely difficult. Therefore, this time complexity simplifies to \( O(mp) \). Further, in the worst case, every zone lies on the highest loaded server and we try to shed each of them to a different server, so we perform this operation \( k \) times. The total time complexity for ZAS is thus \( O(mpk) \). For sufficiently large values of \( p, m \approx k \), therefore we rewrite this complexity as \( O(m^2p) \).

- PZS: PZS begins by finding the highest cost zone \( r \) on the input server, which is always the highest loaded server. It splits \( r \) in half by area, then assigns each player
within that zone to one of the new subzones formed by the split. Finally, if the original hosting server cannot accommodate the zone, PZS looks for an active server to which to assign these new subzones, calling checkQoS() to check for QoS satisfaction. The individual time complexities for these operations follows.

- The time complexity to find the highest cost zone is $k$.
- The time complexity of the zone split is $O(1)$.
- Define $s \subseteq p$ where $s$ contains the players whose assigned zone is $r$. The update operation for the assigned zone for each player in $s$ takes $O(p)$ time, as in the worst case $s = p$.
- As shown previously, checkQoS() runs in $O(p)$ time for a single server.

In the worst case, following the split the original hosting server cannot accommodate $r$, and PZS checks each active server for satisfaction. Across all checkQoS() calls, this makes the total time complexity of PZS $O(1 + k + p + mp)$, or $O(mp)$ as $p \gg k$.

In BreakpointLB’s worst case it performs each of RZS, ZAS and PZS once (note that PZS only runs if ZAS moves no zones). The total time complexity is then $O(m^2p + m^2p + mp)$. The time complexity of BreakpointLB thus simplifies to $O(m^2p)$. Note that we would expect this worst case to occur rarely, if ever. We would not expect to design a system in which it was possible for every server to be overloaded at once (maximum number of users exceeding maximum capacity of the system), so it is not likely that situation would occur.

Additionally, load balancing operations would occur frequently enough that it would not be possible for all servers to become overloaded during the period between breakpoints, so
presumably any overloaded server cases would be addressed as they happen and not allowed to accumulate.

3.5 Centralized Load Balancing Simulation

3.5.1 Introduction to Simulator

In order to measure the performance of BreakpointLB, we have created a simulator to mimic the joins, leaves, and movements of players in a MMOG and balance load among the available servers using our algorithms. Much like our simulator used in our offline CAP-Z work, we generate an \( n \times n \) grid. Each adjacent index in the array corresponds to 1 ms of delay, and the Euclidean distance between any two points (players or servers) determines the delay between them. For example, given a client position of [2][3] and a server position of [2][4], the client would have a 1 ms delay to that server.

We create the zones in each run of the simulator before the simulator begins. We begin with a square simulated game space, for example, 512x512. We will create zone boundaries by choosing a random segmentation point in alternating directions until the original square has been divided into a number of smaller squares equivalent to our desired number of zones. We organize the zone information in a kd-tree structure, and we maintain this structure even when zones are split using RZS or PZS.

In order to utilize a kd-tree and its operations we must always choose a number of zones that is a power of 2. We then perform \( \log_2(\text{number of zones}) \) splits on the original grid to give us the correct number of zones. Each split operation chooses a random segmentation point for each subdivision to prevent zone uniformity. For example, if we desire 16 zones, then splitting a square four times (alternating left-to-right splits and top-to-bottom splits) will accomplish this. The first iteration would divide a square into two
rectangles. The second iteration would divide the two rectangles into four rectangles. The third iteration would divide the four rectangles into eight rectangles. Finally, the final iteration would divide the eight rectangles into sixteen rectangles.

Once the game world is created, an initial number of players joins the game world while no load balancing operations take place. We refer to this as the ramp-up phase. Once the initial players have joined, an initial assignment of zones to servers takes place. We assign each zone to an active server with sufficient capacity that satisfies pQoSZ for that zone. We collect no statistics during this time – players move around the game world freely with no regard for changing load. Once the ramp-up phase ends, we start the sampling phase, in which data is collected and load balancing operations take place. New players join during this time and existing players may leave at any time. Players continue to move throughout the game world during the course of the sampling phase. As the sampling phase continues, we define load balancing breakpoints in which the controller takes into account current data and performs load balancing operations if necessary, as described in Section 3.4. Once the sampling phase is over, we report all collected data for the session.

3.5.2 Simulator Parameters

The simulator has several input parameters that we can adjust to measure the effects on statistical outputs.

- Physical world size
- Virtual world size
- Number of active servers
- Server capacity
- Average system load
- IS delay factor
- Number of zones
- Initial player count
- Ramp up cycles
Physical world size is the size of the simulated physical world grid in which servers and players exist. Virtual world size is the size of the virtual world in which player avatars exist and where the zone partitioning is made. Number of active servers is the total number of servers to which a player can connect and that may host zones. Server capacity is the maximum load any one server can incur. Capacity is the same for all servers, though we can adapt the load balancing algorithm to differing server capacities. Average system load is the average load among all active servers. We use this input to deterministically choose our average system load to gauge heuristic response to low, medium, and high loaded systems. IS delay factor is the ratio of interserver connection delays to player-server connection delays. Number of zones is the number of partitions to the game world we will make that players can move among during the course of gameplay. Initial player count is the number of players that join the game during the ramp-up phase. Ramp up cycles is the number of game cycles in which players will join the game with no leaves or load balancing operations taking place. Sample cycles is the number of cycles in which players may move, join, or leave the game and load balancing breakpoints and operations take place. QoS value is the maximum delay for a player connection to a server for the player to be satisfied. pQoSZ value is the fraction of players in a zone that must be satisfied in order for the zone to be satisfied. LBF value is the maximum difference between the highest loaded server’s load and the lightest loaded server’s load, within which the system is considered to be balanced.
We express LBF values as a percentage of server capacity. *Number of breakpoints* is the number of times during the sampling phase in which we will perform load balancing operations. *Number of hotspots* is the number of predefined hotspot zones in which players will congregate in order to simulate areas of interest in the game, such as dungeons or towns. *Hotspot fraction* is the fraction of the total number of players that will congregate in hotspots.

We next describe the work of Chen et al. [9] and our adaptation of their approach into comparison heuristics.

### 3.5.3 Chen et al.'s Load Balancing Approach and Adaptation into ChenLB

Chen et al. [9] proposed a locality-aware dynamic load management scheme for load balancing. Their approach proposes to balance load in a system while maintaining zone contiguity, monitoring servers for QoS violations and performing load shedding and/or aggregation as necessary. The preservation of zone contiguity among zones on a server aims to reduce communication overhead necessitated by passing data between servers hosting adjacent zones.

As players move throughout the game world, they pass through adjacent zones, each hosted by a server. Two different physical servers may host two zones that are adjacent in the game world. As the player passes between these two zones, the two servers hosting the zones must communicate so that the new server has the player information it needs to satisfy him or her, as well as to update other players that are located in that same zone, but no such communication happens when the same server hosts both zones. If each server hosts contiguous zones, then less communication overhead arises as players move among zones.
During the course of gameplay, each server monitors for QoS violations and triggers action based on whether that violation was caused by excessive client load or excessive interserver communication. Violations due to excessive client load trigger load shedding, while violations from excessive interserver communication trigger load aggregation. Load shedding attempts to shed load from an overloaded server, first targeting neighboring servers to the overloaded server as the destination and then servers with known lightest load. The heuristic triggers load aggregation on servers that violate a QoS value on excessive interserver communication, combining multiple contiguous zones onto a single server to reduce that interserver communication. Both shedding and aggregation seek to preserve or increase the number of contiguous zones hosted by a single server, but are not necessarily limited by preserving this contiguity – they may shed to non-neighbor servers if needed.

There are differences between Chen et al.’s operating environment and our own. The authors do not make assignments subject to pQoSZ, nor do they adhere to a load balancing range such as LBF. Instead, they reference two thresholds, safe thr and light thr. Load shedding uses safe thr to signify the threshold beyond which a server becomes overloaded and needs to trigger load shedding operations. When shedding load, the heuristic sheds with preference to neighbor servers whose loads lie below light thr. If it is necessary to shed load to a non-neighbor server, then the heuristic triggers load aggregation on that server to increase contiguity. Despite these differences, the load shedding and aggregation approach of Chen et al. is comparable to our own zone adding and shedding in ZAS, and we see merit in adopting the authors’ methods into our operating environment in order to compare heuristic performance.
In order to adapt the heuristics of Chen et al. into our operating environment, we will preserve their load shedding and aggregating methodology, converting their inputs and variables into equivalent choices for the inputs into our problem. We will then use their methods to balance load if required by the system and record the number of balanced breakpoints. We will refer to this collection of modified heuristics as ChenLB.

We make the following changes to convert Chen et al.’s ideas to use them on our load balancing problem as ChenLB.

- Instead of being system inputs, ChenLB will compute safe_thr and light_thr from the value of LBF relative to the input of average system load. The values will be as follows:
  - safe_thr = average system load + (LBF / 2)
  - light_thr = average system load – (LBF / 2)
  - Example: Let average system load be set to 25% (0.25) and LBF be set to 0.3. Thus: safe_thr = 0.4 (40% loaded), light_thr = 0.1 (10% loaded).

- From the computations above, ChenLB sets the load gap goal values as static between the computed values of safe_thr and light_thr, instead of dynamically changing as the loads of the highest loaded and lightest loaded servers change. If any server either exceeds the safe_thr value or dips below the light_thr value, this will trigger load shedding or aggregation, respectively.

- Interserver communication is not a measure relevant to this load balancing problem. Therefore instead of triggering load aggregation due to excessive interserver communication, ChenLB will trigger load aggregation on servers whose
load is below $light_{thr}$ in an attempt to bring their loads above $light_{thr}$ and within $LBF$.

- ChenLB will trigger load shedding and aggregation only during breakpoints. When a breakpoint starts, if any server loads are above $safe_{thr}$ then ChenLB will perform load shedding on those servers, and likewise if any server loads lie below $light_{thr}$, then it will perform load aggregation on those servers.

