

LOCALITY AWARE DYNAMIC LOAD MANAGEMENT FOR MASSIVELY
MULTIPLAYER GAMES

by

Jin Chen

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2005 by Jin Chen

Abstract

Locality Aware Dynamic Load Management for Massively Multiplayer Games

Jin Chen

Master of Science

Graduate Department of Computer Science

University of Toronto

2005

Most massively multiplayer game servers employ static partitioning of their game world into distinct mini-worlds which limits cross-server interactions between players. We have designed and implemented an architecture in which the partitioning into regions is transparent to players and interactions are not limited to objects and players in a single region or server. This allows a finer grain partitioning, which combined with a dynamic load management algorithm allow us to better handle transient crowding. Our load balancing algorithm is aware of the spatial locality in the virtual game world. It balances the load and reduces the cross-server communication, while avoiding frequent reassignment of regions. Our results show that locality aware load balancing reduces the average user response time by up to a factor of 4 compared to global algorithms that do not consider spatial locality and by up to a factor of 6 compared to static partitioning.

Acknowledgements

The first person I would like to thank is my supervisor Professor Cristiana Amza. She patiently guides me to explore this research problem in depth. Without her insightful advice and encouragement, I could not finish this thesis.

I am deeply grateful to all research members of *System Support for Massively Multiplayer Online Games* project at University of Pennevenia. They are Professor Honghui Lu, Bjorn Knutsson, Baohua Wu and Margaret Delap. They gave me many precious comments, and developed SimMud with player switch and quests functions.

I would like to express my gratitude to the second reader of my thesis, Professor Hans-Arno Jacobsen, for his precious comments.

I sincerely thank my husband, my parents and my sister for their consistent support and endless love.

Last but no the least, I thank my officemates, and other friends for their priceless friendship.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 2 | Background | 5 |
| 2.1 | Load Balance Background | 5 |
| 2.2 | Game Server Background | 6 |
| 2.2.1 | Game Objects | 7 |
| 2.2.2 | Server Architecture | 8 |
| 3 | Locality-Aware Dynamic Load Management | 10 |
| 3.1 | Overview | 10 |
| 3.2 | Load Thresholds | 11 |
| 3.3 | Load Shedding Algorithm | 12 |
| 3.3.1 | Load Shedding to Neighbors | 12 |
| 3.3.2 | Load Shedding to Lightly Loaded Candidates | 13 |
| 3.3.3 | Heuristic for Partitioning the Regions During Load Shedding | 13 |
| 3.3.4 | Optimization for Maintaining Strong Locality | 16 |
| 3.4 | Other Static and Dynamic Algorithms | 17 |
| 3.4.1 | Dynamic Uniform Load Spread (Spread) | 17 |
| 3.4.2 | Dynamic Load Shedding to Some Lightest Loaded Node Known (Lightest) | 18 |
| 3.4.3 | Static Partitioning | 19 |

| | | |
|----------|---|-----------|
| 3.5 | Discussion | 19 |
| 4 | Implementation | 23 |
| 4.1 | Game objects in SimMud | 23 |
| 4.2 | Multiple Regions in SimMud | 25 |
| 4.3 | Player Migration between Regions | 27 |
| 4.4 | Migration of Regions | 28 |
| 4.5 | Discussion | 33 |
| 5 | Experimental Results | 34 |
| 5.1 | Single Server Experiments | 34 |
| 5.2 | Multiple Servers Experiments | 39 |
| 5.3 | Region Migration | 40 |
| 6 | Simulation Results | 43 |
| 6.1 | Simulation Methodology | 43 |
| 6.1.1 | Resource Usage and AOI Simulation | 43 |
| 6.1.2 | Player Mobility Simulation | 45 |
| 6.2 | Simulation Parameters | 47 |
| 6.3 | Results for Static Partitioning | 48 |
| 6.4 | Results for Dynamic Load Balancing Algorithms | 51 |
| 7 | Related Work | 57 |
| 7.1 | General Load Balancing Algorithms | 57 |
| 7.2 | Load Balancing in Parallel Applications | 58 |
| 7.3 | Load Balancing in Games | 60 |
| 8 | Conclusion and Future Work | 61 |
| 8.1 | Conclusions | 61 |
| 8.2 | Future Work | 61 |

List of Tables

| | | |
|-----|--|----|
| 5.1 | Size of server state updates. | 36 |
| 5.2 | Cost of region migration. | 40 |
| 6.1 | Comparison of total number of player hand-offs, total number of region migrations and the total number of region clusters at the end of simulation for all algorithms. | 55 |

List of Figures

| | | |
|-----|---|----|
| 2.1 | Partitions of the game world. | 8 |
| 3.1 | Pseudo code for Locality-Aware Load Shedding. | 22 |
| 4.1 | Screen shot of SimMud. The dots represent players. Blocks represent road- blocks. Food is represented by coffee mugs. | 24 |
| 4.2 | Multiple server SimMud architecture. | 26 |
| 4.3 | Player migration between regions. | 28 |
| 4.4 | Region migration. | 30 |
| 4.5 | Region migration state diagram (client) | 31 |
| 4.6 | Region migration state diagram (old server) | 32 |
| 4.7 | Region migration state diagram(new server) | 32 |
| 5.1 | CPU utilization. As can be seen, CPU is not a bottleneck. The drop at 256 is due to clients terminating because of network overload. | 35 |
| 5.2 | Server bandwidth utilization. When requirements exceed available bandwidth, delays skyrocket and losses cause clients to terminate. | 35 |
| 5.3 | Periodical update interval (150 clients) | 37 |
| 5.4 | Periodical update interval (175 clients) | 38 |
| 5.5 | Periodical update interval (200 clients) | 38 |
| 5.6 | The number of players v.s. periodical update interval | 39 |

| | | |
|------|---|----|
| 5.7 | Robot players are moving toward a target (quest), in the northwest region (Server 1), many of them crossing region/server boundaries in the process. . . . | 40 |
| 5.8 | 128 (robot) players distributed over four servers. At time $t \approx 65$ a quest in Server 1's region is created, and players flock there for 300 seconds. | 41 |
| 5.9 | As more players join Server 1's region, its CPU consumption goes up, while the others' decline. | 41 |
| 5.10 | We come very close to the maximum sustainable bandwidth. Average delays (not shown in figure) from Server 1 jump from $5ms$ at $t = 50$ to $180ms$ at $t = 700$ | 42 |
| 6.1 | Player distribution snapshot without flocking | 46 |
| 6.2 | Player distribution snapshot during flocking | 46 |
| 6.3 | Comparison of Block Static Partitioning and Cyclic Static Partitioning for centralized LAN-based server simulated according to the SimMud measurements taken in our environment. Center hotspot. Simulation parameters are 100 servers, 400 regions, 6000 players and 1000 seconds hotspot duration. The network to clients is the bottleneck during the hotspot. | 48 |
| 6.4 | Comparison of Cyclic Static Partitioning and Locality-Aware Dynamic Partitioning for centralized LAN-based server simulated according to the SimMud measurements taken in our environment. Corner hotspot. Simulation parameters are 100 servers, 400 regions, 6000 players and 2000 seconds hotspot duration. The network to clients is the bottleneck during the hotspot. | 49 |
| 6.5 | Comparison of Block Static Partitioning and Cyclic Static Partitioning for centralized WAN-based server simulated according to the SimMud measurements taken in our environment. Center hotspot. Simulation parameters are 100 servers, 400 regions, 6000 players and 1000 seconds hotspot duration. The network to clients is the bottleneck during the hotspot. | 50 |

| | | |
|-----|--|----|
| 6.6 | Comparison of Dynamic Load Partitioning for centralized LAN-based server simulated according to the SimMud measurements taken in our environment. Center hotspot. Simulation parameters are 100 servers, 400 regions, 6000 players and 1000 seconds hotspot duration. The network to clients is the bottleneck during the hotspot. | 53 |
| 6.7 | Comparison of Dynamic Load Partitioning for WAN server simulated according to the SimMud measurements taken in our environment. Center hotspot. Simulation parameters are 100 servers, 400 regions, 6000 players and 1000 seconds hotspot duration. The network to clients is the bottleneck during the hotspot. | 54 |

Chapter 1

Introduction

The popularity of massively multiplayer online games (MMOGs) like Lineage [20] and Everquest [24] is on the rise with millions of registered players, and hundreds of thousands of concurrent players. Compared with other online game such as first person shooter game Quake [12], MMOGs are much larger scale and usually hosted by public service providers. Most of MMOGs fall in the RPG (Role Playing Game) category where players are involved in some science fiction or post apocalypse world or fantasy world stories similar to the real world, and could be alive in a game for a long time, typically several months.

Current state-of-the art servers such as Butterfly.net [3] and TerraZona [26] achieve scalability by splitting the game world into linked mini-worlds that can be hosted on separate servers. These mini-worlds are effectively like separate games, with the added ability of moving players between them through special gateways. Admission control at the gateways ensures that no server gets overwhelmed by players. However, newer games such as Sims Online [6] and Tabula Rasa [18] feature more dynamic landscapes making static load prediction harder, hence static partitioning is infeasible.

Flocking is the movement of many players to one area or hotspot in the game world. It may occur because of features of the game design or because of agreements among players. If the game world has some regions that are more interesting, or more profitable to players in terms

of points, experience, treasure, etc., then those regions will attract more players while other regions remain underloaded. Such game hot-spots appear spontaneously and move over time as a result of players chasing each other, completing parts of a quest or following invitations from other players to join them. The exact location of the next hot spot is only partially predictable since it could be determined both by in-game and out-of-game events, such as email exchanges among players. Therefore, no static game partitioning algorithm can effectively address this problem.

To support MMOGs with more dynamic features, we have designed and implemented an architecture in which the partitioning into regions is transparent to players, visibility and interactions are not limited to objects and players in a single region or server, and regions are mapped dynamically to servers. This means that the transparency also extends to designers, reducing the need to adapt the map of the world in order to address load balancing concerns.

To overcome the shortcomings of static partitioning, we design dynamic load management algorithms through adaptively mapping regions to servers, i.e., region migration. It seems an obvious solution to the load management problem, however, any effective dynamic region migration algorithm should evaluate the potential trade-offs between balancing load, decreasing inter-server communication and minimizing migration cost. Hence, we define the game server load management problem as follows:

DEFINITION [The Game Server Load Management Problem] We are given a game map consisting of K regions, and a current assignment of K regions to N servers in which each region is held by one server. The goal of our dynamic load balancing algorithm is to relocate these regions to servers such that to *balance the server load in terms of numbers of players, decrease inter-server communication, and keep the migration cost low.*

To balance the server load in terms of numbers of players, we could split existing partitions across *more* servers. To decrease inter-server communication, we could maintain the locality of adjacent regions or aggregate adjacent regions into large partitions to be assigned to *fewer* servers. Since dynamic migrations of regions is complex and may lead to high overheads, we

should keep the migration cost low.

We target running the game on large wide-area distributed platforms as well as centralized networks as in state-of-the-art game servers. Currently, some load balancing techniques for game or distributed virtual environment systems [16] approximately solve this problem by global optimization algorithms involving all nodes in the system. These load management algorithms are very compute and communication intensive since they require global knowledge. In contrast, we use a localized algorithm. This algorithm is triggered by the individual overload server node and it involves only a small portion of the network (usually corresponding to the in-game neighbor servers or a remote server with light load).

We use a prototype implementation of the SimMud game server [14] and a set of clients running on a 1GHz dual processor Pentium III cluster interconnected by a 100 Mbps LAN to study limitations of a single server. We plot the server's response time curve with increasing number of clients. We measure CPU and bandwidth at the server. We use the measured delays to calibrate a configurable simulator in terms of CPU and network contention curves. We then evaluate various versions of dynamic load management algorithms with and without spatial locality versus static load balancing partitioning, using simulations that involve player flocking. We explore the behavior of locality-aware dynamic load management through simulation in two configurations: a LAN game server architecture and a WAN game server architecture.

Contributions

In summary, the contributions of this thesis are as follows:

1. We study the performance of various state-of-the-art static partitioning algorithms in game servers under scenarios with player flocking.
2. We introduce dynamic load balancing algorithms and we explore the tradeoffs between locality preserving and load balancing in game servers with dynamic load management.

We use a real experimental testbed to study the performance degradation of player flocking, and measure the cost of region migrations. We use a simulator we have developed and cali-

brated to thoroughly compare various static and dynamic load management algorithms under different environments (a LAN and a WAN environment). Our results show:

1. Static partitioning algorithms have inconsistent behavior across environments and hotspot locations, and perform poorly in some experiments. Hence, static partitioning algorithms are not suitable for dynamic gaming involving significant player flocking.
2. Dynamic load balancing algorithms generally outperform static partitioning algorithms in all configurations.
3. Differences between dynamic load balancing algorithms with and without spatial locality are minimal in the LAN environment. On the other hand, in the WAN environment, preserving spatial locality improves performance by up to a factor of 4 compared to global algorithms that do not consider spatial locality and by up to a factor of 6 compared to static partitioning.
4. The migration cost of dynamic load management algorithm with locality preserving is low comparatively with other dynamic algorithms.

The rest of this thesis is organized as follows: Chapter 2 introduces background on load balancing problems and game servers. Chapter 3 introduces our dynamic load management solution. We describe our prototype implementation and experimental methodology in Chapter 4 and experimental results in Chapter 5. We investigate the performance of the different static and dynamic load management algorithms for large numbers of concurrent players through simulation in Chapter 6. Chapter 7 discusses related work. Chapter 8 concludes the thesis and points out the future work.

Chapter 2

Background

2.1 Load Balance Background

For many parallel or distributed applications, uneven workload distributions easily make some machine or processor overloaded, and thus lead to performance degradation. Hence, load balancing is critical for achieving good performance for these applications. The load balancing problem, also referred to as the multiprocess scheduling or parallel machine scheduling problem, or generalized assignment problem (GAP) [22], can be generally defined as follows:

DEFINITION [The Load Balance Problem] We are given K jobs of varying size and N empty processors or machines, and each job is required to be run on exactly one processor or machine. The goal is to assign these tasks to processors such that its optimization criteria are satisfied.