- The core functionality of Chen et al.’s heuristics remains unchanged in ChenLB. We are simply modifying their heuristic to work within the confines of our load balancing problem.

We describe ChenLB in detail in the pseudocode below.

### 3.5.3.1 ChenLB Shedding Functions

**Function: ShedLoad**

**Purpose**
This function is called on servers whose load exceeds $safe_{thr}$. It attempts to shed load to neighboring servers via the $ShedToNeighbors$ function, and then if that fails, to shed load to non-neighbors via direct calls to the $HeuristicGraphPartition$ function.

**Inputs**
- $s_i$ – server to shed load from
- $safe_{thr}$ – value for $safe_{thr}$
- $light_{thr}$ – value for $light_{thr}$
- $pQoSZ$ – value for $pQoSZ$

**Variables**
- $s_k$ – used to hold the identity of non-neighbor servers during iteration
- $flag$ – flag for result of $HeuristicGraphPartition$ to signify success during non-neighbor sheds

**Output**
servers with updated zone assignments if success; otherwise fail

Function: ShedLoad(cont.)
Method

flag <- ShedToNeighbors (si, safe_thr, pQoSZ)
if (flag == 0)
    for all non neighbor servers sk
        if (sk.load < light_thr)
            HeuristicGraphPartition (si, sk, safe_thr)
            if sk.load < safe_thr
                flag <- 1
                break

if (flag == 0) return fail

Function: ShedToNeighbors

Purpose
This function calls HeuristicGraphPartition repeatedly on neighbors of the input server until its load is below safe_thr or we have exhausted the available neighbors to check.

Inputs
si – input server being shed from
safe_thr – value for safe_thr
pQoSZ – value for pQoSZ

Variables
sj – used during iteration to store neighbor
a – iterator for while loop

Output
1 if load successfully brought below or equal to safe Thr, 0 otherwise

Method

a <- 0
while (si.load > safe_thr && a < sizeof(si.neighbors))
    s_j <- si.neighbors_a
    HeuristicGraphPartition(si, s_j, safe_thr, pQoSZ)
    a++
if (si.load ≤ safe_thr) return 1
else return 0
Function: HeuristicGraphPartition

**Purpose**
This function seeks to move load from one server to another while maintaining or improving zone hosting contiguity. It first checks for all pairs of adjacent zones each hosted separately by the potential source and destination server, respectively. These are identified as boundary zones that can improve contiguity by hosting more contiguous zones on the same server. Once identified, we check pQoSZ and capacity restrictions for these zones on the destination server. Any that satisfy these restrictions have a BFS performed from them to see if any additional contiguous zones can be moved along with them to the destination server.

**Inputs**
- $s_i$ – source server
- $s_j$ – destination server
- $\text{safe\_thr}$ – value for safe\_thr
- $\text{pQoSZ}$ - value for pQoSZ
- $H_i$ - the subgraph of zone graph $H$ containing only vertices and edges pertaining to zones hosted by server $s_i$

**Variables**
- $Y_i$ – collection of boundary zones to check to shed on the source server
- $z_k$ – iterator for zones hosted on source server
- $z_g$ – iterator for zones hosted by destination
- $z_y$ – iterator for boundary zones

**Output**
servers updated with any new zone-to-server assignments

**Method**
- $Y_i \leftarrow \text{null}$
- for all zones $z_k$ hosted by $s_i$
  - for all zones $z_g$ hosted by $s_j$
    - if $z_k$ adjacent to $z_g$ and $H_i.z_k$ unmarked
      - $Y_i += z_k$
      - break inner loop

- for all zones $z_y$ in $Y_i$
  - if ($s_j$ satisfies $\text{pQoSZ}$ for $z_y$ & $s_j.load + z_y.cost \leq \text{safe\_thr}$)
    - BFS ($H_i$, $s_i$, $s_j$, $z_y$)
  - if ($s_i.load \leq \text{safe\_thr}$) break
Function: BFS

Purpose
BFS performs a breadth-first search on the neighbor graph of the input zone to find contiguous zones (neighbors) that can also be moved to the destination server along with the input zone.

Inputs
$H_i$ - the subgraph of zone graph $H$ containing only vertices and edges pertaining to zones hosted by server $s_i$
$s_i$ – input server - source
$s_j$ – input server - destination
$z$ – BFS is run on neighbor graph of this input zone

Variables
$Transfer_{ij}$ – collection of zones to be transferred from server $i$ to server $j$
$BFSqueue$ – collection of zones for performing breadth-first search

Output
applicable zones added to $Transfer_{ij}$

Method
$BFSqueue$ <- null
$Transfer_{ij}$ <- $z$
add $H_i.z$'s unmarked neighbors to $BFSqueue$
while $(s_i.load - Transfer_{ij}.cost > safe_thr && BFSqueue$ not empty)
    $tmpZone$ <- dequeue next zone in $BFSqueue$
    add $tmpZone$'s unvisited neighbors to $BFSqueue$
    if $(s_j$ satisfies $pQoSZ$ for $tmpZone && tmpZone.cost + Transfer_{ij}.cost + s_j.load \leq safe_thr)$
        $Transfer_{ij} += tmpZone$
move all zones in $Transfer_{ij}$ from $s_i$ to $s_j$
mark all zones in $Transfer_{ij}$ as moved in $H_i$
add zones in $Transfer_{ij}$ to $H_j$

3.5.3.2 ChenLB Aggregation Functions

Function: HeuristicGraphMerge

Purpose
HeuristicGraphMerge is the primary function for load aggregation. It is called in two cases: if a shed is made to a non-neighbor server during load shedding operations, and on any servers whose loads lie below $light_thr$ during a breakpoint.

Function: HeuristicGraphMerge (cont.)
**Inputs**
s_f – the non-neighbor server that received a shed zone, or underloaded server
H_f – the subgraph of zone graph H, containing only vertices and edges pertaining to zones hosted by server s_f
safe_thr – the value for safe_thr
pQoSZ – the value for pQoSZ

**Variables**
s_g – the neighbors of the input server s_f
z_k, z_j – iterator for zones hosted by s_f and s_g, respectively
Y_f – boundary zones that are candidates for aggregation
z_x – iterator for zones in Y_f
Transfer_fg – zones to be transferred from s_f to s_g. Used as input into BFSagg, which performs the transfer if needed

**Outputs**
Servers updated with new zone assignments, if applicable

**Method**
Y_f <- null
for all neighbors s_g of s_f
   if boundary(s_g) < boundary(s_f) & load(s_g) < safe_thr
      for all zones z_k in s_f.zones
         for all zones z_j hosted by s_g
            if z_k adjacent to z_j & H_f.z_k unmarked
               Y_f += z_k
               break inner loop
res <- 0
for all zones z_x in Y_f
   if (s_g satisfies pQoSZ for z_x & z_x.cost + s_g.load ≤ safe_thr &
      boundary (s_g.zones ∪ z_x) ≤ boundary (s_f.zones - z_x))
      Transfer_fg <- z_x
      res <- BFSagg (H_f, z_x, s_f, s_g, Transfer_fg, pQoSZ, safe_thr)
   if (res == 1) break for loop

**Function: BFSagg**

**Purpose**
BFSagg performs a breadth-first search on the subgraph H_f beginning with the input vertex H_f.z. If all the neighbor zones discovered during the breadth-first search can be transferred to the destination server along with the input zone z while keeping the load on the destination server below safe_thr, then they will be transferred. Otherwise the transfer operation fails and no zones are transferred.

Function: BFSagg (cont.)
Inputs

- $H_f$ - the subgraph of zone graph $H$, containing only vertices and edges pertaining to zones hosted by server $s_f$
- $z$ – zone eligible for transfer from $s_f$ to $s_g$ to check boundary zones for transfer
- $s_f$ – source server for zone transfers
- $s_g$ – destination server for zone transfers
- $\text{Transfer}_{fg}$ – collection of zones to be transferred from $s_f$ to $s_g$
- $\text{pQoSZ}$ – value transferred from $s_f$ to $s_g$
- $\text{safe}_\text{thr}$ – value for $\text{safe}_\text{thr}$

Variables

- $\text{BFSqueue}$ – used to store the graph vertices to be visited during the BFS
- $\text{tempZone}$ – zone graph vertex corresponding to zone to be checked for satisfaction by destination server

Outputs

- $\text{flag}$ – returns 1 if zones in $\text{Transfer}_{fg}$ successfully transferred; 0 if no zones transferred

Method

1. $\text{flag} <- 1$
2. $\text{BFSqueue} <- \text{null}$
3. add $H_f$'s unmarked neighbors to $\text{BFSqueue}$
4. while $\text{BFSqueue}$ not empty && $\text{flag} != 0$
   - $\text{tempZone} <-$ dequeue next zone in $\text{BFSqueue}$
   - if ($s_g$ satisfies $\text{pQoSZ}$ for $\text{tempZone}$ && $s_g$.load + $\text{Transfer}_{fg}$.cost + $\text{tempZone}$.cost $\leq \text{safe}_\text{thr}$ && boundary($s_g$.zones $\cup$ $\text{Transfer}_{fg}$) $\leq$ boundary($s_f$.zones $-$ $\text{Transfer}_{fg}$))
     - add $\text{tempZone}$'s unvisited neighbors to $\text{BFSqueue}$
     - $\text{Transfer}_{fg} += \text{tempZone}$
   - else $\text{flag} <- 0$
5. if ($\text{flag} == 1$)
   - Transfer zones in $\text{Transfer}_{fg}$ from $s_f$ to $s_g$
   - Mark zones in $\text{Transfer}_{fg}$ as moved in $H_f$
   - Add zones in $\text{Transfer}_{fg}$ to $H_g$
6. else empty $\text{Transfer}_{fg}$
7. return $\text{flag}$

In the following section, we will test the performance of $\text{BreakpointLB}$ against $\text{ChenLB}$ to determine if our heuristics can produce more balanced breakpoints than $\text{ChenLB}$. 
3.5.4 Centralized Load Balancing Simulation Results

To gauge the effectiveness of BreakpointLB, we use experimentally chosen parameter values as inputs to the simulator and measure the number of balanced breakpoints. We partition our results into parameter groups that change a subset of the parameter values to measure that parameter's impact on simulation results and the effectiveness of BreakpointLB in producing balanced breakpoints.