Some typical optimization criteria are listed as:

- Minimize makespan which is defined as the load on the maximum loaded processor or the completion time of the last job. Makespan can be used to measure the utilization of the system.
- Minimize the total (weighted) cost or total completion times (TWC). TWC is defined as the sum of all the (weighted) completion times of the jobs. This objective indicates the

total holding or inventory.

- Minimize the mean waiting time or maximum waiting time. The waiting time of the a job is the time from arrival to the start of processing.
- Minimize the total weighted tardiness (TWT) or the maximum tardiness. Tardiness is the difference between completion time and due date of a job.

It's known that the load balancing problem for the single criterion problem of minimizing the makespan is NP-hard, and no approximation algorithm can be better than 1.5-approximation, where a ρ -approximation algorithm is one that is guaranteed to produce a solution with objective function value at most ρ times the optimum value [15]. According to the concrete applications, there are many variations of this load balancing problem. The processors may have different capabilities, and thus the cost of executing a job could be distinct on different processors. Jobs may have dependencies, and in this case dependent jobs should be run in some specific order. Moreover, jobs may have locality, and thus some jobs could be assigned on the same or nearby processor to reduce communication cost. The scheduling of load balance is classified as static scheduling and dynamic scheduling. The former suits for applications whose load information is roughly known and nearly unchanged; and the latter works for the applications whose load information is not known at program startup or is dramatically changed during the execution.

2.2 Game Server Background

Online games come in two general flavors: those with a large number of simultaneous players in large, often persistent, worlds, called massively multiplayer games (MMOGs), including massively multiplayer role play games (MMORPGs); and those with a smaller number of players in smaller, often transient, worlds, called (classical) multiplayer games (MPGs). MMOGs are exemplified by MMORPGs like EverQuest [24], while MPGs include first person shooter

games (FPS) like Quake III [12] and real-time strategy games (RTS), but also some role-playing games (RPGs). We focus on large-scale MMOGs that may support vast numbers of concurrent players and even more registered players.

2.2.1 Game Objects

The core of online games is the use of computing engines to *simulate* entities that interact with each other, where the entities are modelled as objects. An object consists of a collection of fields (state variables and attributes) and a set of methods that model behavior. The relationship of objects to one another is specified through three approaches:

- Attributes that indicate those state variables and parameters of an object that are accessible to other objects.
- Association between objects (*e.g.*, one object is part of another).
- Interactions between objects that indicate the influence of one object's state on the state of another object.

A typical multiplayer game *world* is made up of immutable landscape information (*terrain*); characters controlled by players (*PCs* or *avatars*); *mutable objects* such as food, tools, and weapons; mutable landscape information (*e.g.*, breakable windows); and non-player characters (*NPCs*) controlled by automated algorithms. *NPCs* can be either allies, bystanders or enemies, and are not always immediately distinguishable from *PCs*, except by their interactions.

The state of a player includes his position in the world and the state of his game avatar, such as its abilities, health and possessions. Avatar states are often persistent and can be carried along from one game session to another. Similar states exist for *NPCs* and game objects. In general, a player is allowed three kinds of actions: *position change*, *player-object interaction*, and *player-player interaction*. A player interacting with objects (including *NPCs*) or other players may, subject to game rules, change other objects' state as well as the state of his avatar, *i.e.*, drinking from a bottle would empty the bottle and decrease the thirst of the avatar.

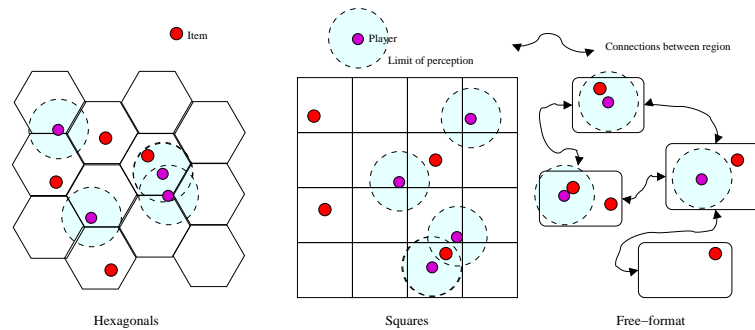


Figure 2.1: Partitions of the game world.

2.2.2 Server Architecture

The client-server architecture is the predominant paradigm for implementing online games, in which a centralized server receives server requests, updates game state and then propagates the new state to clients. Decentralized architectures such as in MiMaze [5] and Age of Empires [21], also exist, but are much less popular. Scalability is an important issue in game servers, because players enjoy complex interactions, detailed physical simulations, and the possibility to interact with a large number of players and objects. The requirements demand powerful servers both in CPU cycles and network bandwidth. Current game servers typically fall into two categories. The first person shooter (FPS) servers, such as Quake servers [12], simulate fast actions and typically support 8 to 64 players per server. On the other hand, the often-clustered servers for strategy or role playing games can support thousands of players, because of a much slower pace and simplified simulation of physical laws. More recent developments include Sony Online’s *Planetside* [25], which combines a FPS with large shared persistent worlds and large numbers of simultaneous players (thousands per world).

One of the simplest approaches to game world partitioning is to split the world into separate mini-worlds with links to other mini-worlds. We refer to this approach as “free-format” partitioning, since these mini-worlds do not require any strong spatial relationships. For example, each mini-world may be a country in the world, and links can be modelled as airports. This works well for games where the world can be abstracted into separate areas.

Where this is not possible, and the world truly is one single contiguous map and strong spatial relationships are required, “true” partitioning is required.

Figure 2.1 illustrates three possible approaches to partitioning a game world. The first two model a contiguous world, and differ only in the geometric shape used for the partitioning. The third illustrates one possible free-format world.

The player’s avatar has limited movement speed and sensing capabilities. In the figure, the area a player avatar can sense is illustrated by the shaded area surrounding the player. Items and players outside this area cannot be detected by the avatar. This means that data access in games exhibits both temporal and spatial localities. Networked games and distributed real-time simulations have exploited this property and applied *interest management* [19] or area of interest (AOI) to game state. Interest management allows us to limit the amount of state any given player has access to, so that we can both distribute the game world at a fine granularity and localize the communication.

Interest management details are highly application dependent [19], but it is reasonable to assume that a player might be interested in state in neighboring regions because its sight extends across region boundaries. Thus, inter-server communication is needed in a partitioned game map for communication of state information between servers to ensure cross-partition visibility for players. Furthermore, as players move between server partitions, they incur player hand-off communication costs between the servers hosting the affected regions.

Chapter 3

Locality-Aware Dynamic Load Management

3.1 Overview

Locality-aware dynamic partitioning is a decentralized algorithm that uses a heuristic approach to shed load from an overloaded server while keeping locality in mind. Locality-aware dynamic partitioning uses locality heuristics in two dimensions: region clustering based on adjacency on the game map and network proximity in the game. The former means that it is desirable to keep adjacent regions together on one server. The latter means that the distributed load balancing algorithm favors localized communication between servers hosting neighboring partitions for load shedding. Note that the two definitions are orthogonal.

A server periodically communicates its load information with neighbor servers hosting adjacent partitions along with data for area of interest consistency maintenance. Each node maintains the current load on its neighbors and a limited subset of other nodes that are currently lightly loaded based on its knowledge. Each lightly loaded node on the locally maintained list has an expiration time. Sufficiently stale information is discarded by each server node. Initially, a game map is divided into blocks and each block contains several continuous regions of the

game map; each block is assigned to one server, and neighbor blocks are assigned to servers close in network proximity. When overloaded, a node favors shedding load to its neighbors if they can accommodate the load shedding without exceeding a safe load threshold themselves rather than any other node in its locally maintained lightly loaded set.

In the following sections we detail our definition and use of load thresholds (i.e., lightly loaded, overloaded and safety load level) in section 3.2, and our overall algorithm for shedding load by an overloaded server in section 3.3. Section 3.4 introduces other algorithms for comparison and we discuss and compare these algorithms in section 3.5.

3.2 Load Thresholds

The definition of an overloaded is done by comparing a node's current level of service with a predefined service level agreement (SLA), in our case, 1 second update interval to each client. If the SLA is exceeded for 90% of the clients, a server is thought as overloaded, and thus we define an *overload threshold* as the server load that satisfies 90% of client updates within the SLA. The highest possible safety level is determined by off-line measurements. The definition of the actual *safety threshold* and *lightly loaded server* is configurable to any value lower than the actual off-line measured highest safety level for normal execution. *Safety threshold* is the target load after load shedding for an overloaded node. Once a safety threshold has been chosen, the lightly loaded threshold is computed from it by the formula below. The formula corresponds to a scenario where after load shedding both the previously overloaded node and the lightly loaded node achieve the safety threshold.

$$Light_load_th = 2 * Safety_th - Overload_th$$

For example, if the *Overload_th* is 100 and the *Safety_th* is 75, the *Light_load_th* would be 50. Both nodes would achieve 75 after load shedding in an ideal case.

The setting for the *Safety_th* depends on how much resource usage we want to maintain. If we want a high overall system utilization, then a higher threshold is desirable. The down side

is that the higher the desired overall utilization we want to maintain, the more adaptations and region migrations we are likely to need. Conversely, the lower the overall utilization, the more aggressive load shedding can be, hence the overloaded node decreases the likelihood of future adaptations (e.g., if more players continue to cluster around the quest).

Note, these thresholds selection does not guarantee a previously overloaded node always achieves the safety threshold after shedding load with a lightly loaded node. Because the load shedding grain is a region instead of a unit of load. Load is usually not evenly distributed on each region. In an extremely poor case, 99% load of a overloaded node may all fall in one single region so that this node is impossible to shed load even with an empty node. But proper admission control over a game region is able to limit this sort of extreme cases to happen. For instance, game designers can limit no more than 100 people in a hall of a palace simultaneously.

3.3 Load Shedding Algorithm

Load balancing is triggered when a node is overloaded as defined above. In this case, either the bandwidth or CPU of the server node is close to exhausted due to excessive number of clients on server. The server periodically checks the current SLA of its players and keeps cumulative averages of the update interval to clients. If 90% of updates arrive late over a time interval then the server triggers the ShedLoad algorithm shown in figure 3.1. In our algorithm the target load after load shedding is the safety threshold (see section 3.2).

3.3.1 Load Shedding to Neighbors

The load shedding algorithm first attempts to shed load to as few of its neighbors as possible. Each candidate neighbor's load should be below the safety threshold level, but not necessarily below the lightly loaded level. This is why shedding to several neighbors may be necessary. The algorithm iterates through all eligible neighbors trying to give each a contiguous region cluster starting with the respective neighbor's boundary regions using a heuristic graph par-

tioning (see section 3.3.3). If the load left after shedding to neighbors still exceeds *safety threshold*, it is continuously shed to one or several nodes of the lightly loaded server set known as described next. Since we initially assign continuous partitions in game map to servers close in network proximity, so first shedding load to neighbor nodes try to keep preserving network proximity for continuous regions during the process of region migrations.

3.3.2 Load Shedding to Lightly Loaded Candidates

Seeking a lightest loaded server occurs as follows. First the algorithm checks the lightly loaded server set maintained locally. It finds the minimum loaded server S_j from this list and contacts it. Since the information kept in the lightly loaded server set may not be accurate, S_j may reject the request because it is not in lightly loaded status anymore. If the server cannot get a candidate from the lightly loaded server set, it floods the network with a `SeekLightLoadedServer` request message for searching a lightly loaded server. These messages are propagated among neighbor pairs and the request message carries the identity and the address of the overloaded server. If an underloaded server cannot be found, this means that the resources are all used and load shedding is not possible.

In the case that a node S_j accepts the load shedding from the overloaded node, the actual region migrations occur as dictated by the region graph partitioning algorithm described next. In the case where load shedding cannot be accommodated by a single remote server in spite of the remote node being lightly loaded (i.e., due to non-uniform distribution of players in regions on the overloaded server), the algorithm iterates and selects a new lightly loaded candidate for load shedding.

3.3.3 Heuristic for Partitioning the Regions During Load Shedding

Each server node maintains a local load graph as an undirected weighted graph $G(V, E, W)$ where V is vertices set, E edges set, and W weight set of vertices. In G , each vertex v repre-

sents a region. If two regions are neighbors, there is one edge e between their corresponding vertices. The weight of a vertex v represents the load of every region.

Problem Statement

We format our requirements as follows:

Problem 1 divide a graph into 2 equal size components.

Given an undirected graph $G = (V, E)$, where each edge e in E has a positive cost $c(e)$. The problem is to find a subset of edges whose deletion separates the graph into two strong connected components of essentially equal size.

Problem 2 Given an undirected graph $G = (V, E, W)$, where each edge e in E has a positive cost $c(e)$; and different vertex v_i has different weight w_i . The problem is to find a subset of edges whose deletion separates the graph into two strong connected components of essentially same sum of *weights* of vertices.

Problem 3 Given an undirected graph $G = (V, E, W)$, where each edge e in E has a positive cost $c(e)$; and different vertex v_i has different weight w_i . We have a constant target value t . The problem is to find a subset of edges whose deletion separates the graph into 2 strong connected components and the sum of weights of one component $\leq t$.

Problem 4 Given an undirected graph $G = (V, E, W)$, where each edge e in E has a positive cost $c(e)$; and different vertex v_i has different weight w_i . The problem is to find a subset of edges whose deletion separates the graph into N strong connected components of essentially same sum of weights of vertices.

Problem 1 aims to divide a graph into 2 strong connected components; problem 2 is a weighted version of Problem 1; problem 3 has a target value t for partitioning instead of bisection of problem 2; problem 4 aims to do a global optimization so that it requires to divide the graph into N components.

Each problem can be strengthened by adding a MIN-CUT constraint which means the set of cut edges has minimum communication cost. MIN-CUT is proposed to minimize the

communication cost between regions. Moreover, another goal of minimizing re-partitioning cost can be added to each problem if we compare previous partitioning with current one, and define the cost as the number of vertices which change their set in the new partitioning. These problems are some variations of balanced partition or balanced CUT which is proved to be NP [23]. The main difference is that we emphasize that the partition results are strong connected components which are not guaranteed by most of approximate algorithms in literatures [23].

Heuristic for Partitioning the Regions

Our heuristic algorithm aims to solve Problem 3 but the two strong connected components can be relaxed to a desired number of strong connected components. Every server exchanges its local load graph with neighbors and maintains their load graphs in addition to its own. We quantify locality maintenance in terms of the number of strongly connected components in the overall game map graph. Ideally, during load shedding, the following three constraints would be satisfied:

- the same number of strongly connected components is maintained after load shedding as before.
- the minimum number of region re-partitions occurs.
- the load shedding target is achieved.

The three constraints may often conflict with each other. Shedding with a lightly loaded node of the system may easily satisfy the third constraint, but it will increase the number of strongly connected components if the lightest node is not its neighbor node.

We use the following graph partitioning heuristic algorithm to determine a load shedding schedule given the above constraints and optimization goals. If any acceptable schedule is found for shedding load to either a neighbor or remote node as the case may be, the region repartitioning to neighbors or remote underloaded nodes occur accordingly. Otherwise the load

shedding returns an error, by signaling that the `leftLoad` remaining on the node is unchanged. The algorithm proceeds with the next option as shown in the `ShedLoad` pseudocode.

Our graph partition problem is to divide a graph with a desired number of strongly connected components. The sum of weights of the component to be shed should be as close as possible to the target sum of weights to shed for reaching the safety threshold. In the case of shedding load to a neighbor denoted by S_j , we set the root of our search as one of the regions on the *boundary* of S_j . In case of shedding to remote servers a random region at a boundary is chosen. We then use breath first search (BFS) to get a partition such that the total weights of the partition we shed plus the weights of S_j is smaller than the *Safety threshold*. While the total weights of the regions of the part remaining on the local server is greater than the target load we continue adding to the partition to be shed by traversing regions in BFS order. After it, we count the number of strongly connected components of the new graph partition for the two nodes. If the number of strong components exceeds the desired value, we try to find another partition by reselecting a root or tracing back to a previous step of BFS.

3.3.4 Optimization for Maintaining Strong Locality

Directly shedding load with a lightly loaded remote non-neighbor node will increase at least one strong connected components on the remote node and may cause performance degrading since the communication boundary of this remote node increases and thus it has to send more region area of interest information to more neighbor servers. To maintain strong locality, we introduce an optimization technique. Before accepting the load of an overloaded node, the lightly loaded node first sheds its all regions to its neighbors so that the number of strong connected components is same after shedding. This optimization maintains strong locality while it may increase the total number of migrations. In case that migration cost is high, the optimization will bring serious side effects in performance and should not be adopted in such an environment.

3.4 Other Static and Dynamic Algorithms

In this section, we introduce other dynamic load balancing algorithms for comparison with the main Locality-Aware dynamic partitioning algorithm. We also introduce some baseline static partitioning algorithms. By considering different optimization goals we explore different trade-offs with each algorithm, in order to demonstrate the relative impact of different optimization strategies. We consider an algorithm that optimizes global load balancing, and an algorithm that optimizes the migration cost during load balancing. In contrast, as we have seen, Locality-Aware partitioning optimizes locality.

3.4.1 Dynamic Uniform Load Spread (Spread)

Spread is a dynamic load balancing algorithm that aims at optimizing the overall load balancing through a uniform load spread in the server network. It takes only server load into account and ignores locality when making migration decisions. Load shedding is triggered when the number of players exceeds a single-server's capacity (in terms of CPU or bandwidth towards its clients) just as in our main dynamic partitioning algorithm.

The algorithm attempts to uniformly spread the players across all participating servers through global reshuffling of regions to servers. The algorithm is meant to be an extreme where the primary concern is the global optimum in terms of the minimizing the load of the highest loaded server in the system, i.e. minimum makespan. There are no attempts at locality preservation in either network proximity or region adjacency.

The algorithm is a bin-packing algorithm that needs global load information for all participating servers. The algorithm starts with one empty bin per server, then in successive steps takes a region that is currently mapped at any of the servers and places it in the bin with the current lightest load. This is done iteratively until all regions have been placed into bins. After the global reshuffle schedule is created, each bin is assigned to a particular server and the corresponding region migrations occur. While the algorithm could be further optimized to in-

clude only a subset of servers (e.g., just neighbors and their neighbors) into the region reshuffle process, we currently involve all servers in this process.

This global reshuffle schedule will lead to a large number of region migrations. To reduce the number of region migrations, we could add a limit K on the maximum number of region migrations in one global schedule as suggested in [1]. However, it is not clear how to define the maximum number K of region migrations in reality because region migrations could simultaneously happen on different server pairs and thus it is hard to give a fixed K in terms of migration performance.

3.4.2 Dynamic Load Shedding to Some Lightest Loaded Node Known (Lightest)

Lightest is a dynamic load balancing algorithm that attempts to optimize the cost of region migrations by prioritizing shedding load to a single server instead of several servers. Clustering of adjacent regions is maintained whenever possible but is of secondary concern compared to load shedding to a single server.

An overloaded server tries to shed load directly to the lightest loaded node known. The precondition is that this node's load has to be below *light load threshold*. Note that our definition of *light load threshold* try to ensure that a single light node should be able to accommodate a sufficient load shedding from an overloaded node. While this is true in most cases, depending on the actual load distribution of regions, if some regions are more overloaded than others, more than one light nodes may be involved in a load shedding schedule.

The lightest loaded node may in fact be a neighbor, but the Lightest algorithm does not give preference to neighbors when shedding load except in case of load ties. Instead, the overloaded node prefers shedding a sufficient portion of its load to a single server even if this server is remote and even if this implies some declustering of the shedded regions. Regions of the overloaded node are scanned in order and placed into a bin to be assigned to the lightly loaded node. Thus Lightest has similarities with Spread by using a form of bin-packing, but it

uses a single bin and attempts to keep region clusters together by scanning adjacent regions in sequence. On the other hand, if a region cannot be placed in the bin due to exceeding the safety threshold, a subsequent region is selected, hence sacrificing on region locality. In contrast, the main Locality-aware algorithm prioritizes region locality.

3.4.3 Static Partitioning

Several static partitioning of regions to servers are possible: static block partitioning, row-based or column-based static partitioning and cyclic partitioning. We have shown in section 6.3 and [4] that no single static partitioning algorithm performs well for all types of scenarios in terms of hot-spot location and server environment. In our study, static partitioning algorithms had inconsistent behavior across environments and hot-spot locations but performed extremely poorly in at least one experiment. In this thesis, we use static block and cyclic partitioning as two general static load partitioning algorithms for a baseline comparison with our dynamic partitioning algorithms.

3.5 Discussion

None of the heuristics for any of the dynamic partitioning algorithms introduced uses any heuristics hard-wired for a particular environment. The same algorithm will trigger and alleviate locally perceived overload if “non-standard” or heterogeneous environments are used (e.g., a peer-to-peer environment as opposed to a centralized server cluster). On the other hand, from the dynamic load balancing versions, the main Locality-Aware algorithms may be more appropriate in distributed server environments because as opposed to Spread, it requires only localized information from neighbor servers to make migration decisions.

There is a trade-off between shedding to potentially several neighbors as in Locality-Aware, versus shedding to a remote server node with lower load than any of the neighbors as in the Lightest algorithm. Shedding to several neighbors may imply a higher migration cost, but may

be the best in terms of region clustering type locality and communication cost during load shedding. The higher migration cost may be caused by a chain effect. The neighbor node of the original overloaded node becomes next overloaded node and continue to shed its load to its corresponding neighbors, and similar processes happen again to its neighbor's neighbor. This chain effect will eventually disappear as a wave, and the system will finally become stable, but it may be slow and incur more migrations than simply shedding load to a lightest loaded node.

Neighbor servers exchange lightly loaded node information with each other along with the other information exchanges about each other. Thus, information about lightly loaded nodes in charge of remote regions of the map propagates through the network. Neighbor load information is likely to be more accurate than information about any remote lightly loaded node. This is because the load information on servers that host remote areas on the game map propagate slowly through neighbor-to-neighbor "epidemic-style" communication.

Shedding to the lightest loaded node as in the Lightest algorithm may involve on the down-side including some region locality loss and communication penalties such as a search if the local node currently does not know of any such light node. This will occur in two cases:

- the case where the overloaded server's current lightly loaded set is empty.
- the case where load information about the remote nodes may be inaccurate, because their load may have changed since they were put on the list.

Hence, there is a probability that the load shed will be refused while load information about neighbor servers is periodically updated hence more up-to-date.

In the Lightest algorithm, load shedding may induce subsequent penalties in terms of inter-server communication created by region de-clustering compared to our main Locality-aware algorithm. On the up-side, the migration cost may be minimal because in many cases, load shedding involves only one node and message aggregation can occur during migration of regions. All three algorithms are fast, the Lightest and the Locality-Aware running in sub-linear time in terms of the number of server participants, and the spread algorithm can achieve 2-

approximation with a running time of $O(n \log n)$, where n is the total number of regions.

```

ShedLoad(targetLoad) {
    //Attempt to shed to neighbors
    ShedtoNeighbors(targetLoad);
    //leftLoad is updated after each shedding
    if (leftLoad <= targetLoad) stop;

    //shed to the lightest loaded nodes known
    //Flooding to find a set of lightly loaded nodes
    //if local lightly loaded server set is empty
    SeekLightestLoadedNodes();
    if (not found) return; //No more resources available
    //shed to lightly loaded nodes known
    ShedtoLightNodes(targetLoad);
}

ShedtoNeighbors(targetLoad)
{
    while ( leftLoad > targetLoad and some neighbor unvisited) {
        Sj = GetLightestLoadedNeighbor();
        HeristicGraphPartition(Sj, targetLoad);
    }
}

ShedtoLightNodes(targetLoad)
{
    while ( leftLoad > targetLoad and some light node unvisited){
        Sj = GetLightestLightNodes();
        HeristicGraphPartition(Sk, targetLoad);
    }
}

```

Figure 3.1: Pseudo code for Locality-Aware Load Shedding.

Chapter 4

Implementation

We implement our load balancing algorithms within the context of SimMud [14], a simple game developed at University of Pennevenia. SimMud includes a cluster-based game server and clients, and is implemented in Java using UDP to communicate between clients and servers. We extend SimMud with multiple regions by adding a scheduler and allowing multiple regions to reside on a single server. Players are allowed to migrate between different regions, and regions can be migrated between servers. This section starts with a brief overview of SimMud, followed by the extensions to enable dynamic load balancing.

4.1 Game objects in SimMud

Game objects include players, food, quests, and roadblocks, each of which has its own position in the game map at any given time. Roadblocks exist in fixed locations in the map, and as in a maze, they restrict the movement of players. Currently, roadblock positions are determined before the game begins, and roadblocks cannot be added or removed while the game runs. Thus, roadblocks are a part of the terrain. The terrain could be made more complex by adding the third dimension (height), buildings, doors, etc., but the basic idea of the roadblock is that, like terrain features, it puts constraints on where players can move.

In SimMud, food objects are similar to roadblocks in that they cannot be moved to new



Figure 4.1: Screen shot of SimMud. The dots represent players. Blocks represent roadblocks. Food is represented by coffee mugs.

positions. However, they can be eaten by players, making them a form of mutable landscape information. Each food item has a *quantity* attribute that can be reduced as players eat it. Food is removed when this quantity reaches zero. Servers can also add new food objects during play. These food objects can be thought of as a simplified representation of a larger variety of objects, such as money, tools, and weapons. Currently, the food objects cannot be carried by players, but we may choose to extend the game with portable objects in the future.

Players are the most complex game object in SimMud. Players are able to move, eat, and fight other players. SimMud includes both a robotic, AI-controlled player, which we will call the *robot*, and one controlled by user keypresses. In the sense that robots are controlled by an algorithm rather than a user, they can be considered as NPCs; however, in all other respects they resemble regular players, and therefore (at least in this simple game) can be used to simulate them. Each player has a health attribute which can be increased by eating or decreased as the result of a fight. Since SimMud players can move and perform other actions that change their attributes, they are a simplified representation of players in more complex games.

Quests are a mechanism for causing robots to flock to one region, thus loading the server hosting that region. Like food, a quest is a stationary object, and it can be added to the map

during game play. Each quest has a duration, after which it expires and is removed from the map. While the quest is present, robots are attracted and will move toward the quest. Quests are useful in that they allow simulation of the flocking behavior that occurs in real games. Human players may flock to one location because of an invitation to join other players (e.g., to watch a fight), or in general because of some event or object of interest that they know to be there. Since the motivations of robotic players are easy to control, quests can be used to simulate flocking without difficulty.

Players, including robots, interact in SimMud by eating and fighting. Eating food involves sending a request to the region's server, which updates the player's health and decreases the quantity of the food. Concurrent requests are serialized at the server to avoid race conditions. Fighting is started by one player (the attacker) who sends a request to the server. The server determines who wins the fight, and decreases the health of the losing player. Players' movement within a region does not require requests to the server, although players must send position updates to the server periodically. Changes in food and in players' positions and health are periodically sent by the server in general region status updates. The interval between updates can be adjusted to reflect different game implementations.

4.2 Multiple Regions in SimMud

To show the effects of player migration and clustering of players on one server, we extend the single server game to multiple servers. Each server controls one or more rectangular *regions* of the map. Servers are given the contact information for each of their neighbors in east, west, north, and south directions (if they have any) at start-up time. Robot players start out in one region, but are able to cross over to other regions as they move around the map.

We use quests to attract robots to a single hotspot in one region, thereby loading one server more than the others. In a multi-server SimMud game, quests are created and assigned to the appropriate server by a separate QuestMaker program. The QuestMaker is configured with

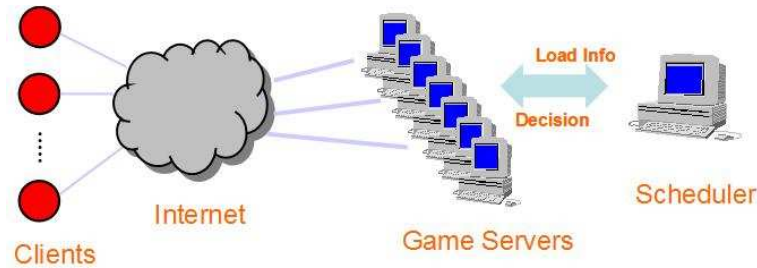


Figure 4.2: Multiple server SimMud architecture.

information about the game servers, which it reads from a file. It multicasts quest information to all servers. The server whose region actually contains the quest will include the quest on its map, but other servers will also notify their robot clients of the presence of the quest and its location in periodic updates. Quests can be added and removed from the game over time. This setup emulates what happens in real games where hot spots appear spontaneously and move over time.