When choosing parameters, we eliminated choices that would be mathematically difficult or impossible to satisfy. To identify these choices, we computed the average zone size, which is the average server load impact for a zone shed or add. We pruned cases in which this impact was greater than or equal to one-half the LBF value. The reasoning is that in these cases, the granularity of zones allows few options, leading to difficulty in finding destinations for zones. Balancing load in these situations will be difficult or impossible regardless of method. In the graphs of the following section, when a set of results is not present, this is because it did not meet the mathematical requirements.

The formula we use is \( L = \frac{(S \times A)}{Z} \), where:

- \( S \) = number of servers,
- \( Z \) = number of zones,
- \( A \) = system load,
- \( L \) = average impact on server per zone operation (referred to as Average Zone Size, or AZS).

For example, if we have 10 servers, 128 zones, and 25% average system load, then on average AZS will be 1.95; that is to say, on average a zone would add or subtract 2% of a server’s load relative to its capacity when moved. Let us compare this to the case of 30 servers, 128 zones, and 75% system load – this produces an AZS of 17.6 – movement of an
average zone will add or subtract nearly 18% to a server’s load relative to capacity. In this case, fine tuning load would be very difficult, especially if we had an LBF of 10 or 15, which is less than AZS.

The following parameter choices are the same among all simulated parameter groups.

- Initial Player Count – 1% of the average number of players based on average system load
  - Number of active servers * server capacity * system load * 0.01
- Physical World Size – 256x256
- Virtual World Size – 256x256
- Ramp Up Cycles – 500
- Sample Cycles – 20000
- QoS – 150
- IS delay factor – 0.6
- Number of breakpoints – 400
- Server capacity – 1500

The following parameter choices will be varied among the listed values.

- Number of servers – 10, 20, 30, 40
- Number of zones – 128, 256
- LBF – 10, 15, 20, 25, 30, 35

We will simulate BreakpointLB as well as ChenLB with identical inputs to measure the total fraction of all breakpoints that ended in balance and the total fraction of all breakpoints that began unbalanced but were brought into balance by the applied load balancing heuristic.
3.5.4.1 Parameter Group A (128 Zones, 25% System Load)

**Figure 3.1.** Results for Parameter Group A, 10 servers.

Beginning with parameter group A, for 10 servers we observe that BreakpointLB is able to bring nearly all breakpoints into balance. We deduce that there is high player churn in this parameter group with 10 servers, since fewer breakpoints stay in balance – often, when we encounter a breakpoint, it needs to be brought back into balance, and BreakpointLB is successful in doing so. ChenLB does not perform as well. As LBF increases, ChenLB is able to bring more of the breakpoints into balance, but never matches the performance of BreakpointLB.

**Figure 3.2.** Results for Parameter Group A, 20 servers.
When the number of servers is increased to 20, we notice a significant decrease in number of breakpoints being brought into balance and total balanced. ChenLB struggles with nearly every breakpoint under stricter choices of LBF. As LBF is relaxed, again we see an increase in breakpoint balance from both heuristics, and BreakpointLB balances over five times as many breakpoints in the 15 LBF case and over twice as many breakpoints in the 20 and 25 LBF cases.

**Figure 3.3.** Results for Parameter Group A, 30 servers.

We continue the trend of lowered breakpoint balance as servers are further increased to 30. Both heuristics balance less than 25% of breakpoints in the 15 LBF case, with ChenLB ending with 10% or less in all but the 35 LBF case. As LBF rises BreakpointLB performs better and nearly acceptably in the 35 LBF case, while ChenLB does not. AZS for this group is 5.85, so smaller values of LBF such as 15 and 20 are tough cases for which to fine tune load, allowing for only a few loads or sheds within which to bring load into LBF.
Figure 3.4. Results for Parameter Group A, 40 servers.

Rounding out Parameter Group A, the 40 server cases prove toughest of all. ChenLB fails to balance any breakpoints in any choice of LBF. BreakpointLB also balances very few breakpoints. AZS for this case is 7.8, and thus far is the most demanding value we've encountered. As we increase LBF, we see more balanced breakpoints, primarily in the 35 LBF case – in this case, with an average system load of 25%, loads should lie on average between 42.5% loaded and 7.5% loaded in order to be balanced. Therefore, an entirely unloaded server needs to have only one average-sized zone assigned to it to be within balance. This may still be difficult depending on player distribution, but still more manageable than stricter LBF choices.

In Parameter Group A we observe that BreakpointLB consistently balances more breakpoints than ChenLB does. We also observe that as the number of servers increases, the total number of balanced breakpoints for both heuristics decreases. Finally, as LBF is increased, the number of balanced breakpoints increases.
3.5.4.2 Parameter Group B (128 Zones, 50% System Load)

**Figure 3.5.** Results for Parameter Group B, 10 servers.

For 10 active servers in Parameter Group B, BreakpointLB balances nearly every breakpoint as it did in Parameter Group A, though we do noticed a greater, but still small, number of breakpoints unbalanced for lower values of LBF. ChenLB suffers a noticeable dip in balanced breakpoints as compared to its Parameter Group A counterparts, with less than half as many breakpoints or more for LBF of 10, 15 and 20. We can again observe that balanced breakpoints increase as LBF increases.

When servers increase to 20 in Parameter Group B, we see that balanced breakpoints decrease slightly for BreakpointLB in comparison to its Parameter Group A counterparts. ChenLB’s balance drastically decreases, going from as high as 70% balanced in the 35 LBF case of Group A down to just over 10% balanced in the corresponding Group B choice.
Figure 3.6. Results for Parameter Group B, 20 servers.

Figure 3.7. Results for Parameter Group B, 30 servers.

Figure 3.8. Results for Parameter Group B, 40 servers.
For choices of 30 and 40 active servers, ChenLB fails to bring any into balance apart from the 35 LBF case with 30 active servers. BreakpointLB performs similarly to its Parameter Group A counterparts in these cases, suffering very little from the additional load brought on by increasing system load from 25% to 50%.

In Parameter Group B, the additional system load causes a drastic drop-off in balanced breakpoints for ChenLB. While BreakpointLB also brings fewer breakpoints into balance, the difference is much smaller and less noticeable in many cases.

3.5.4.3 Parameter Group C (128 Zones, 75% System Load)

![Figure 3.9. Results for Parameter Group C, 10 servers.](image)

Again in parameter group C, we see in the 10 server case that BreakpointLB balances nearly all breakpoints in all cases. ChenLB continues its performance decrease in this case compared to Parameter Groups A and B.

In the 20 and 30 server cases, ChenLB is able to balance only a very small percentage of breakpoints. BreakpointLB increases in performance over its Parameter Group A and B counterparts. Although the larger number of players overall makes this set of parameters challenging, it is here where our methods of PZS and/or RZS can help – because average system load is high, it is much more likely in Parameter Group C than in
Parameter Group A or B that a server becomes overloaded, and so we can split zones to increase our chances of getting load back into balance. Additionally, RZS can also help by splitting zones in the cases where we are not able to get things back into balance to help future breakpoints. These two working in tandem helps BreakpointLB succeed where ChenLB does not.

We provide some sample data to show how much of the overall load balancing effort is performed by our different methods under the low, medium and high choices of average.
system load to validate our above postulation that under higher system load, the methods of PZS and RZS help increase load balancing performance.

**Table 3.1.** Sample of load gap reduction data by system load.

<table>
<thead>
<tr>
<th>Average System Load</th>
<th>Average # RZS splits</th>
<th>Average Load Gap Reduction Per Split</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>1.68</td>
<td>13.87</td>
</tr>
<tr>
<td>75</td>
<td>16</td>
<td>9.79</td>
</tr>
</tbody>
</table>

The data in the table comes from the 20 server, 30 LBF, 128 zone cases for the listed average system load choices. We can observe that in these cases, RZS does nothing in the 25% load case, as we would expect – if average system load is 25%, no server should ever become overloaded. In the 50% case, there are few splits on average – while occasionally a server may become overloaded, it is rare. However, we note that when it does split, RZS is able to reduce server load by nearly 15% in the average case. The greatest increase in number of splits comes in the 75% system load case. The average number of splits increases by nearly an order of magnitude from the 50% case, and although the average load reduction per split decreases, we can see that as RZS is doing a much more significant amount of work in the 75% case than the 25% case, and can make the difference in whether a breakpoint is balanced.

In Parameter Group C we found that BreakpointLB again performed well even when system load was high, likely attributed in part to our implementation of RZS and PZS. ChenLB buckled under the increased load, turning out the fewest balanced breakpoints of groups A, B and C.
3.5.4.4 Parameter Group D (256 Zones, 25% System Load)

Figure 3.12. Results for Parameter Group D, 10 servers.

We now increase the number of zones to 256, which will decrease the average players per zone and should allow for a finer tuning of load and increase success rates compared to the tougher parameter choices of groups A, B and C. This is because the respective AZS values for Groups D, E and F will be 50% of their values for their A, B and C counterparts with the number of zones doubling. In the 10 server case, this translates from BreakpointLB turning out near-perfect performance in Group A to now achieving 100%

Figure 3.13. Results for Parameter Group D, 20 servers.
balance in Group D. We also see an increase in ChenLB’s performance, also balancing all breakpoints in the 30 and 35 LBF cases.

When we increase servers to 20, we again see that breakpoint balance increases across the board when compared to the 20 server Parameter Group A counterparts. While we saw a significant performance decrease between the 10 and 20 server cases of Parameter Group A, in Parameter Group D that decrease is less pronounced, and BreakpointLB still achieves 100% balance in multiple cases here where it did not in Group A.