Since only one quest is created at one time, we expect that quests will cause robots to gather at one server if they are given enough time to do so. A single quest is sufficient to demonstrate, for example, the load imbalance caused by hot spots. Multiple quests can be added easily, and robots can be programmed to approach any individual quest.

While quests can be “seen” by robots even in different regions, food and other robots cannot, in this particular version of SimMud. Therefore, food and attacks on other players work in essentially the same way as in the single-server version — food and players in any given region can be “seen” only by players that are present in that region. The positions of these game objects are multicast by the region’s server to its own robots, as before. This is common to current game servers, where each node hosts an isolated game world. However, we are also extending the game to allow some visibility of objects in neighboring regions.

The multiple server SimMud, as illustrated in Figure 4.2 involves a scheduler in addition to the region servers. Although the locality aware load balancing can rely on distributed decision making, we include a centralized scheduler in order to accommodate centralized scheduling

algorithms such as Spread. In order to reduce the load on the scheduler, it is not involved in player migration. Instead, each server keeps track of their own load and report to the scheduler periodically. Neither does the scheduler provide a naming service. The region to server mapping is stored locally on each server. Because we assume players display spatial locality in movements, each server only has to keep track of servers that host neighboring regions, which are called *Neighbor Server*. The multiple region game map is made up of one or more adjacent rectangular *regions*. Players may migrate between regions as they move around the map.

4.3 Player Migration between Regions

Quests or exploration may lead a robot or a human player to try to move outside its region. To do so, the player sends a request to its current server, containing the direction in which it is trying to move. The server checks whether it has any neighboring server in the requested direction. If so, it contacts its neighbor with the player's information and the appropriate new position for the player. The player is then removed from the old region and added to the neighboring region. Requiring servers to directly contact each other during player migration prevents the player from falsifying their states during migration. In addition, we added timeout and retransmit to handle possible message delays and client failures during the handoff. On the other hand, if the player's current server has no neighbor in the direction the player wishes to move, the server replies to the region-switch request with a failure message. This will occur whenever the player is trying to move to a nonexistent region, i.e., off the map. In that case, the player will remain at its current position. The progress is illustrated by Figure 4.3. More details are given as follows:

- Msg1: The client (i.e. player) sends a *Region Switch Request* Message to its current hosting server 1 for requesting switch to a region on server 2.
- Msg2: Server 1 checks the client's current state, and sends a *Region Switch Reply* Message to the client.

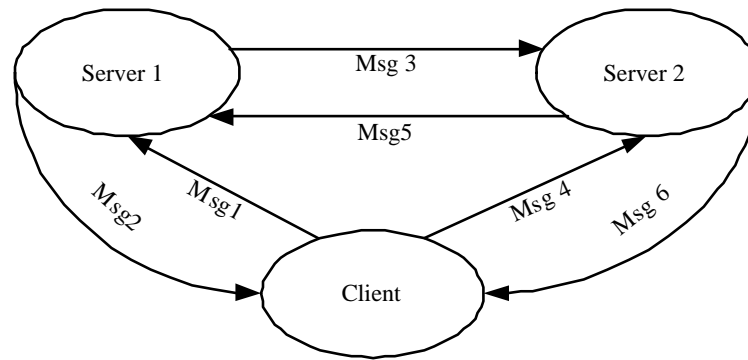


Figure 4.3: Player migration between regions.

- Msg3: If server 1 agrees the request, it will send a *Migration Forward* Message to notify server 2
- Msg4: If the client receives Msg2 with agreement, the client sends a *Migration Join Request* Message to server 2 whose address is contained in Msg2.
- Msg5: If server 2 receives Msg3 and Msg4, and it accepts the migration request, it will send a *Forward Approval* Message to server 1.
- Msg6: At the same time with sending Msg5, server 2 sends a *Join Approval* to the client.

According to the implementation, the duration for a player to migrate from server 1 to server 2 is around two roundtrips, with one between the player and server 1, and the other between the player and server 2.

4.4 Migration of Regions

Region migrations may or may not involve the scheduler. We present the former case in this paper because its activities consist of a superset of the latter. Figure 4.4 illustrates the message sequence of a scheduler-initiated region migration. The messages are numbered by their time line in the process. The odd numbered messages are quests, and the corresponding even numbered messages are replies. Messages of the region migration are listed as follows:

- Msg1: The scheduler sends a *Region Migration Decision* Message to both the original server (i.e. old server) and the new server of a region to initiate the region migration.
- Msg2: The server sends a *Decision ACK* Message back.
- Msg3: The old server sends a *Region Migration Request* Message to the new server. The message contains the region status. At this time, the old server stops all update activities of this region (including updating the region status, sending reply and Region Status Update Message), and just saves all of incoming messages from players and other neighbor servers of this region.
- Msg4: The new server sends a *Region Migration Reply* Message back to the old server. If the new server agrees the migration, then it uses this region state to update corresponding data structures.
- Msg5. Upon receiving a Region Migration Reply Message with agreement, the original server multicasts a *Region Migration Notify* Message to all clients and neighboring servers of this region to notify them the new server's IP address.
- Msg6. The player or the neighbor server sends *Region Migration Notify ACK* Message to the old server.
- Msg7. Upon receiving the new server's address, both clients and neighbor servers contact the new server by sending a *Player Region Migration Request* or *Neighbor Region Migration Request* Message.
- Msg8. The new server sends *Player Region Migration Reply* or *Neighbor Region Migration Reply* Message to the corresponding player or neighbor server. When the player or neighbor server receives this Message, the player or neighbor starts normal activities with the new server.

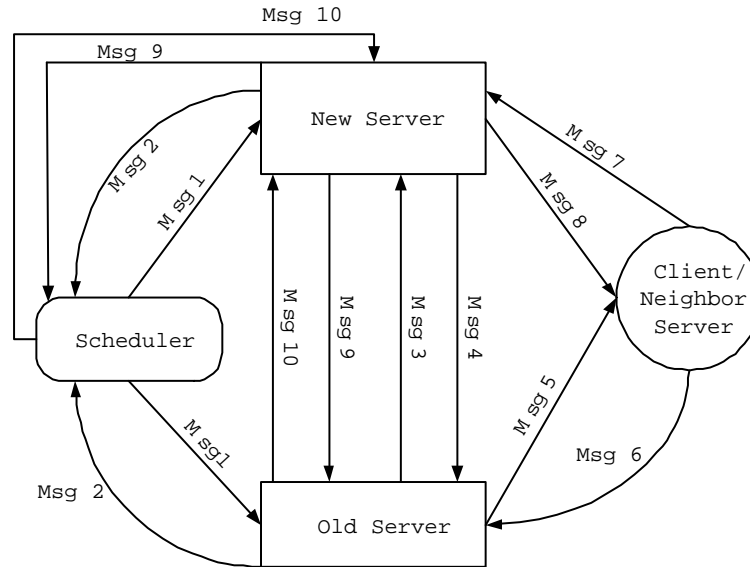


Figure 4.4: Region migration.

- Msg9. Finally, after all relevant clients and neighbor servers have contacted the new server, it sends a *Region Migration Success* Message to both the scheduler and the original server to signal the completion of the migration.
- Msg10. The *ACK* message for Msg9. The old server removes the region and sends ACK back. The scheduler updates the load distribution of whole system and sends ACK back.

Explanation

Msg1 and Msg2 are designed for the global scheduler, they do not appear in a distributed dynamic load balancing algorithm. If the new server rejects the migration request, the migration progress ends by Msg4. All messages have a sequence number so that duplicate messages can be detected and then discarded. For the old server, because it stops update the region status since sending Msg3, it will forward incoming messages of this region to the new server until the migration ends. These forward messages are omitted in Figure 4.4. The new server starts sending updates to known players since sending a Msg4 with agreement.

Players and servers all maintain their state machine for migration during different phases of the progress as shown from Figure 4.5 to Figure 4.7. Upon receiving Msg5, a client sends

Msg6 and Msg7, and set timeout timer for Msg7, and then enter *Migration Request* state; the client goes back to normal state upon receiving Msg8. For the old server, upon receiving Msg1 from the scheduler, it sends Msg2 back and sends Msg3 to the new server, and set timer for Msg3 and then enters *Migration Request* state; upon receiving Msg4 with agreement from the new server, it multicasts Msg5 to all players of the region, sets timer for Msg5, and enters *Migration Notify* state; finally it receives Msg9 from the new server, then sends Msg10 back and remove the region, and goes back to the normal running state. For the new server, upon receiving Msg3, it enters *Migration Reply* state and sends Msg4 back; upon receiving Msg7 from all players and neighbor servers of this region, it enters *Migartion Success* state, and sends Msg9 and set its timer; finally upon receiving Msg10, it declares the migration ends and goes back to the normal running state. Since some player or neighbor server may be not able to connect to the new server although it's rare because all servers are public available, a migration timeout is set to avoid servers are stuck in migration states.

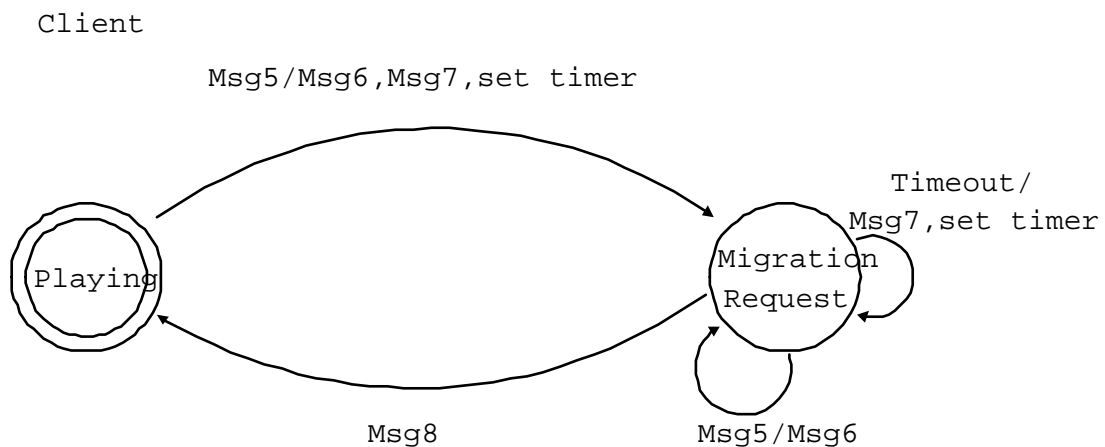


Figure 4.5: Region migration state diagram (client)

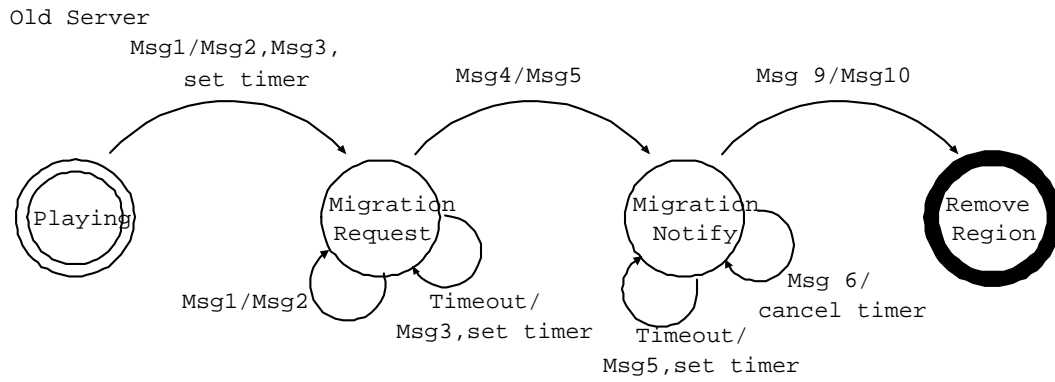


Figure 4.6: Region migration state diagram (old server)

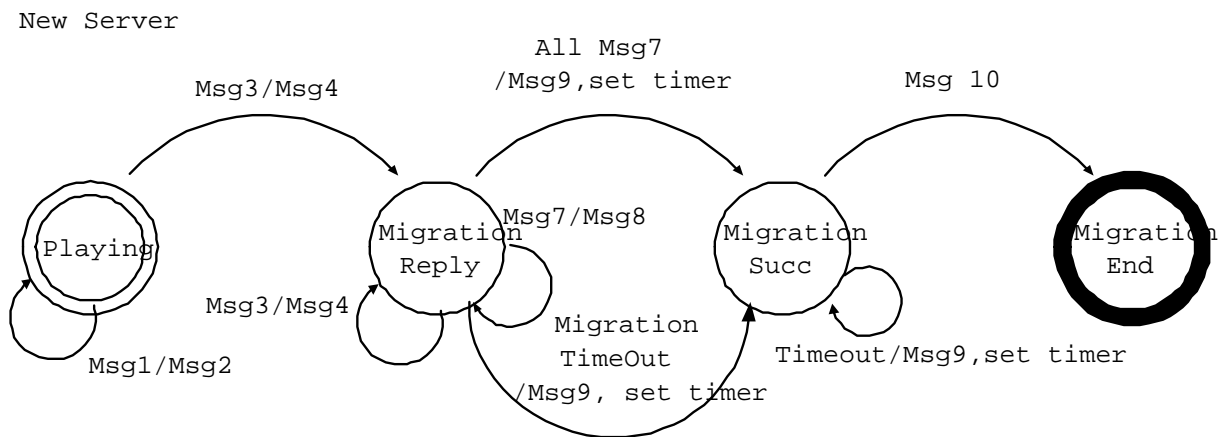


Figure 4.7: Region migration state diagram(new server)

4.5 Discussion

We first discuss the potential unfairness caused by region migration. From above implementation, we know region migration may change the order of messages from different players. For example, we assume the delay between players and servers are same, player A sends a food eating request at time t_1 , and player B also sends a food eating request at time $t_2 (t_2 > t_1)$; shortly before t_1 , region migration happens. Player A does not receive Msg5 before t_1 , and thus continue sending requests to the old server while player B is lucky to receive Msg5 and connect with the new server before t_2 , and hence sends requests to the new server. Although old server will forward player A's requests to the new server, player B's request arrives earlier if the delay between the old server and the new server is greater than $t_2 - t_1$, and thus B eats the food while A loses. We argue this kind of unfairness can also be caused by network delay jitter between players and their server, and it is usually transparent to game players.