However, results overall are still acceptable in most choices of LBF, as opposed to the 30 server case of Group A in which nearly all of the choices of LBF resulted in breakpoint balance below 50%. ChenLB’s performance increases significantly over its 30 server Group A counterparts, particularly in the 30 LBF case, rising from 10% to over 70%. BreakpointLB still performs better in this case, however. Note that the increase of zones to 256 also allowed us to simulate a choice of 10 for LBF, and BreakpointLB is relatively successful in this case, where ChenLB fails to produce a balanced breakpoint.

**Figure 3.14.** Results for Parameter Group D, 30 servers.
Figure 3.15. Results for Parameter Group D, 40 servers.

Rounding out Parameter Group D is the 40 server case. Although AZS warrants the simulation of the LBF choices of 10 and 15 where it did not in Group A, we see that remains a tough choice even for 256 zones. BreakpointLB ends with just over 10% breakpoints balanced, and ChenLB fails to produce a balanced breakpoint in these cases. Increasing LBF helps, and we see increased performance for higher choices of LBF for both BreakpointLB and ChenLB, but only BreakpointLB succeeds in eclipsing 60% balanced breakpoints and that is in the 35 LBF case. Performance is better for both heuristics than in Group A (ChenLB failed to produce a balanced breakpoint in the 40 server case of Group A), but 40 servers remains a tough choice.

Overall in Group D, we observed that the increase of zones from 128 to 256 helped increase the number of balanced breakpoints overall for both BreakpointLB and ChenLB. The 40 server case was still tough to balance even with the finer tuning of balance that the higher number of zones allowed. BreakpointLB remains the clear-cut winner, consistently outperforming ChenLB in terms of balanced breakpoints.
3.5.4.5 Parameter Group E (256 Zones, 50% System Load)

Figure 3.16. Results for Parameter Group E, 10 servers.

In Parameter Group E, we increase average system load to 50%. BreakpointLB continues to balance nearly all breakpoints. While ChenLB balances a greater number of breakpoints as LBF increases, its performance is overall only marginally better than its 10 server Group B counterparts. The benefit of the increased number of zones is mitigated by ChenLB’s susceptibility to increased system load.

Figure 3.17. Results for Parameter Group E, 20 servers.

For the remainder of the server choices in Group E, we see that ChenLB fails to produce an acceptable number of balanced breakpoints in all cases and fails to balance any
breakpoints in many of them. BreakpointLB does well in many cases, but again as the number of servers increases the number of balanced breakpoints decreases. For smaller choices of LBF the number dips below 50% in the 40 server, 20 LBF case, but overall BreakpointLB performs well in these tough cases compared to ChenLB, which cannot produce any balanced breakpoints.

Group E displayed a continuation of the trends we’ve noted in previous Parameter Groups, and added further evidence of BreakpointLB’s ability to create balance where ChenLB cannot.
3.5.4.6 Parameter Group F (256 Zones, 75% System Load)

**Figure 3.20.** Results for Parameter Group F, 10 servers.

**Figure 3.21.** Results for Parameter Group F, 20 servers.

**Figure 3.22.** Results for Parameter Group F, 30 servers.
In Parameter Group F, we again see the same trends in performance noted in the
previous five parameter groups. ChenLB produces a significant number of balanced
breakpoints only in the 10 server case, where BreakpointLB balances nearly all of them. As
the number of servers increases, BreakpointLB produces fewer balanced breakpoints,
however as we noted in Parameter Group C, PZS and RZS allow us to increase the number
of balanced breakpoints over Parameter Group E.

We again provide some example data to display the increasing work performed by
RZS as average system load increases.

**Table 3.2.** Sample data to show work performed by RZS based on average system load.

<table>
<thead>
<tr>
<th>Average System Load</th>
<th>Average # RZS splits</th>
<th>Average Load Gap Reduction Per Split</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>0.76</td>
<td>10.8</td>
</tr>
<tr>
<td>75</td>
<td>12.4</td>
<td>10.34</td>
</tr>
</tbody>
</table>

The table data is sample data from the 20 server, 30 LBF, 256 zone data set. We can
observe that much like its 128 zone counterpart, in the low and medium choices of average
system load the number of splits is few. Again, in the 50% system load case, the load gap
reduction is high, but the number of splits performed on average are few. However, when
the average system load increases to 75%, the average number of splits increases tremendously. We can again attribute BreakpointLB’s increased success in the high average system load case to the work of RZS.

3.5.4.7 Load Gap Reduction by Method Example

![Load Gap Reduction by Method](image)

**Figure 3.24.** Load Gap Reduction by Method (ZAS, RZS, PZS) Method for 20 servers, 128 zone, 25 LBF case.

Figure 3.24 displays an example case of load gap reduction by method for load balancing. Here we see for the 20 server, 128 zone, 25 LBF case the fraction of work for which each piece of our centralized load balancing heuristic is responsible. In the 20 system load case, ZAS is solely responsible for all load gap reduction. Since average system load is low, it is highly unlikely that any server becomes overloaded and requires the operations of RZS, and this confirms that. As we increase average system load, the likelihood that a server overloads increases, and thus RZS is responsible for a larger fraction of load gap reduction. Observe that PZS does not directly reduce load gap. This is because PZS splits zones proactively to increase movement options in future breakpoints –
therefore it is not directly responsible for any load gap reduction. In theory it can directly
reduce load - if PZS proactively splits a zone and that split results in a subzone unsatisfiable
by the original hosting server, then PZS will move that subzone to a different server,
potentially decreasing load gap. However, this rare case did not occur during the course of
our simulations, as this data exemplifies.

3.5.4.8 Confidence Intervals

![Figure 3.25](image)

**Figure 3.25.** 95% confidence interval for 20 server, 128 zone, 25 LBF case.

Figure 3.25 shows example data to display the statistical margin of error for data we
report in centralized load balancing simulations. The graphs display the 95% confidence
interval for the 20 server, 128 zones, 25 LBF case. Each column represents a different
choice of average system load – 25, 50, and 75, respectively. We show the margin of error
for both the total breakpoints balanced as well as the unbalanced breakpoints brought into
balance. (Please note that the scale on the y axis is different for each of these graphs for
viewing purposes.) For both measures, we note that as average system load increases, the
margin of error in our data decreases. For low system load, the number of breakpoints
beginning in balance was generally large, and the heuristic's ability to bring into balance
those that did not begin in balance varied largely. Since beginning breakpoint balance was
already high, it is likely that the breakpoints that were not already in balance were tough cases, and the heuristic’s success in handling these tough cases was inconsistent. This inconsistency lead to a large disparity in the data range. As we increased the average system load, fewer breakpoints began in balance, so there were more breakpoints on which to perform our load balancing operations. This lead to greater number of breakpoints brought into balance on a consistent basis, and thus our margin of error decreased as we increased average system load.

### 3.5.4.9 Overall Observations

We have seen that for several choices of zones, servers, LBF, and average system load that BreakpointLB consistently balances more breakpoints than ChenLB. We also show that in situations where ChenLB is not able to produce a single balanced breakpoint, BreakpointLB is still able to bring some balance to that same system. In many cases, BreakpointLB was able to maintain balance throughout the entire duration of the sample period. Additionally, we saw that in situations of high system load, RZS was able to assist ZAS in creating higher overall breakpoint balance than in lower system load cases. We find merit in our approach and could potentially adapt or augment it to work in other problems, which we will explore briefly in Chapter 5.

Next in Chapter 4, we will again look at load balancing, but this time as a distributed approach, where servers take action with knowledge of local information only.
4. Distributed Load Balancing

4.1 Distributed Load Balancing Introduction and Background

A drawback to a centralized load balancing approach like the one we described previously is that implementation of such an approach requires either global communication or a central server responsible for tracking all of the information required to perform the global balancing operations. If each server must update each other server in the system with all information required for load balancing, the communication traffic among servers could become excessive, especially as the number of servers increases. If one central server runs load balancing, then the communication sent to and from the entire system could overload it, and additionally, it would be a single point of failure for the system were it to fail.

An alternative approach to centralized load balancing is *distributed or local load balancing*. In a local load balancing approach, each server makes decisions relative to local knowledge rather than global knowledge. A server will attempt to balance load with the servers with which it can communicate – its immediate neighbors – and will decide to shed or take on load only with information gathered from those neighbors. Such an approach limits the global communication overhead, since each server need not update every other server in the system or potentially overload a global monitoring server with communication traffic. This type of approach can reduce implementation cost and potentially reduce execution and processing time, as it considers fewer options and passes less information among servers.

We propose a distributed load balancing heuristic that extends our centralized load balancing work. This multi-phased heuristic sees each server send or receive requests for
zones based on its own load relative to its neighbors. Unlike our centralized load balancing work, neighbors are not a dynamic set relative to the zones each server hosts in the game world. Instead, distances in the real world determine neighbors. If the delay between a pair of servers is within a predefined threshold (an implementation detail we will discuss later), then those two servers are neighbors. Once each server has received all of its respective requests during the first phase of the heuristic, in the second phase it will determine which zones it can send to the servers that made requests, and then in the final phase, shed the appropriate zones to neighbors. We name this approach *Pairwise Load Gap Resolution* (PLGR).

As in BreakpointLB, PLGR begins with a set of clients, set of servers, set of zones, value for QoS, and value for pQoSZ, along with the value for LBF. Each server holds this information initially, and the values for QoS, pQoSZ and LBF do not change during the course of gameplay. Each server has a finite capacity for the load incurred by hosting a zone. The primary difference between PLGR and BreakpointLB is that each server knows only the information about itself and the information that it can gather from its immediate neighbor servers. There is no indirect communication between two servers that are not neighbors. Each server makes decisions about load to send or add based on this local information. Since no one server contains the information needed to calculate a global load gap value, we instead define a *pairwise load gap*, which is the largest single pairwise load difference between a server and its neighbors. The neighbor with which this pairwise load gap is shared is defined as the *load gap neighbor*. We also define a server to be in *local balance* if all of its pairwise load gaps with its neighbors are within LBF. We devise operations to take action if a server's pairwise load gap lies outside of LBF and try to
indirectly achieve local balance for each server this way. We perform these operations periodically, as we did in centralized load balancing, for the same motivations defined previously.