The correct FIFO order of messages from one player can be guaranteed by using sequence number. Although region migration does not influence the consistency of the game, it has a risk to lose some players in some case because some player may be not able to connect to the new server mainly due to permanent network link failures between them. In reality, all servers should be public available and thus the possibility of losing players is low.

There are several optimizations for the implementation of region migration. First, since the new server may reject the migration request especially if using a distributed load balance algorithm, we could further optimize the migration process to separate Msg3 with region states transmission to avoid unnecessary update suspension in case of rejection. Second, we could aggregate migration messages of several regions if they are moved from the same source to the same destination. Third, since Msg3 includes region states information, and thus Msg3 size is big and its transmission seriously affects the start time of sending updates from the new server, we can use TCP to transmit Msg3 instead of UDP since TCP works well in reliably sending messages with large size.

Chapter 5

Experimental Results

This section presents our experimental setup and results with the actual multi-server SimMud implementation. Because the largest testbed we have access to consists of 32 nodes, we are unable to fully demonstrate the benefit of dynamic load balancing with experiments. We supplement the experiments with simulations results of up to 6000 players.

Our testbed consists of a 32-machine cluster of dual 1GHz Pentium III Linux 2.3.18 machines connected by a 100Mbps Ethernet. To allow us to experiment using large numbers of players, we have constructed a simple robot player. The robot player runs on top of a normal client, and the only difference compared to a human player is the behavior. Our robots use a variant of a simple path finding algorithm [17] to move around the map and avoid roadblocks. Robots are programmed to explore, seek out food, fight weaker opponents and flee from the stronger. If a quest is announced, this will override all other priorities, and the robot will move simplymindedly to the quest goal. Up to ten robots can be run on a single node without slowing down player activities.

5.1 Single Server Experiments

Our robots run on clients that propagate two position updates per second. The server aggregates and disseminates position updates every 500ms.

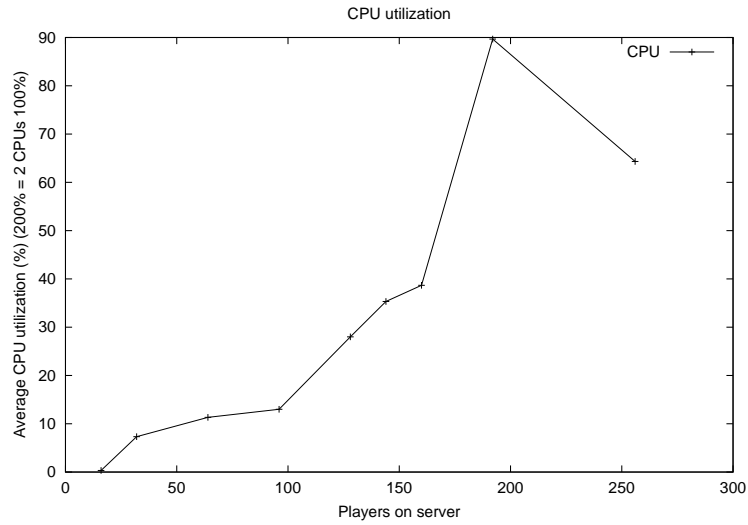


Figure 5.1: CPU utilization. As can be seen, CPU is not a bottleneck. The drop at 256 is due to clients terminating because of network overload.

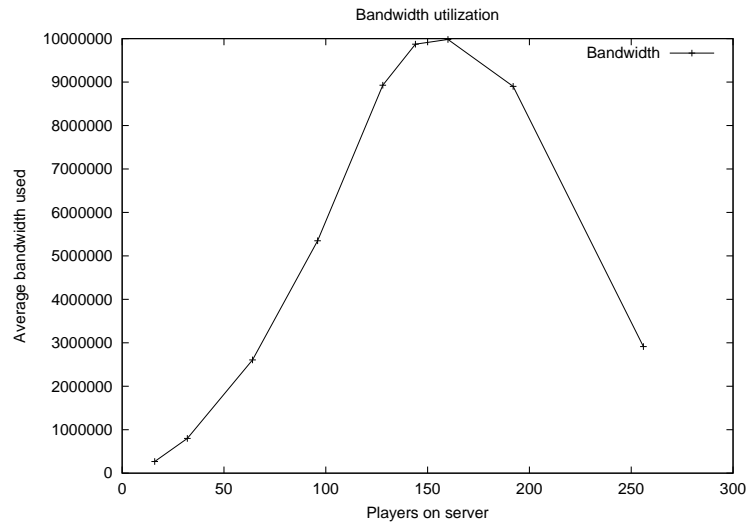


Figure 5.2: Server bandwidth utilization. When requirements exceed available bandwidth, delays skyrocket and losses cause clients to terminate.

| | | | | | | | |
|------------------------|-----|-----|-----|------|------|------|------|
| No. of Players | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
| State Update Size (KB) | 4.5 | 5.2 | 7.2 | 10.0 | 16.8 | 34.2 | 60.0 |

Table 5.1: Size of server state updates.

In this experiment, we vary the total number of robots between 4 and 256 to explore the resource consumption of SimMud. The server performance, averaged over 15 minute runs, are shown in Figures 5.1 and 5.2. The client updates are approximately 800 bytes each and consist of the entire player object. The size of server updates, shown in Table 5.1, depends on the number of players — 4.5 KBytes for 4 players up to 60 KBytes for 256 players¹.

Our experiments show that the server can handle up to 128 players, but when increasing further, delays increase sharply — from 214 *milliseconds* for 128 players to 33 *seconds* for 144 players. Conversely, Figure 5.2 show that at 144 players, the bandwidth consumption curve flattens out as the limits of the network are reached, and then falls as clients terminate due to overload.

CPU consumption remains low throughout the experiment, with peak consumption being less than half the available cycles from the dual processors. CPU can be made the bottleneck by either increasing game world complexity, or by adding more players and either increasing available bandwidth, or reducing per-player bandwidth consumption. The results are similar — as delays increase and clients cannot update game world in time, however, in the case of overload, the server can unilaterally mitigate the situation by reducing the update frequency and/or switching to faster, but less accurate, algorithms. This issue is orthogonal with our load balancing algorithm.

We are improving congestion control methods used in SimMud, and try to further reduce the size of update messages. Based on our observation, the trend that the response delay dramatically increases under overloaded cases is very similar in an improved versions although the

¹Message sizes can easily be reduced and server updates can multi/broadcast instead of unicast, but in this experiment we are trying to saturate the network.

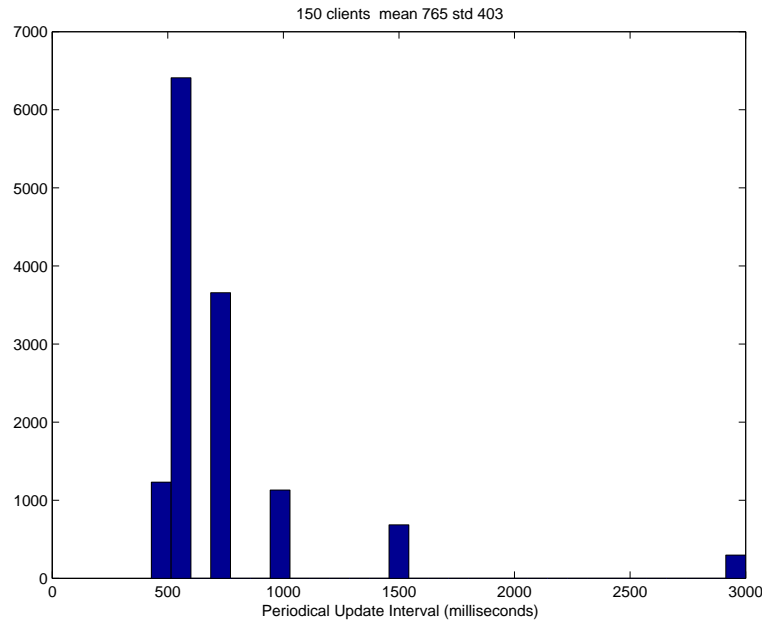


Figure 5.3: Periodical update interval (150 clients)

overload threshold is different. Figure 5.3 to Figure 5.5 shows that the histogram distribution of periodical update interval varies with the increase of the number of players under an improved SimMud version which can support up to 175 players. From Figure 5.6, we can see that when the number of players is smaller than 100, the mean of update interval are stable around 500 milliseconds and the variation is small; when the number of players approaches 175, the mean gradually increases and the variation also greatly increases; finally when the number of players is close to 200, the mean rises to more than 2 seconds, and the variation is quite large. In overloaded cases, many players may lose contact with the server for a very long time because of network congestion. This explains the reason that we only have a small number of samples for Figure 5.5. This sort of poor performance will seriously affect the interests of human players for the game.

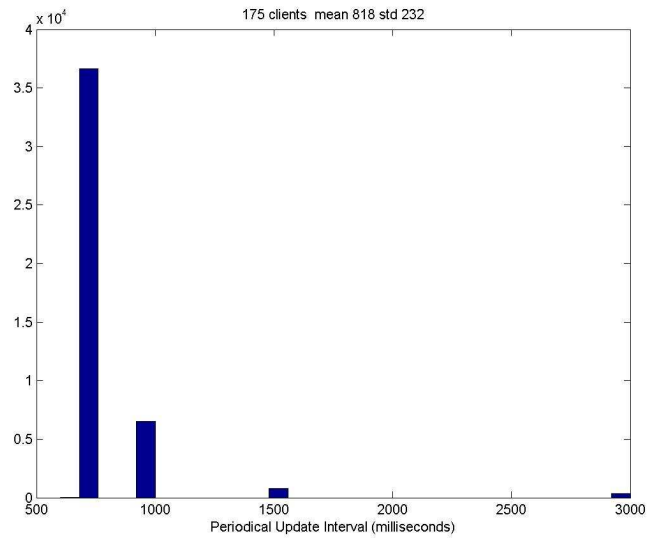


Figure 5.4: Periodical update interval (175 clients)

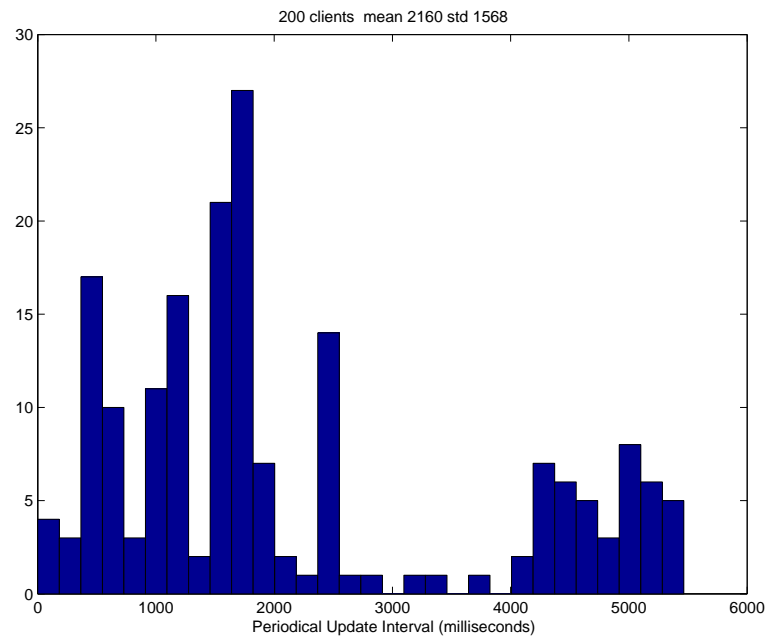


Figure 5.5: Periodical update interval (200 clients)

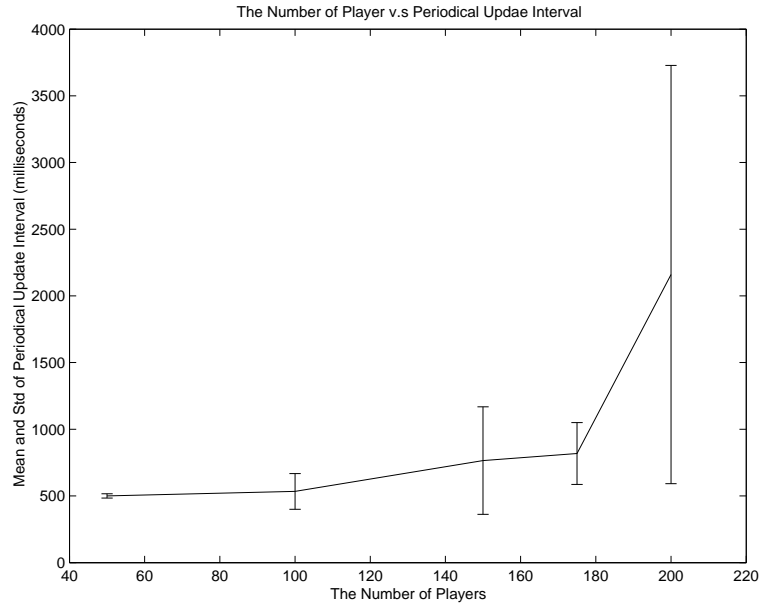


Figure 5.6: The number of players v.s. periodical update interval

5.2 Multiple Servers Experiments

A four region world distributed on four servers is set up as shown in Figure 5.7 with 128 robot players evenly distributed, seeking food and fighting. A temporary goal (quest) is set up in the northwest (Server 1) region at time $t \approx 65$ lasting 300 seconds, and players flock towards it. This type of flocking occurs in many MMOGs, where players have to go to a certain area to earn points or reach a new level of the game. Traversal of a region in this case takes about 100 seconds, unless detoured by roadblocks or opponents, meaning most players can reach the goal in 300 seconds. We sample player density, update interval and CPU/bandwidth consumption for each server every 3 seconds.