4.2 Distributed Load Balancing Method

In PLGR, we initialize each server with knowledge of its server capacity, the value for LBF, the zones that it hosts and the player information within those zones, and the current load those zones incur on it. As with centralized load balancing, when the game begins, players move around the world and we do not make any incremental changes in response to the changing loads. We instead define periodic breakpoints during which the servers assess load and zone satisfaction and then take action if necessary to balance load if it is imbalanced.

We again define balance contingent with LBF, however we do not know if global balance is within LBF since there is no method of obtaining global information in our distributed system. Therefore, we define balance on a per-server basis. A server is in local balance if each of its pairwise load gaps among all of its respective neighbors lies within LBF. If each server at each breakpoint takes measures to reduce its largest pairwise load gap with a neighbor to within LBF, then across the system on average we should be able to reduce global load gap. While we certainly can construct examples where this is not the case, on average we believe that it will be, and we will measure this in our simulations.

PLGR identifies and resolves these load gaps in a three-phase approach.

- Phase 1 is the request phase, in which each server decides if it has a neighbor with which its pairwise load gap lies outside of LBF, and which of those neighbors’ load gap is the largest. The server also decides if the sender in the pairwise transaction
should be either itself or that load gap neighbor based on which of the two servers
has the greater load. If the server is a sender, it adds a request to its own request
queue on behalf of its load gap neighbor to determine the load it can send.
Otherwise, the server is a receiving server, and it sends a message to its load gap
neighbor to add a request on the receiver’s behalf to the neighbor’s request queue to
determine how much load that neighbor can send.

- Phase 2 is zone identification. Each sending server processes requests to determine
which zones, if any, it can send to the receiving server(s) based on pQoSZ
satisfaction and capacity constraints, while not violating LBF.

- Phase 3 is resolution. Sending servers process the queue of zone transfers to the
receiving servers in descending order of load gap, i.e., the neighbors with the largest
load gaps have transfers attempted first. If two receiving servers requested
transfers from the same sending server and both receiving servers satisfy a zone
from the sender, the receiver with the larger load gap to the sender will get that
zone.

Once all transfers are complete, the heuristic is finished until the next breakpoint. The full
pseudocode for PLGR follows.

### 4.2.1 PLGR Pseudocode

<table>
<thead>
<tr>
<th>PLGR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
</tr>
<tr>
<td>Note: Though this heuristic has no global inputs, each server $sl_a$ is assumed to know the following information about itself or the system.</td>
</tr>
<tr>
<td><strong>neighbors</strong> – a list of all servers ${nbr_0, nbr_1, ..., nbr_n}$ that are neighbors to $sl_a$. Contains at most $m-1$ servers where $m$ is the total number of servers in the system.</td>
</tr>
</tbody>
</table>
**Variables**

- **loadgaps** – \( sl_a \) uses this to track the differences in load between itself and its neighbors.
- **role** – a simple value to track whether the server is sender or receiver in load transactions.
- **localGap** – the load gap between \( sl_a \) and its load gap neighbor.
- **checks** – a collection of the server checks \( sl_a \) has received to determine which, if any, zones can be sent to a requestor.
- **requests** – a collection of pending zone transfers resulting from calculations performed in Phase 2. Each element is of the form \( \{ z_s, s_r \} \), where \( z_s \) is the zone hosted by \( sl_a \) to be sent and \( s_r \) is the server that will receive the zone transfer.
- **tmpZone, tmpSrv** – temporary variables used to hold identities of a zone or server, respectively, during Phase 3.

**Outputs**

Each server \( sl_a \) updated with any new zone assignments and load value.

```plaintext
// We describe this distributed algorithm as run by each server \( sl_i \).
// Phase 1 – determine the local load gap for each server and if it is greater than LBF.
// If so, queue up potential zone transfers from the server or make requests to send zones
// to the server, depending on if it is the source or destination based on load for potential
// transfer in Phase 2.

at breakpoint start, broadcast \( sl_i.load \) to each neighbor

// determine pairwise load gaps of the server and its neighbors
for each neighbor \( nbr_x \) of \( sl_i \)

\[ sl_i.loadgaps[ID(nbr_x)] \leftarrow | sl_i.load - nbr_x.load | \]

// gapNeighbor is the server with which \( sl_i \) will be attempting moves
\[ y \leftarrow \text{ID(MaxLoad}(sl_i.loadgaps)) \]

\[ \text{gapNeighbor} \leftarrow sl_i \]

// check whether \( sl_i \) is the low or high end of the load gap
if \((sl_i.load > \text{gapNeighbor}.load)\)

\[ sl_i.role \leftarrow \text{HIGH} \]

else

\[ sl_i.role \leftarrow \text{LOW} \]

\[ sl_i.localGap \leftarrow | sl_i.load - \text{gapNeighbor}.load | \]

// if the loadGap is greater than LBF, take measures to reduce. If not, then
// no action needed

PLGR (cont.)
if (sl.localGap > LBF)
  //if sl is the high server, look for zones to move to gapNeighbor
  //already have access to zones on this server and can compute
  //pQoS satisfaction locally with no additional information
  if (HIGH == sl.role)
    add gapNeighbor to sl.checks

  //if sl is the low server, it is the destination for the zones. Since
  //computations are performed on the sending server, we need
  //to request that the sending server determine which zones it can
  //send
  else if (LOW == sl.role)
    //tell gapNeighbor to check for zones to send to sl
    send message to gapNeighbor to add sl to gapNeighbor.checks

//Phase 2 – determine zones to send
//If a server has a request to determine zones to send, it calculates which zones it
//can send and adds them to its own request queue
if (sl.checks is nonempty)
  for a <- 0 to sizeof (sl.checks) - 1
    destSrv <- sl.checks.a.server
    for all zones zm on sl
      if (checkQoS (zm, destSrv, pQoS, QoS) == 1)
        sl.requests += (zm, destSrv)

//Phase 3 – request resolution
//In this phase, each sending server will assess the requests to send zones from itself to
//various destination servers. It will proceed in descending order of pairwise load gap
//between itself and its neighbors until it has resolved all requests.
if (sl.requests is nonempty)
  sort sl.requests in descending order of pairwise load gap to sl.neighbors

  //though we check servers in descending order of pairwise load gap
  //calculated in phase 1, zone transfers during the course of
  //phase 3 can change sl’s load, therefore we want to make
  //sure before attempting transfers that the load gap is still
  //outside LBF
  //pQoS has already been verified before the request
  //was made, iterate through requests and make as many as we
  //can until within LBF or out of requests for that server
  for a <- 0 to sizeof (sl.requests) - 1
    tmpZone <- sl.requests.a.zone //zone to transfer
    tmpSrv <- sl.requests.a.destSrv //destination server
    //ensure that load gap is not already within LBF and that the

PLGR (cont.)
//requested zone still exists on sl
if( |sl.load – tmpSrv.load| > LBF && sl.hasZone(tmpZone) == 1)
    sl.removeZone(tmpZone)
    sl.load -= tmpZone.cost
    tmpSrv.addZone(tmpZone)
    tmpSrv.load += tmpZone.cost

4.2.2 Time Complexity Analysis

We derive the time complexity for PLGR as follows. We derive this time complexity for a single server since the heuristic runs in a distributed system.

- PLGR: Let $m$ denote the number of active servers in the system, let $p$ denote the number of players in the system, and let $k$ denote the number of zones in the system. Let $q$ denote a server on which PLGR runs during a breakpoint. Phase 1 begins by broadcasting server $q$’s load to each of its neighbors. Next, it updates $q$’s load gap value with each of its neighbors after collecting each neighbor’s load information from the prior broadcast. Finally, it determines whether $q$ is the sender or receiver in the upcoming pairwise load gap operation and adds the corresponding check request to itself or requests its load gap neighbor add a check on its behalf. If $q$ has pending requests for load, Phase 2 determines which zones $q$ can send to those requesting neighbors, checking for QoS satisfaction by calling checkQoS(). Phase 3 sorts those potential zones to send in descending order of pairwise load gap, then processes the send requests until all it has iterated through or sent all zones. The individual time complexities for these phases follows.

  - Phase 1: In the worst case, $q$ is a neighbor of each active server in the system. The load broadcast then takes $O(m)$ time, as does the update of $q$’s pairwise load gaps with its neighbors. Determining whether $q$ is the sender or receiver
and the addition of the check request takes $O(1)$ time. The time complexity for Phase 1 is thus $O(m + m + 1)$, or $O(m)$.

- Phase 2: In the worst case, each active server requests a transfer from $q$, and $q$ hosts each zone in the system. Then for each of $m$ servers, there are $k$ calls to checkQoS(), which we have previously shown runs in $O(p)$ time. The time complexity for Phase 2 in this worst case is thus $O(mpk)$. As explained previously, $p >> k$, and $m \approx k$ as $p$ becomes sufficiently large. We then rewrite this time complexity as $O(m^2 p)$.

- Phase 3: In the worst case, $q$ contains $mk$ requests for zones to send to its neighbors, since if $q$ hosted all $k$ zones, each of its neighbors could make requests for those zones. PLGR sorts these requests in descending order of pairwise load gap, and as shown previously, the zone sort operation is a modified mergesort [40] that runs in $O(n \log(n))$ time in the worst case. This sort thus takes $O(mk \log(mk))$ time. Once the requests have been sorted, PLGR iterates through them and makes assignments as necessary. This takes $O(mk)$ time. The time complexity of Phase 3 is thus $O(mk \log(mk) + mk)$, or $O(mk \log(mk))$.

The time complexity for PLGR is $O(m + m^2 p + mk \log(mk))$. $p >> m$ and $p >> k$, so we can simplify this time complexity to $O(m^2 p)$. Note that although this is the worst-case time complexity, in the average case we expect this to be much less. We would not expect all servers to be neighbors in a distributed system, nor would we expect a single server in the system to host all zones and all players. Therefore, the average runtime should be much less than the worst-case time complexity might indicate.
4.3 Distributed Load Balancing Simulation

In order to simulate heuristic performance, we use the same underlying simulator infrastructure as we used for centralized load balancing (described in detail in Chapter 3), but in accordance with the confines of this problem, we use no global information in order to make load balancing operations.

The parameters we may adjust remain the same:

- Physical world size
- Virtual world size
- Number of active servers
- Server capacity
- Average system load
- IS delay factor
- Number of zones
- Initial player count
- Ramp up cycles
- Sample cycles
- QoS value
- pQoSZ value
- LBF value
- Number of breakpoints
- Number of hotspots
- Hotspot fraction

As in centralized load balancing simulations, the following parameter choices are the same among all simulated parameter groups.

- Initial Player Count – 1% of the average number of players based on average system load
  - Number of active servers * server capacity * system load * 0.01
- Physical World Size – 256x256
- Virtual World Size – 256x256
- Ramp Up Cycles – 500
- Sample Cycles – 20000
- QoS – 150
- IS delay factor – 0.6
- Number of breakpoints - 400
Additionally, to determine which servers are neighbors, we define a radius around each server's physical location, and any other servers that lie within this radius are considered neighbors to that server. This radius is set at \((QoS / 4)\). For example, suppose a server \(S_1\) was located at \([0][0]\) in the physical world representation and a server \(S_2\) was located at \([0][10]\) in the physical world representation, and \(QoS\) was set at 100. \((QoS / 4) = 25\), therefore \(S_1\) and \(S_2\) would be neighbors, as they are located a distance of 10 from one another.

We chose the value of \((QoS / 4)\) experimentally. Originally, the value was set as simply \(QoS\), but this resulted in each server being neighbors with nearly every other server in the system. If all servers are neighbors, then using a distributed approach is not needed, since all servers can communicate with each other directly – it would be best to use a global load balancing approach in this case. Through experiments, \((QoS / 4)\) resulted in servers having an average neighbor count of 30 to 50 percent of the total number of servers, which provides options for load shedding without making the server graph fully connected.

The following parameter choices will be varied among the listed values, with specific combinations eliminated mathematically as described in the centralized load balancing chapter.

- Number of servers – 10, 20, 30, 40
- Number of zones – 128, 256
- LBF – 10, 15, 20, 25, 30, 35

In the next section, we will describe a comparison problem by Deng and Lau [16], who perform load balancing using principles of heat diffusion, and we will adapt their approach into a heuristic we name DengLauLB. We will simulate PLGR as well as DengLauLB with identical parameter choices to measure the fraction of servers that were not in local
balance but then brought into local balance by the work of the heuristic (referred to as Fraction Brought Into Local Balance), the fraction of all servers that ended in local balance (whether by load balancing heuristic operations or by entering the breakpoint already in balance), and the global load gap at the end of the simulation (to get a sense for how these local operations affected global load gap). Additionally, for PLGR only, we will measure the pairwise load gap resolution to see how well our heuristic performs its primary operation – resolving LBF violations with each server’s load gap neighbor.

4.3.1 Deng & Lau’s Dynamic Load Balancing using Heat Diffusion

To the best of our knowledge there are no existing heuristics other than our own to apply to the problem of distributed load balancing to within a load balancing factor. Therefore, in order to compare performance of our heuristic to another approach, we must find an existing heuristic for a problem that is similar to ours and then adapt that approach to operate under the constraints of our problem. We find a good candidate in Deng and Lau’s Dynamic Load Balancing using Heat Diffusion [16]. In this problem, the authors use a well-known heat diffusion method adapted to perform distributed load balancing. The general idea is to balance load in the system by continually sending load from higher loaded servers to lower loaded servers, while determining the portion of load to send using a calculated diffusion coefficient. This coefficient, along with the load difference between the sending and receiving server, dictates the maximum load that may be transferred from a server to a single neighbor so that the sum of transfers to its neighbors does not bring the load of that server below the load of a receiving neighbor.

More formally, the diffusion coefficient $d_c$ between two servers that are performing a load transfer is calculated by:

174
\[
\frac{1}{\max(\deg(S_i), \deg(S_j)) + 1}
\]

where \( S_i \) and \( S_j \) are the servers between which load should be transferred, and \( \deg(S) \) is the degree of the server’s node in the server graph, i.e., its number of neighbors. They then multiply this coefficient by the load difference between the two servers to determine the amount of load to send. The direction of the transfer is always from the higher loaded server to the lower loaded server. If \( S_i \) is the higher loaded of the two, then

\[
\text{Load to send} = dc \times (S_i.\text{load} - S_j.\text{load})
\]

In order to adapt Deng and Lau’s approach to our problem, we must make a few adjustments. First, Deng and Lau’s heuristic runs continually over time, reducing load by smaller and smaller amounts until it achieves balance. This is not permissible under the constraints of our problem because load, i.e., zones and their costs, is not infinitely divisible and load balancing is performed at periodic breakpoints. In order to preserve the foundation of the authors’ approach through this adaptation, we perform their load balancing distribution functions periodically at breakpoints, as our heuristic does. Additionally, to implement balance dictated by LBF, we define a threshold value above which the authors’ adapted heuristic should trigger its operations. The value is calculated similarly to the adapted heuristics of ChenLB described in centralized load balancing.

\[
DL_{thr} = \text{average system load} + \left( \frac{LBF}{2} \right)
\]

At each breakpoint, each server checks its load and if that load lies above the computed \( DL_{thr} \) value, then the server triggers the load balancing operations of this adapted heuristic, which we name DengLauLB. The pseudocode for DengLauLB follows.
4.3.1.1 DengLauLB Pseudocode

DengLauLB

**Inputs**

Note: Though this heuristic has no global inputs, each server $s_l$ is assumed to know the following information about itself or the system.

- **neighbors** – a list of all servers \{nbr_0, nbr_1, ..., nbr_n\} that are neighbors to $s$. Contains at most $m$ servers where $m$ is the total number of servers in the system.
- **load** – the current load incurred on the server by the zones it hosts
- **LBF** – the value for LBF
- **DL_thr** – the computed value for which load balancing operations should be triggered, if the server’s load is greater than it
- **pQoS** – value for pQoS
- **QoS** – value for QoS