In Figure 5.8, the player density is illustrated with time. Since players that originate from the southeast region must traverse an intermediate region, these intermediates will be populated longer. At the end of the quest, 123 players have made it to the right region, with 5 stragglers still on their way. CPU load and the bandwidth (Figure 5.9 and 5.10, respectively) on each server correlate with player density. Server response time stays within 250ms for the duration

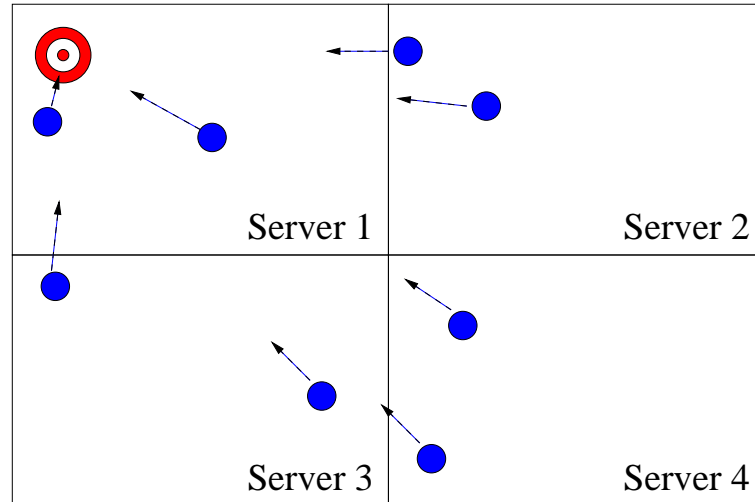


Figure 5.7: Robot players are moving toward a target (quest), in the northwest region (Server 1), many of them crossing region/server boundaries in the process.

of the experiment. Player migration takes about 100 milliseconds when both servers are lightly loaded. During the quest, player migration occurs less than once per second and the (960 byte) message(s) incur only a minor impact on bandwidth.

This experiment shows that the flocking behavior of players poses a significant challenge on game server resource management. It may requires several times of resources during flocking than normal cases. Static resource over-provisioning technique is not desirable here since it is difficult to predict when and where the flocking will happen and how long it will last.

5.3 Region Migration

| | | | | | | | |
|----------------------|-----|-----|------|------|------|------|------|
| No. of players | 1 | 10 | 20 | 40 | 60 | 80 | 100 |
| Time to migrate (ms) | 415 | 439 | 514 | 535 | 715 | 743 | 778 |
| Message size (KB) | 4.5 | 7.7 | 11.1 | 18.0 | 22.8 | 31.8 | 38.0 |

Table 5.2: Cost of region migration.

As we discussed in Section 4.4, in case of enough resources (CPU, bandwidth) are avail-

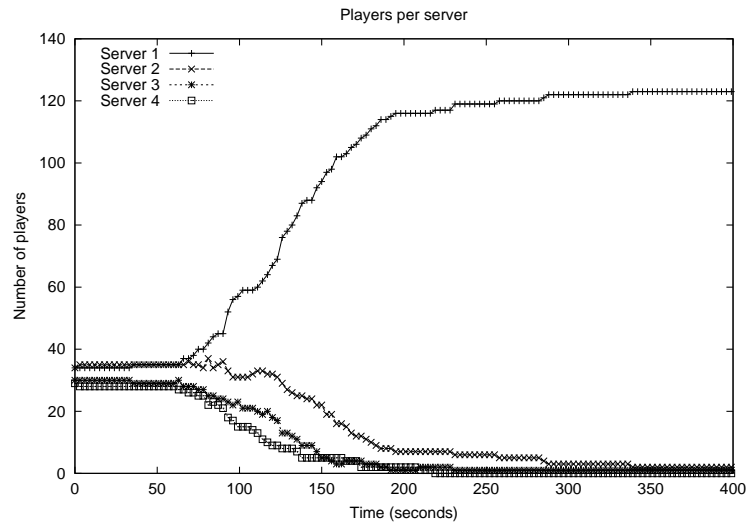


Figure 5.8: 128 (robot) players distributed over four servers. At time $t \approx 65$ a quest in Server 1's region is created, and players flock there for 300 seconds.

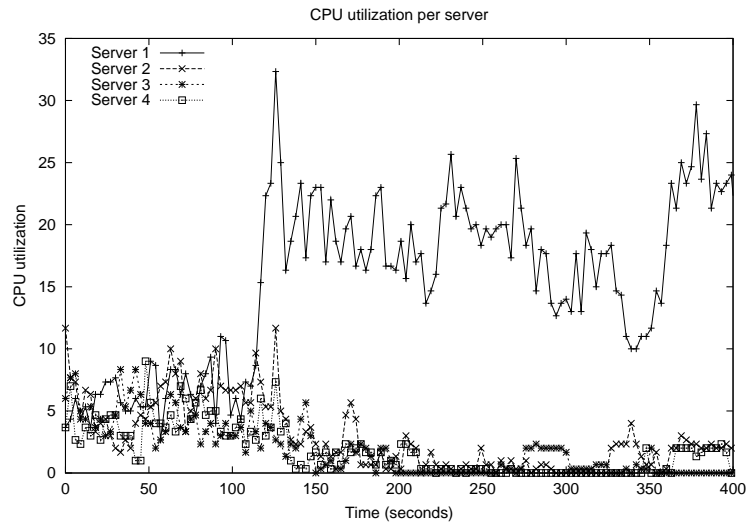


Figure 5.9: As more players join Server 1's region, its CPU consumption goes up, while the others' decline.

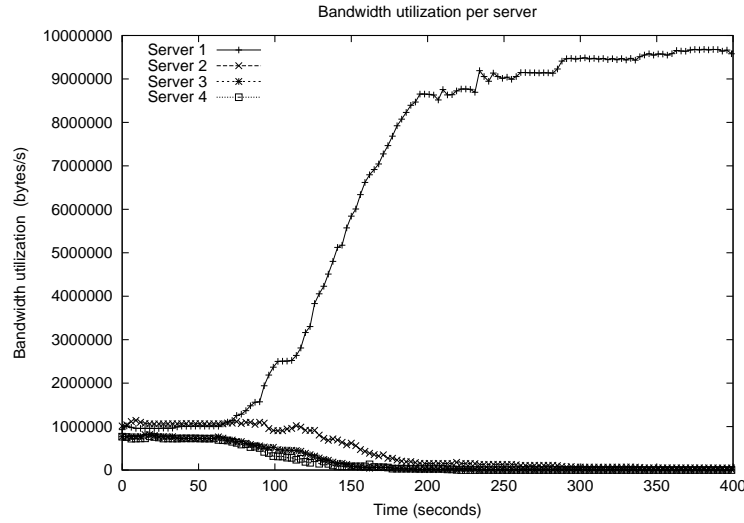


Figure 5.10: We come very close to the maximum sustainable bandwidth. Average delays (not shown in figure) from Server 1 jump from $5ms$ at $t = 50$ to $180ms$ at $t = 700$.

able, the cost of region migrations mainly depends on the size of *Migration Request* message that is sent from the original server to the new server, contains region state information, and dominates the size among other migration related messages. The size of this message is heavily dependent on the number of players in the region because the more players, the more player states information is contained. Table 5.2 shows the migration time as well as the message sizes of region migration under different number of players. The migration process time is measured at the original server from the start sending *Msg3* to receiving *Msg9* which marks the end of the migration.

However, if network congestion happens or packet loss rate is high, the duration of region migration may be much longer than we measured in Table 5.2 which is measured under no timeout happens. We observed some migration messages are retransmitted due to timeout, and some players try several times to set up a connection with the new server under network congestion cases.

Chapter 6

Simulation Results

We have developed a configurable simulator that is powerful enough to model hundreds of servers, thousands of clients and the network between them. Each server accepts client requests, and sends periodic state updates to all corresponding players it hosts. Each server performs its normal functions of computing player moves, eat and fight requests which are all simulated in terms of their CPU costs only. In addition, servers hosting neighbor regions also periodically send visibility information and other information to their neighbor servers as in our experimental environment. The average size of these messages is measured and used in the simulator.

6.1 Simulation Methodology

6.1.1 Resource Usage and AOI Simulation

The simulator maintains a queue of events ordered by their simulated start time. We currently simulate just two types of resources, CPU and network. Whenever an event is triggered (e.g., by a client request) a record specifying the event's future start time is placed on the queue. The simulator estimates a completion time for the event, using the same event execution time as measured in the experimental testbed. This calibration of the simulated system against mea-

surement of the real server allows us to simulate CPU and network contention by extrapolating from server response time measurements obtained in the real system with increasing player load. A CPU cost is assigned to each event type such as: sending an update to clients, processing a player request for moving, sending and receiving a visibility message from a neighbor server. An average message size as measured in the experimental environment is associated to a client update. The visibility costs are computed as follows. Sending and receiving a visibility message has some fixed CPU cost and incurs a message that is proportional in size to the number of players in the area of interest.

We keep track of the total data in the message exchanges and of the available bandwidth on simulated links between different servers and between servers and their corresponding clients. The state update messages from a server to clients dominate the bandwidth consumption between a server and its clients, and the concrete size of this message depends on the Area of Interest (AOI) of a player and the number of objects in its AOI.

Since game map is continuous in our study, players should be able to see and act in its AOI even though its AOI crosses servers, so neighbor servers should exchange boundary region information in order to satisfy the AOI requirements of boundary players. This communication cost between servers is affected by game architecture, consistency requirements and other design details. If we target at weak consistency, game server can exchange boundary region information in a long period, but it may lead to that boundary players do not able to see some latest map information on their neighbor servers. This kind of delay is dangerous since it may cause players to lose in a fight. To achieve stronger consistency, the information exchanges between servers should catch up with the pace of state update messages. In our simulation, we consider stronger consistency since it brings more real experience for game players.

The size of server exchanging information is mainly decided by the number of boundary players and the distance between them and the boundary, and thus is usually larger than an AOI of a single player. In a LAN game setting, servers are all in LAN environment but clients may be in WAN (for instance, Internet), the network interfaces between inter-server and between

server-client could be different, and thus inter-server communication does not influence the server-clients communication in this case. While in a WAN game setting, servers and clients are all in WAN, and inter-server and server-clients communication share the same bandwidth, so locality should be considered carefully in order to reduce inter-server communication cost. Our simulation shows the results under both LAN and WAN environments.

We report periodical update interval experienced by players. In our simulation, the server periodically sends a state update message to players. We compare the various partitioners through the delays in receiving periodical updates induced by CPU or network resource contention. The *Overloadthreshold*, *Safetythreshold* and *Lightloadthreshold* are 128, 80 and 32 in number of players according to the experiment results of section 5.

6.1.2 Player Mobility Simulation

Because we need traces from thousands of robots in order to run our simulator, to simulate player mobility we prefer to use a modified Random Way Point (RWP) [13] mobility model instead of actual robot traces. We have observed that, since our robots stop eating and fighting when a quest is initiated and just move towards the quest, the resulting player movement pattern is very simple and easy to model with slightly modified traditional mobility models. As in traditional RWP, a node picks a destination at random, proceeds to it following a straight-line trajectory at a random speed and pauses for a random time on arrival. To simulate quests, we have added attraction points to the model, where instead of random destinations a player chooses a given attraction point as its final destination with a high probability. Every player gets a random initial position. The average moving speed of players is 2 meters per second and the size of the whole game map is 400 regions and 10000 by 10000 meters. After a particular quest is over, the players can hover at the quest location for an interval of time, then return to their initial positions.

This model can simulate the flocking behavior of players in MMOG environment. Figure 6.1 shows a distribution of players without flocking behavior, in which black numbers and red

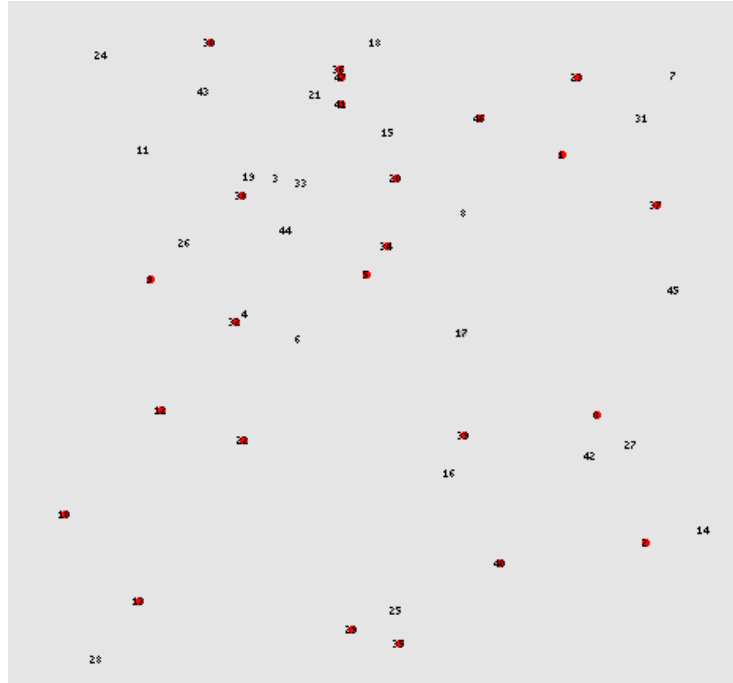


Figure 6.1: Player distribution snapshot without flocking

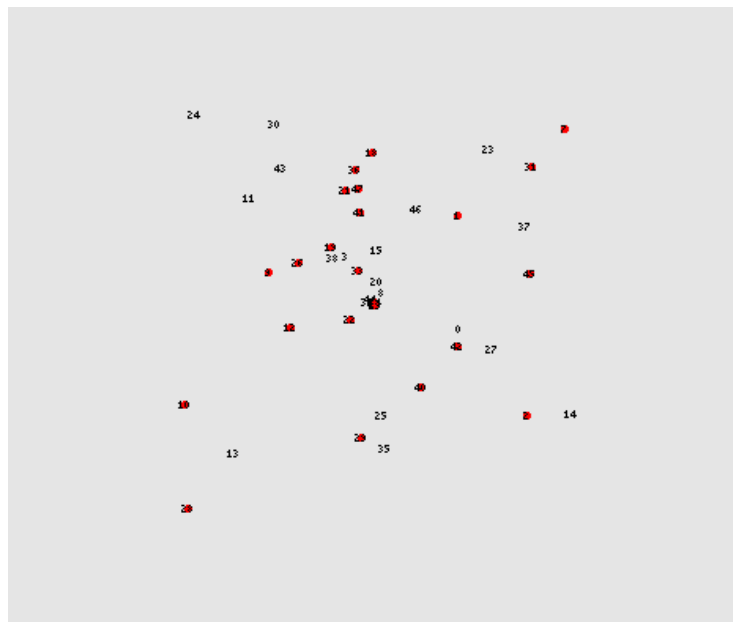


Figure 6.2: Player distribution snapshot during flocking

dots represent one player or several players in the map. Players are nearly evenly distributed in the game map without the disturbing of flocking. Figure 6.2 shows one snapshot when players move toward a center quest, in which more and more players cluster in the center and some corner regions are nearly empty. To be more real, we could set proper probability to allow some players ignore the quest. In addition, we could further simulate complex group behavior of players' movement.

6.2 Simulation Parameters

We use simulation to extrapolate from our experimental results in two different directions: simulating thousands of clients as a state-of-the-art server would support, varying the dynamic partitioning algorithm and varying the environment. Specifically, we simulate a game with 100 servers, 400 regions, and 6000 players. We first study the behavior of various static partitioning algorithms, and compare them with a dynamic load-balancing algorithm. Next we study the main dynamic load-balancing, and its variations (Lightest and Spread). We explore dynamic versus static partitioning of the game world under two different server configurations:

- a LAN-based centralized server with 100 Mbps inter-server and server-client bandwidths corresponding to our experimental platform.
- a WAN distributed server system (e.g., as in a peer-to-peer decentralized server or a set of proxies run by a trusted third-party) which shares 100 Mbps network bandwidth among clients and servers.

In all environments the CPU power and contention is modeled after our experimental environment.

In terms of player movement pattern, we pick a hotspot location in our game world and let all players converge towards it using our mobility model (see section 6.1.2). We make the players move toward the hotspot for half of total simulation time, then players spread out again. We have varied the time that the players stay in the hotspot area and the hotspot location across

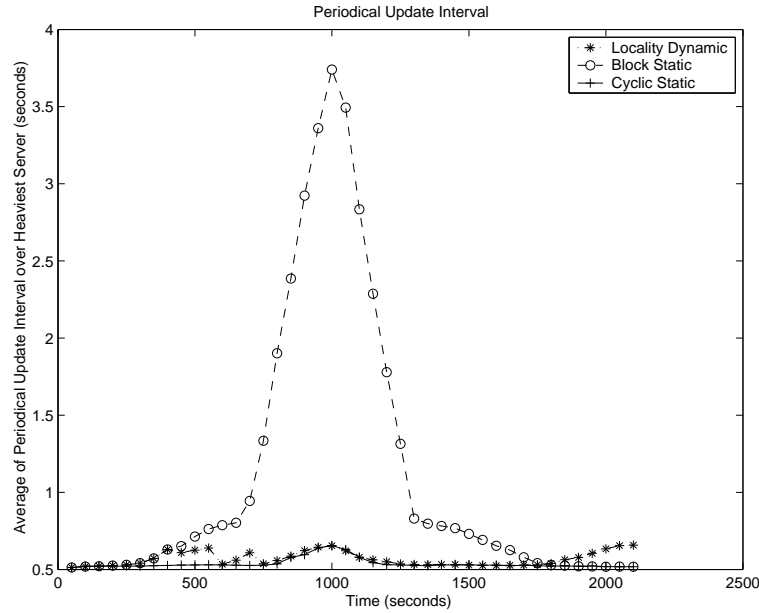


Figure 6.3: Comparison of Block Static Partitioning and Cyclic Static Partitioning for centralized LAN-based server simulated according to the SimMud measurements taken in our environment. Center hotspot. Simulation parameters are 100 servers, 400 regions, 6000 players and 1000 seconds hotspot duration. The network to clients is the bottleneck during the hotspot.

experiments, and we have determined that our results are insensitive to these parameters.

6.3 Results for Static Partitioning

we show a comparison of two typical static partitioning algorithms (Block, Cyclic) and our main locality dynamic load balancing algorithms by varying the position of quest and environments. Block static partitioning divides the whole map into multiple blocks with each block containing several continuous regions of the map, and then assigns one such block to one server. Cyclic partitioning traverses all regions in a predefined order (nearly continuous in a map), and assigns a region to some server in a round robin way.

For the view of load distribution, cyclic partition is good since it tries to distribute the region containing hotspot and its neighboring regions on different servers. But block partitioning

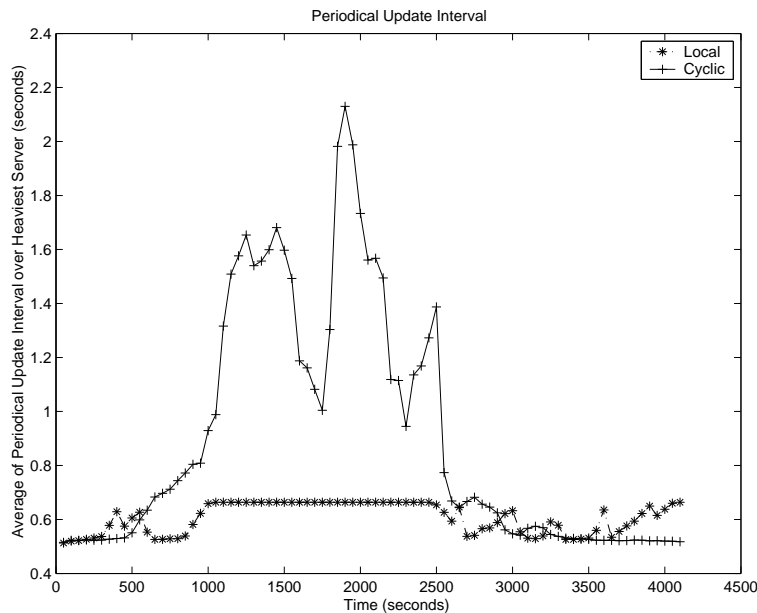


Figure 6.4: Comparison of Cyclic Static Partitioning and Locality-Aware Dynamic Partitioning for centralized LAN-based server simulated according to the SimMud measurements taken in our environment. Corner hotspot. Simulation parameters are 100 servers, 400 regions, 6000 players and 2000 seconds hotspot duration. The network to clients is the bottleneck during the hotspot.

may lead to some server to be overloaded for a long time if the hotspot region and some of its neighbor regions are happen to be assigned to the same server. The comparison of two static partitioning are shown in Figure 6.3 by putting the hotspot in the center of the game map. Although cyclic partitioning work well if the overloaded regions are assigned on different servers, it could work poorly in flocking if several overloaded regions are unfortunately assigned to one server caused by multiple hotspots or a certain single hotspot. To show this inconsistent behavior we change the hotspot location from center in Figure 6.3 to corner in Figure 6.4. We observe that cyclic partitioning behaves well in a center hotspot setting, while works poorly in a corner hotspot setting.

Cyclic partitioning does not take locality into consideration, in many cases, a server containing k regions has k strong connected components in its local map, and thus have much

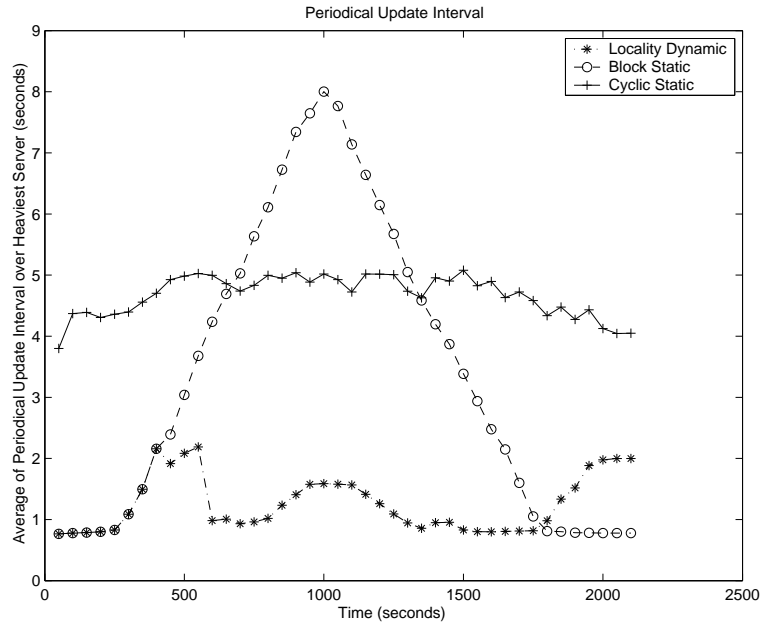


Figure 6.5: Comparison of Block Static Partitioning and Cyclic Static Partitioning for centralized WAN-based server simulated according to the SimMud measurements taken in our environment. Center hotspot. Simulation parameters are 100 servers, 400 regions, 6000 players and 1000 seconds hotspot duration. The network to clients is the bottleneck during the hotspot.

more inter-server communication than Block partitioning. On the other hand, blocking partitioning takes some advantage of locality since each server only contains one strong connected components, and has minimum number of neighbor servers. So blocking partitioning behaves better in WAN setting than cyclic partitioning without the disturbing of flocking. As shown in Figure 6.5, before flocking happens, cyclic partitioning works much worse than block partitioning; after flocking occurs, server-player network bandwidth gradually becomes bottleneck so that block partitioning works worse than cyclic partitioning when more and more players cluster around the hotspot.

Locality aware dynamic load balancing algorithm generally works well. For the cases where a single server capacity exceeds the *overloadedthreshold* in terms of either CPU or server to player bandwidth limitations, Locality algorithm will quickly shed the load of overloaded servers. In the case of limited inter-server bandwidth (occurring in the WAN setting), Locality behaves similar as blocking partitioning when no flocking and better than static partitioning during the process that players move towards the hotspot and finally spread out. The WAN environment is complex because the bottleneck may alternate between an inter-server bandwidth bottleneck and a server-clients bandwidth bottleneck.

Overall, we observe that the static partitioning algorithms have inconsistent behavior across environments and hotspot locations and perform poorly in some experiment. Dynamic load balancing algorithm generally outperforms a single static partitioning.

6.4 Results for Dynamic Load Balancing Algorithms

In figures 6.6 and 6.7 we show a comparison of the four dynamic partitioning algorithms, the main optimized Locality-aware algorithm (Locality), the Locality-aware algorithm without optimization (No Opt Locality), dynamic partitioning that sheds load to lightest loaded servers (Lightest), Spread dynamic partitioning (Spread) and our baseline block static partitioning algorithm in the LAN and WAN distributed environments, respectively.

Both figures show the average of update interval of players of the least responsive (usually most loaded) server. Overall, from both figures we see that congestion causes severe degradation in the average update interval to clients in the static block algorithm. Furthermore, the dynamic partitioning algorithms perform better overall compared to Static Block Partitioning for both platforms.

In the LAN environment (Figure 6.6), all dynamic partitioning algorithms work well maintaining the average update latency below 1 second. Due to the low cost penalty of all types of inter-server communication in our LAN cluster environment such as player hand-off, region migration and inter-server communication for maintenance of the area of interest, all dynamic partitioning algorithms perform roughly the same in this environment.

In contrast, in the WAN environment (Figure 6.7), only the Locality-Aware algorithm manages to maintain an acceptable average update interval after triggering adaptation. Because other algorithms tend to decluster regions, hence increase inter-server communication through their adaptations, the respective average update intervals of Spread and Lightest dynamic partitioning algorithms remain relatively high even after the hot spot disappears since they break locality during load shedding.

The Locality-Aware algorithm performance is very similar to Lightest in the initial phases of hotspot creation because all players are close to the hotspot location and neighbor servers are over the safety threshold themselves. Thus, both algorithms shed to remote servers. The Locality-Aware algorithm outperforms Lightest during the second part of the simulation when the inter-server communication begins to penalize algorithms that have not maintained region clustering. The locality algorithm without optimization (No Opt Locality) behaves between the Lightest and the optimized Locality-Aware algorithm since optimization one requires strict locality.

In order to differentiate the inter-server communication costs further, Table 6.1 shows a comparison of all our algorithms in terms of several metrics: total number of player hand-offs for the whole simulation, total number of region migrations and total number of region clusters

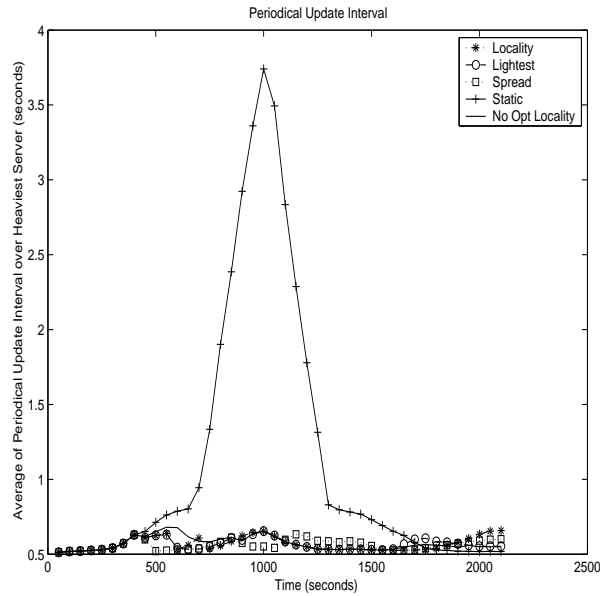


Figure 6.6: Comparison of Dynamic Load Partitioning for centralized LAN-based server simulated according to the SimMud measurements taken in our environment. Center hotspot. Simulation parameters are 100 servers, 400 regions, 6000 players and 1000 seconds hotspot duration. The network to clients is the bottleneck during the hotspot.