**Variables**

- **loadSendAmt** – an array used to hold the amount of load that a server can send to a neighbor, computed as the diffusion coefficient times the load difference between the two servers
- **tmpZonesToSend** – holds a copy of a server’s hosted zones
- **tmpLoad** – temporary variable to track load sent versus max load that can be sent
- **tn, tz** – temporary neighbor/zone variables, respectively

**Outputs**

Each server $s_l$ updated with new load values and zone assignments, if any

**Method**

//We describe this distributed algorithm as run by each server $s_l$
//First phase – identify servers with loads higher than neighbors and greater than LBF. Set //load amount to send as $(1/(\text{max}(\text{deg}(\text{sender}),\text{deg}(\text{receiver}))) + 1) \times \text{load difference}$. //Assume this amount is initialized to 0, so servers with loads lower than their neighbors //or all within LBF dedicate sending no load
at breakpoint start, broadcast $s_l.load$ to each neighbor
if $s_l.load > DL\_thr$
    for each neighbor nbr_x of $s_l$
        if $s_l.load - nbr_x.load > 0$
            loadSendAmt[$s_l.ID$][$nbr_x.ID$] <- ($s_l.load - nbr_x.load)/max(deg($s_l$),deg($nbr_x$)) + 1

//Now that we have determined how much load to send, check for zone satisfaction //create a temporary copy of hosted zones to that we do not dedicate sending //a zone more than once

DengLauLB (cont.)
instrument $\text{tmpZonesToSend} \leftarrow \text{zones hosted by } sl_i$

//go through neighbors to find those with loads to be sent
for each neighbor $nbr_x$ of $sl_i$
  $\text{tmpLoad} \leftarrow \text{loadSendAmt}[sl_i.ID][nbr_x.ID]$
  if $\text{tmpLoad} > 0$
    //go through remaining zones that $s$ can send
    for all zones $z$ in $\text{tmpZonesToSend}$
      //check pQoSZ satisfaction and that the load change from moving
      //the zone does not exceed the load to be sent
      if checkQoS($z$, $nbr_x$, $pQoSZ$, $QoS$) == 1 and $\text{tmpLoad} - z.Cost \geq 0$
        //create an element consisting of zone to send and to
        //whom to send
        $sl_i.zonesToSend.add(z, nbr_x)\$
        $\text{tmpLoad} -= z.cost$ //adjust load left to be sent
        $\text{tmpZoneToSend}.remove(z)$ //prevent zone from
        //being sent elsewhere
  else
    break

//Finally, perform all transfers
for all elements $e$ in $sl_i.zonesToSend$
  $tn \leftarrow e.n$
  $tz \leftarrow e.z$
  $tn.add(tz)$ //implicit load adjustment
  $sl_i.remove(tz)$ //implicit load adjustment

Now that we have described our approach, Deng and Lau’s adapted approach, and our
simulator, we next review the results of our simulations.
4.3.2 Distributed Load Balancing Simulation Results

4.3.2.1 Parameter Group A – 128 Zones, 25% System Load

![Graphs showing simulation results for 10 servers in Parameter Group A.](image)

**Figure 4.1.** Simulation results for 10 servers in Parameter Group A.

We begin with the 10 server case of Parameter Group A. We note that each of pairwise load gap resolution, fraction brought into local balance, overall load balance fraction, and average load gap at simulation end increase as LBF increases. The pairwise load gap reduction fraction is 0 for the 30 and 35 LBF cases, but the overall local balance fraction data reveals the cause – each breakpoint remained in balance throughout the course of the simulation, and no pairwise operations were required. (Note that DengLauLB
also achieved this in the 35 LBF case.) In the cases where balancing operations at breakpoints were needed, PLGR both brought more servers into local balance and achieved a higher fraction of locally balanced servers overall than DengLauLB. Both PLGR and DengLauLB have an average load gap below LBF at the end of the simulation for all choices of LBF, and PLGR's produced load gap was always smaller than DengLauLB's.

![Graphs showing simulation results for 20 servers in Parameter Group A.](image)

**Figure 4.2.** Simulation results for 20 servers in Parameter Group A.

Continuing to the 20 server case of Parameter Group A, we see that overall the load gap resolution, fraction brought into local balance, and overall local balance fraction generally decrease compared to the 10 server case, while average load gap at simulation
end increases. Although both PLGR and DengLauLB suffer decreases in performance across the board, PLGR attains a higher overall load balance fraction than DengLauLB in all cases, and is at or above a 0.8 fraction in each of the 25, 30, and 35 LBF cases. PLGR also ends with load gap within LBF at simulation end in each of the 25, 30, and 35 LBF cases, while DengLauLB does not end within LBF in any case. DengLauLB does, however, bring a larger fraction of servers into local balance than PLGR in the 25 and 35 LBF cases, however as previously mentioned, PLGR still attains a higher overall local balance fraction in those cases.

**Figure 4.3.** Simulation results for 30 servers in Parameter Group A.
Increasing the number of servers to 30 in Parameter Group A again has a noticeable effect on the performance of both heuristics. We see a very small fraction of servers able to be brought into local balance. PLGR does bring a larger fraction into local balance than DengLauLB in all cases but 35 LBF, and still has a respectable overall local balance fraction in the 30 and 35 LBF cases. Average load gap at simulation end is high, yet PLGR is still able to come within LBF in the 30 and 35 LBF cases. As with centralized load balancing, this is a challenging parameter set to satisfy, and PLGR performs better than DengLauLB in nearly all facets.

**Figure 4.4.** Simulation results for 40 servers in Parameter Group A.
Rounding out Parameter Group A is the 40 server case. As we might expect from the simulation results in centralized load balancing, 40 servers is a challenging parameter choice for all parameter groups and that holds true for distributed load balancing as well. Very few servers are able to be brought into local balance, and the overall local balance fraction is under 0.5 in all cases apart from PLGR’s output in the 35 LBF case. No heuristic ends with average load gap within LBF in any case, though PLGR ends with a lower gap than DengLauLB, and its overall local balance fraction is higher than DengLauLB in all cases.

Overall in Parameter Group A we see many of the same trends we noted in centralized load balancing, such as a general increase in performance as LBF increases within each parameter group and a general decrease in performance as number of servers increases. Although there are a few cases in which DengLauLB performed better than PLGR, overall PLGR outperformed DengLauLB in each metric. Additionally, while pairwise load gap resolution is very good in the 10 server case, as with the other metrics, it decreases sharply as the number of servers increases. Observe that as this number of servers increases, the average number of zones per server decreases. This decreases the number of choices a server has for shedding its own load, and the restrictions of pQoSZ further limit which zones it can send to its pairwise load gap neighbor. This makes resolution of that load gap within LBF more difficult for a larger number of servers.

4.3.2.2 Parameter Group B – 128 Zones, 50% System Load

Parameter Group B increases the average system load to 50%. Although performance reduces across the board when compared to the Parameter Group A
Figure 4.5. Simulation results for 10 servers in Parameter Group B.

counterpart, PLGR still performs well in the 10 server case as before, particularly in the 20/25/30/35 LBF cases. Additionally, PLGR performs better than DengLauLB across the board, apart from its average load gap at simulation end in the 15 LBF case. PLGR’s load gap is within LBF in the 20/25/30/35 cases, as is DengLauLB’s in three of those four cases.

The 20 server case of Parameter Group B displays a sharp decline in heuristic performance from the 10 server case, even more so than the transition between the Parameter Group A counterparts. In particular, the fraction of servers brought into local
balance is much lower. However, the overall local balance fraction is still good for PLGR. Additionally, it is able to keep the global load gap within LBF in the 35 LBF case and nearly does so in the 30 LBF case. DengLauLB performs very poorly here, and its load gap at end is double (or more) that of PLGR in all cases. The increased average zone size resulting from the increase in average system load can cause zones to be more difficult to satisfy, and additionally, with zones being larger, the fine tuning of load on which DengLauLB depends can become difficult to achieve. This will be even more apparent as we further increase numbers of servers and average system load.
Figure 4.7. Simulation results for 30 servers in Parameter Group B.

We increase the number of servers to 30 in Parameter Group B and again observe some of the same trends we have seen previously in Parameter Group A as well as centralized load balancing, PLGR’s performance dips further. DengLauLB struggles in all metrics, and its average load gap at end is very poor, ending with presumably several overloaded servers in the 30 and 35 LBF cases, as the average load gap is greater than 100%. (Note the change in scale on the y axis of the graph.) Although PLGR also fails to end with load gap within LBF, its load gap is less than half that of DengLauLB in all cases. In this challenging parameter set, PLGR clearly outperforms DengLauLB.
Figure 4.8. Simulation results for 40 servers in Parameter Group B.

The 40 server case shows no redeeming performance from either heuristic and continues to be difficult. DengLauLB brings more servers into local balance that were originally unbalanced, but PLGR ends with more servers in local balance overall. Neither gets near to LBF in load gap at the end, and DengLauLB again ends in overload, with a gap of nearly twice the size of PLGR. (Again, note the increased scale of the y axis.) For servers of this number or greater, a more tailored heuristic may be necessitated.

Overall in Parameter Group B we noted the continuation of previously observed trends in performance subject to value of LBF and numbers of servers, as well as the
average system load trends seen in centralized load balancing. The performance difference between PLGR and DengLauLB was more apparent in Group B, particularly in the average load gap at the end of simulation metric, where DengLauLB ended in overload for two choices of number of servers. PLGR, while also unable to end with LBF in these cases, was still about half of DengLauLB’s load gap in those same cases.

4.3.2.3 Parameter Group C – 128 Zones, 75% System Load

![Graphs showing comparison between PLGR and DengLauLB in parameters Group C.](image)

**Figure 4.9.** Simulation results for 10 servers in Parameter Group C.

In the last of the 128 zone cases, Parameter Group C sees the average system load increase to 75%. The 10 server case results are close to their Group B counterparts, and PLGR performs better than DengLauLB in nearly every metric, save for the 15 LBF case of
average load gap at simulation end. PLGR is able to keep load gap within LBF at simulation end for four of the five choices of LBF, while DengLauLB does not for any LBF choice.

The 20 and 30 server choices in Parameter Group C further cement the trends and observations seen in their Parameter Group A and Parameter Group B counterparts. For 20 servers, PLGR is able to keep load gap at or near LBF, while DengLauLB’s produced load gap is twice or more that of PLGR. PLGR’s overall local balance fraction is respectable in this group. However for 30 servers, things are poor across the board for both heuristics, though again PLGR outperforms DengLauLB.

**Figure 4.10. a)** Simulation results for 20 and 30 servers in Parameter Group C.
b) Simulation results for 20 and 30 servers in Parameter Group C.

Parameter Group C solidified the observations seen in other groups and centralized load balancing. Among all parameter groups, it is apparent that for 128 zones, PLGR is better able to cope with the parameter choices, and although it does not perform great under strenuous parameter choices, it performs better than DengLauLB.

We next increase the number of zones to 256, which we previously observed to generally increase heuristic performance in centralized load balancing compared to its 128 zone counterparts. We will investigate whether that trend holds true in distributed load balancing also and see how each heuristic responds.
4.3.2.4 Parameter Group D – 256 Zones, 25% System Load

The 10 server case shows the heuristic performance improvement in both heuristics that we expect based on the centralized load balancing results. As with the 10 server case of Parameter Group A, we see in the 30 and 35 LBF cases, and additionally, the 25 LBF case, that all breakpoints are balanced for the duration of the simulation. (This explains the lack of data for pairwise load gap resolution for those LBF choices.) Both heuristics keep load gap within LBF at simulation end in all LBF choices. Overall, performance is very good for both heuristics for all parameter choices in this case.

Figure 4.11. Simulation results for 10 servers in Parameter Group D.
When we increase servers to 20 in Parameter Group D, we see a performance decrease that is consistent with previous trends, though the results are still good. Both PLGR and DengLauLB perform well for the most part in terms of overall local balance fraction. For LBF choices of 20 and higher, PLGR maintains load gap within LBF at simulation end, while DengLauLB does so for choices of 25 and higher. Other statistic trends are in accordance with their prior group counterparts.

With 30 servers in Parameter Group D, we note the first parameter set in which DengLauLB has a string of successes – for each of 25, 30, and 35 LBF, it brings a larger
Figure 4.13. Simulation results for 30 servers in Parameter Group D.

to PLGR. However, when compared with the overall local balance fraction produced by each heuristic, PLGR ends with a greater fraction of servers in local balance, doing a better job of maintaining local balance over time across breakpoints. PLGR is the only heuristic to keep average load gap at end within LBF, specifically in the 25 and 35 LBF cases.