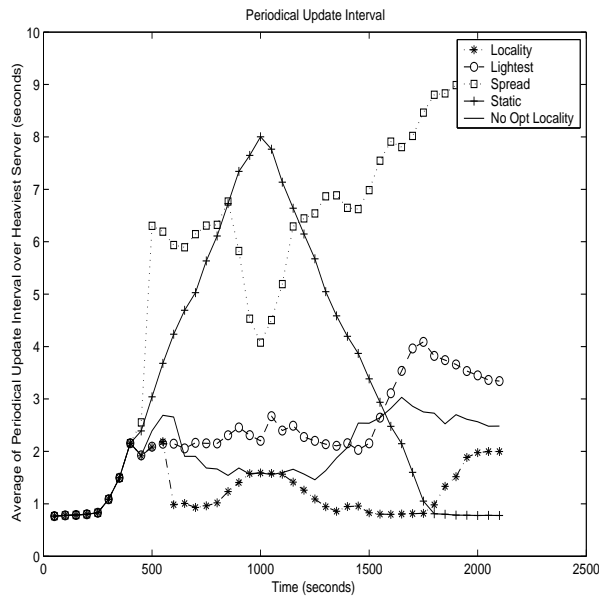


Figure 6.7: Comparison of Dynamic Load Partitioning for WAN server simulated according to the SimMud measurements taken in our environment. Center hotspot. Simulation parameters are 100 servers, 400 regions, 6000 players and 1000 seconds hotspot duration. The network to clients is the bottleneck during the hotspot.

| Metric | Loc-Aware | No Opt Loc | Lightest | Spread | Static |
|------------------|------------------|-------------------|-----------------|---------------|---------------|
| Player hand-offs | 46544 | 48676 | 49046 | 61576 | 44604 |
| Region clusters | 100 | 108 | 110 | 399 | 100 |
| Reg. Mig. Cost | 34 | 12 | 16 | 23268 | 0 |

Table 6.1: Comparison of total number of player hand-offs, total number of region migrations and the total number of region clusters at the end of simulation for all algorithms.

at the end of simulation for all algorithms. We see that the locality-aware algorithm maintains the best region-cluster locality by keeping the number of strongly connected region clusters to the same value as when simulation starts (i.e., 100 region clusters). The total number of player hand-off correlates to the number of strongly connected region cluster at any given point in time since player-hand-offs occur across partition boundaries. Locality maintains total number of player hand-offs close to the Static algorithm which is the ideal. The slight degradation in Locality-Aware versus Static occurs due to the fact that borders become more irregular after region migrations, hence slightly more hand-offs may occur even if the total number of region-clusters remains the same (100). The Lightest dynamic region migration algorithm maintains overall region-clustering, and player hand-off latencies very close to the main Locality-aware algorithm. However, the most locality disruption occurs on the overloaded server that sheds load or load receiving servers. Thus, the performance of the server with the highest update interval latency (shown in Figure 6.7) will be affected the most by the region declustering and the resulting inter-server communication.

The total number of region migrations in Lightest is less than in Locality-aware as expected because that optimized Locality requires the lightest node to first shed their regions out, and then accept new regions. Hence No Opt Locality also has the advantage of a smaller number of region migration times versus Locality. There is a tradeoff between reducing update interval and migration times in Locality algorithm.

In contrast to Lightest and Locality, we see that Spread does much worse in terms of all

metrics because it completely disregards locality maintenance and does not try to minimize the number of region migrations. Hence, the extra inter-server communication seriously impacts its performance in the wide-area environment.

Chapter 7

Related Work

7.1 General Load Balancing Algorithms

Since the load balancing problem for minimizing the makespan is NP-hard [15], many heuristic algorithms have been developed. **The List Scheduling (LS)** algorithm maintains a list of jobs and assigns the next job in this list to the machine with minimum load [9]. LS can achieve a $(2 - 1/m)$ -approximation where m is the number of total machines [9]. **Longest Processing Time (LPT)** [10] orders the jobs in the order of decreasing processing time (or load); whenever a machine is freed (or has minimum load), the largest job ready at the time will be assigned to this machine. In contrast with LS, LPT requires that the processing time of jobs be known before scheduling.

The simplified bin-packing algorithm in section 3.4.1 that we use for Spread is a LS algorithm. The disadvantages of general load balancing algorithms are that they only consider the aspect of balancing load, and ignore other important optimization goals and constraints specific to the concrete applications. For instance, in our game problem, we need to minimize the inter-server communication, which general load balancing algorithms do not take into consideration. In addition, LS and LPT do not consider job relocation or migration cost during dynamic scheduling of processes.

Recently, Aggarwal et al. [1] addressed the dynamic load balancing problem with migration cost. They formulate the problem as follows: Given an assignment of the n jobs to m processors, and a positive integer k , relocate no more than k jobs so as to minimize the maximum load on a processor. More generally, we are given a cost function c_i which is the cost of relocating job i , and the constraint is that the total relocation cost be bounded by a specified budget B . The authors first prove that a simple greedy algorithm can achieve 2-approximation with a running time of $O(n \log n)$. Then, a more complicated algorithm is presented to achieve 1.5-approximation and obtain a running time of $O(n \log n)$. This algorithm requires global knowledge about the load of all tasks which is usually not available in a distributed environment. Moreover, they do not give a method to estimate the parameter k and budget B . Since it is hard to set a budget for region migrations in our problem, so we do not implement this algorithm in our experiments.

7.2 Load Balancing in Parallel Applications

Many applications are load balancing critical, especially for scientific parallel applications, such as molecular dynamics simulations, optimal layout of VLSI chips and parallel databases. This sort of applications usually have multiple objectives for load balancing. They not only require load to be evenly distributed, but also want to minimize the inter-processor communication. They could be formulated as a graph partitioning problem which tries to achieve load balance and to minimize the inter-processor communication simultaneously. Since the data or load distributions are quite different for distinct applications, we cannot use their algorithms directly.

Parallel databases have the problem of data skew, where data may be unequally distributed across the partitions. An online balancing algorithm is presented by Ganesan et al. [7] for range-partitioned databases in Peer-to-Peer systems. In this problem, every load movement is required to keep the range continuous for any specific partition. Instead of minimizing

makespan, their objective is to minimize the number of tuples moved to achieve a desired constant imbalance ratio.

The algorithm first classifies load into different levels. Tuple movements are limited in two operations: neighbor adjustment and reorder. Adjustment means neighbor nodes change their boundary by transferring data from one node to the other. Reorder means an empty node changes its position and splits the range $[R_j, R_j + 1]$ managed by a node N_j . The load threshold has a form of $T_i = c\delta^i$. Adjustments are triggered when a node's load increases beyond a threshold. In this case, the node first attempts to do neighbor adjustment, and perform reorder if both neighbors have high load. Their threshold algorithm guarantees a constant imbalance ratio σ with low amortized cost per tuple insert and delete. An optimization on this base algorithm based on Fibonacci numbers guarantees an imbalance ratio of $\sigma \simeq 4.24$ in this system. In this application, data are continuously distributed in one dimension and the load unit is a tuple. In contrast, in our problem, the game map is in 2-dimensions or more, the load of a region corresponds to the number of players in this region and could change dramatically.

Load Balancing algorithms are also used in dynamically structured P2P Systems [8] because their use of Distributed Hash Table may result in some nodes having up to $\log(N)$ times more load than the average. This paper solves the problem of nonuniform distribution of objects by dynamically moving virtual servers from heavily loaded physical servers to lightly loaded ones. In their algorithm, a virtual server joining a destination physical server corresponds to a leave of some other virtual server of a heavy loaded physical server. The algorithm ensures that the new virtual server is the successor of the old one so that the objects of the old virtual server can be moved to the new virtual server. The load management problem for game servers in a structured P2P environment will be explored in our future work.

7.3 Load Balancing in Games

MMOGs have different workload characteristics compared with previous applications. Flocking patterns cause the load imbalance of regions. The intensely increasing load of some region may be due to having a quest nearby or having a group of players passing the region toward a remote quest. Moreover, there may be multiple quests in a game, hence, the players' behaviors is hard to predict.

Companies such as Butterfly.net [3] and TerraZona [26] develop middleware that provides cluster support for MMOGs. However, these systems provide only static partitioning and newer, more dynamic games like Sims Online [6] and Tabula Rasa [18] can not be effectively handled by them [18]. Each server can support only one region at a time. Updates do not propagate across regions.

Lui and Chan [16] reduce load balancing in distributed virtual environment systems to a graph partition algorithm, and assume global knowledge of computation cost and pairwise communication cost between avatars. They try to find a global partition with minimum total cost. However they do not show how to measure the computation and communication cost in a real game environment. We argue that the centralized partitioning they use can be too expensive in computation and communication.

CittaTron [11] is a multiple server game with load adaptation mechanisms. Their algorithm dynamically moves rows of boundary regions to neighboring servers. Their idea of incremental load shedding is similar to the neighbor shedding part of our Locality-aware algorithm, but they do not explicitly consider the inter-server communication cost resulting from lack of locality.

In our current simulation and implementation, the game map is 2-dimension. In more complex cases, the game map can be 3-dimension, or even higher dimensions. To reduce the running time of graph partitioning in high dimensions, we can use many space-filling curves [2], such as Hilbert curves, for mapping the multidimensional map into our two dimension space while preserving some proximity information present in the multidimensional space. We plan to explore this direction further in our future work.

Chapter 8

Conclusion and Future Work

8.1 Conclusions

In this thesis we have investigated scaling for a large-scale networked game server. We introduce dynamic load management techniques that allow the networked server to support large shared game worlds with a high number of interacting players. We show that dynamic load management significantly outperforms some commonly used static partitioning schemes which are not appropriate for games with very dynamic player mobility. A locality-based dynamic load management scheme alleviates the severe bottlenecks that appear due to game hotspots and shows fast adaptation times. Our results show that the locality-aware dynamic load balancer preserving spatial locality improves performance by up to a factor of 4 compared to global algorithms that do not consider spatial locality and by up to a factor of 6 compared to static partitioning.

8.2 Future Work

We plan to deploy SimMud in a very large scale real testbed. We hope SimMud will attract thousands of players to join in and has hundreds of game servers. These game servers will be widely distributed in different geographical places, and thus we can test and verify our

algorithm in WAN or Internet environments. Although SimMud now has basic features of MMOGs, we will further study real features of real MMORPGs ,and improve SimMud to better emulate the behavior of real game servers. Our current algorithm does not make use of game knowledge and does not consider the movement trend of players. If we are able to trace the trails of players, we could recognize the movement trend of players and thus make migration decisions more intelligently.

From the architecture aspect, we are interested in exploring load balance issues in various peer-to-peer game architectures, structured versus non-structured. In a P2P environment, a game server is not dedicated any more and could just be a regular player. How to ensure security, how to build a P2P game overlay network, how to maintain load balance should be carefully considered together.

Bibliography

- [1] Gagan Aggarwal, Rajeev Motwani, and An Zhu. The load rebalancing problem. In *The Fifteenth Annual ACM symposium on Parallel algorithms and architectures*, pages 258 – 265, 2003.
- [2] S. Aluru and F. Sevilgen. Parallel domain decomposition and load balancing using space-filling curves. In *In Proc. 4th International Conference on High-Performance Computing*, pages 230–235, 1997.
- [3] Butterfly.net, Inc. The butterfly grid: A distributed platform for online games, 2003. <http://www.butterfly.net/platform/>.
- [4] Jin Chen, Baohua Wu, Margaret Delap, Bjorn Knutsson, Honghui Lu, and Cristiana Amza. Locality aware dynamic load management for massively multiplayer games. Technical Report TR-12-01-04, University of Toronto, December 2004.
- [5] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet. *IEEE Networks magazine*, 13(4), July/August 1999.
- [6] Electronic Arts Inc. Sims online, 2003. <http://www.eagames.com/official/thesims/thesimsonline/us/nai/index.jsp>.
- [7] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of 30th International Conference on Very Large Data Bases (VLDB)*, Aug 2004.

- [8] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in dynamic structured p2p systems. In *INFOCOM '04*, Hong Kong, China, March 2004.
- [9] R.L. Graham. Bounds for certain multiprocessing anomalies. In *Bell System Tech. J.*, 45:1563-1581, 1966.
- [10] R.L. Graham. Bounds on multiprocessing anomalies. In *SIAM J. Applied Math.*, 17:263-269, 1969.
- [11] Masato Hori, Takeki Iseri, Kazutoshi Fujikawa, Shinji Shimojo, and Hideo Miyahara. Cittatron: a multiple-server networked game with load adjustment mechanisms on the internet. In *SCS Euromedia Conference*, pages 253–260, 2001.
- [12] Id Software. Quake, 2004. <http://www.idsoftware.com/games/quake/quake/>.
- [13] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages pages 153–181. Kluwer Academic Publishers, 1996.
- [14] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM '04*, Hong Kong, China, March 2004.
- [15] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, 46(3):259–271, 1990.
- [16] John C. S. Lui and M. F. Chan. An efficient partitioning algorithm for distributed virtual environment systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(3), March 2002.
- [17] Mike Mika and Chris Charla. Simple, cheap pathfinding. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 155–160. Charles River Media, Inc., 2002.

- [18] Mitch Ferguson and Michael Ballbach. Product review: Massively multiplayer online game middleware, January 2003. http://www.gamasutra.com/features/20030115/ferguson_01.htm.
- [19] Katherine L. Morse. Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27, University of California, Irvine, 1996.
- [20] NC Soft. Lineage, 2002. <http://www.lineage.com/>.
- [21] Paul Bettner and Mark Terrano. GDC 2001: 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond, March 2001. http://www.gamasutra.com/features/20010322/terrano_01.htm.
- [22] D. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. In *Mathematical Programming*, 62:461–474, 1993.
- [23] David B. Shmoys. *Approximation algorithms for NP-hard problems*, chapter Cut problems and their application to divide-and-conquer, pages 192–235. PWS Publishing Co., 1997.
- [24] Sony Computer Entertainment Inc. Everquest online adventures, 2002. everquestonlineadventures.station.sony.com/.
- [25] Sony Online Entertainment Inc. Planetside, 2004. <http://planetside.station.sony.com/>.
- [26] Zona Inc. Terrazona: Zona application frame work white paper, 2002. <http://www.zona.net/whitepaper/Zonawhitepaper.pdf>.