Lastly, the 40 server case shows reduced performance as in the previous parameter groups. PLGR is able to maintain overall local balance in cases of 25 LBF or greater, and average load gap at simulation end lies within LBF in the 35 LBF case, but as we have seen
Figure 4.14. Simulation results for 40 servers in Parameter Group D. Previously, this is simply a very demanding parameter choice and often smaller LBF choices are difficult to satisfy.

Overall in Parameter Group D, PLGR and DengLauLB saw increased performance metrics as a result of the increase in number of zones, which is in accordance with our expectations from centralized load balancing results. Though there were a few parameter choices under which DengLauLB saw better performance metrics than PLGR, overall PLGR clearly outperformed DengLauLB in most or all performance metrics. The 40 server case
remains a difficult case as it has for each parameter group in both distributed and centralized load balancing.

4.3.2.5 Parameter Group E – 256 Zones, 50% System Load

![Graphs showing performance metrics for Parameter Group E](image)

**Figure 4.15.** Simulation results for 10 servers in Parameter Group E.

In Parameter Group E in the case of 10 servers, performance is good for both heuristics. Although in the 15, 20, and 30 LBF cases DengLauLB brings a greater fraction of servers into local balance, PLGR maintains a larger overall local balance fraction in all but the 15 LBF case. As we have seen in previous parameter groups, these two statistics do not necessarily imply anything about the other – it would appear that PLGR is better at maintaining balance over time, so it has fewer servers that are not in local balance at the
start of breakpoints. The set on which it operates is a smaller and more difficult set of servers on which to perform these balancing operations, and so although it brings a smaller fraction of servers into balance than DengLauLB, overall its local balance fraction is better consistently. Here, its average load gap at end is very close to DengLauLB’s, and smaller in all but the 10 LBF choice. Each heuristic keeps the load gap below LBF in all but the 10 LBF choice.

![Figure 4.16](image.png)

**Figure 4.16.** Simulation results for 20 servers in Parameter Group E.

When we increase servers to 20 in Parameter Group E, a large disparity between the results of PLGR and DengLauLB appears. The overall local balance fraction reduces drastically for DengLauLB, and its fraction brought into local balance drops significantly.
While PLGR also suffers a performance dip consistent with previous increases of number of servers, it is not as severe as the drop shown by DengLauLB. Also, where each heuristic was able to keep average load gap mostly within LBF in the 10 server case, DengLauLB fails to do so in any LBF choice here. PLGR does so for values of LBF 20 and above. We saw this same performance drop from the 10 server case to the 20 server case of Parameter Group B, though with more zones, the overall performance here is still better, as we’d expect from prior observations.

**Figure 4.17.** Simulation results for 30 servers in Parameter Group E.

The 30 and 40 server choices in Parameter Group E see the continued decrease in performance of these metrics. PLGR is able to bring average load gap to within LBF in the
**Figure 4.18.** Simulation results for 40 servers in Parameter Group E.

30 server, 30/35 LBF cases, but DengLauLB’s average load gap grows well beyond LBF in both the 30 and 40 server cases for all choices of LBF. Other performance is poor across the board, as has been the case prior, but PLGR is able to outperform DengLauLB in each of these tough cases.

Parameter Group E saw the same performance decline as Parameter Group B, although performance was better across the board for both heuristics than each set’s Group B counterpart. DengLauLB saw a few successes against PLGR, but overall PLGR was able to achieve greater success where DengLauLB could not. Additionally, PLGR was far closer to
getting average load gap to within LBF. DengLauLB simply cannot get anywhere close as numbers of servers increase, especially with higher average system load.

**4.3.2.6 Parameter Group F – 256 Zones, 75% System Load**

**Figure 4.19.** Simulation results for 10 servers in Parameter Group F.

In the final parameter group, average system load increases to 75%. Overall performance is good for both heuristics in the 10 server case as with all parameter groups. There are no new trends to report here.

The remaining choices of servers display trends consistent with previous parameter groups. Note the change of scale in the average load gap graph – DengLauLB's average load
Figure 4.20. Simulation results for 20 servers in Parameter Group F.

Figure 4.21. a) Simulation results for 30 servers in Parameter Group F.
b) Simulation results for 30 servers in Parameter Group F.

Figure 4.22. Simulation results for 40 servers in Parameter Group F.
gap again approaches or goes into overload as seen previously. Increasing the number of servers shows the same decrease in performance noted in prior groups, and PLGR still performs better overall.

Across all parameter groups we found that heuristic performance followed trends similar to or the same as the trends we observed in centralized load balancing. As number of servers increased, the restrictions imparted by pQoSZ were amplified, particularly in the 128 zone case. Increasing zones to 256 lessened this strain, since the average zone size decreased as a result, and thus resolving load gaps became easier. However, when servers increased to 30 or beyond, especially for mid to high system loads, these operations still failed to consistently resolve load gap disparities for smaller values of LBF. These cases require a larger LBF value is to go along with the larger, tougher-to-satisfy zones naturally produced from the other parameter choices, allowing for a large enough buffer to move them around. Tighter LBFs simply cannot accommodate these zones, and that is apparent from the results we have shown.

Although the diffusion method utilized by DengLauLB is a tried-and-true method of load balancing, its dependence on infinitely or highly divisible load causes it to perform worse as the loads it must manipulate become larger in size and with fewer servers on which to place them satisfactorily. Although neither of these heuristics deals with directly bringing a server into local balance, PLGR does better overall. Additionally, PLGR’s performance is almost always better in the load gap metric.
4.3.2.7 Confidence Intervals

Figure 4.23. 95% confidence interval for 20 server, 128 zone, 25 LBF case.

Figure 4.23 shows example data to display the statistical margin of error for the data we report in our distributed load balancing simulations. The graphs display the 95% confidence interval for the 20 server, 128 zone, 25 LBF case. We show the margin of error for each of our measures for distributed load balancing – pairwise load gap reduction, fraction of breakpoints brought into balance, overall load balance fraction, and average load gap at simulation end. (Please note that the scale on the y axis is different for each of these graphs for viewing purposes.) As opposed to the margin of error graphs shown for our centralized load balancing simulations, the data for distributed load balancing shows no strong trends in margin of error. In the graphs other than average load gap at simulation
end, the margin of error starts to decrease as average system load decreases, however in each of those graphs the margin of error instead increases again once average system load is 75. The average load gap at simulation end graph displays no trend in margin of error. This metric can vary greatly due in part to the nature of our snapshots of data collection as well as the nature of periodic load balancing, where load can change drastically between breakpoints. The combination of these factors makes it possible that the load gap could increase significantly just before the final breakpoint before the end of the snapshot, giving the heuristic just a single breakpoint to attempt to bring load back into balance. Although we did observe some trends in the average load gap results themselves relative to the input parameters, the margin of error for those results does not appear to show any similar trends.
5. Future Work

Our findings in our heuristics for Offline CAP-Z as well as centralized and distributed load balancing merit further investigation into other related problems or alternative approaches. In load balancing, we can further augment our centralized load balancing solution for application in a production environment where average system load is unfixed. We would remain committed to load balance within LBF, but introduce a maximum or minimum value for current average system load that warrants activating new servers or deactivating current resources to which to spread load. This allows average system load to grow or shrink and keeps the system in balance all the while, but also makes accommodations for the system taking on more load than it can handle or less load than it should with the current number of servers. For example, define max average system load to be 80% and min average system load to be 20%. As long as the average server load is between the max and min system load values, then we need no additional action apart from normal load balancing operations as in the simple static value method. However, if average system load were to exceed 80%, we would activate a new server to decrease average system load, and we would also use it as a new target for load balancing operations. Likewise, if the average system load were to fall below 20%, we would deactivate an active server (preferably lightly loaded or with zero load) to increase average system load and server utilization. Nae et al. [36] considered a problem similar to this and would be a good candidate for heuristic performance comparison.

In distributed load balancing, the performance of our heuristics when number of servers is high warrants further research into augmenting our heuristic for these scenarios. It is possible that more exhaustive or nuanced operations could find balance in these
difficult parameter sets. Although we attempt to refrain from computationally expensive solutions, additional computation that results in a successful solution is preferable to a solution resulting in an excessive number of unsatisfied players, so additional research in this area is warranted.

We would like to adapt our learnings and methods for Offline CAP-Z into other variants of the CAP. Although we outlined a few of the variants, we anticipate that as new infrastructures and methods of gaming online develop, so will CAP variants. Online CAP-Z and both Offline and Online CAP-C are existing candidates for which we could adapt the work here to approximate solutions. We also would be interested in augmenting our simulator with “inaccurate” input data as Ta et al. [44] do, which could improve its applicability for adaptation into a production online game.

Finally, the proposal by Morillo et al. [34] that server response time increases non-linearly as players join, i.e., one overloaded server or several highly loaded servers can cause performance degradation, is an interesting one. We propose to augment our approaches in each of the problems we investigated to accommodate for this and see how they respond. Additionally, we would like to implement each of our heuristics in a real online game, or a production server environment, to see how they respond and perform.
6. Conclusion

In the particular problem of Offline CAP-Z, we have shown that, in practice, solutions should consider server capacities and that the runtime of solutions to an uncapacitated version of the problem does not scale well in the capacitated version. We have developed three heuristics to improve upon the results of Ta et al.’s heuristics. While there is not one clear “best” heuristic, each performs well under specific criteria, and each could suit an application based on the pre-specified known criteria for best performance.

The other problem of load balancing, for which we investigated centralized and distributed solutions, is another online gaming problem that still merits consideration as the industry continues to grow. Providing a satisfactory online experience for dozens or hundreds or thousands of players located around the world as they interact and move throughout the game will continue to be an issue as long as the computational resources to process those actions are finite, and so methods to balance the load incurred must improve as the architecture improves. We have shown methods for both centralized and distributed load balancing architectures that outperform existing methods in the literature and warrant implementation in existing or future games, and further refinement and adaptation to improve their performance under difficult constraints.

Online gaming is a fascinating arena for which we have investigated just a few of the many problems that merit attention. We look forward to continued research in this area to develop improved heuristics that deliver quality player experiences for years to come as the media and underlying infrastructures for these games adapt and improve.
Works Cited


[40] Java Documentation – Collections.sort runtime. https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#sort(java.util.List,%20java.util.Comparator)


Vita

Shawn Michael Farlow was born in Metairie, Louisiana. His interest in computing began in the early 1990s after being given his first computer and continued to grow from there, leading to attending Louisiana State University and entering the Division of Electrical Engineering and Computer Science, obtaining his bachelor's degree in computer engineering. Following a hiatus from university, he returned to obtain his master's degree, and will graduate with his doctorate in 2018.