Locality Aware Dynamic Load Management for Massively Multiplayer Games

Jin Chen
Department of Computer Science
University of Toronto, Canada
jinchen@cs.toronto.edu

Baohua Wu
Department of Computer and Information Science
University of Pennsylvania, USA
baohua@cis.upenn.edu

Margaret Delap
Department of Computer and Information Science
University of Pennsylvania, USA
delap@cis.upenn.edu

Björn Knutsson
Department of Computer and Information Science
University of Pennsylvania, USA
bjornk@cis.upenn.edu

Honghui Lu
Department of Computer and Information Science
University of Pennsylvania, USA
hhl@cis.upenn.edu

Cristiana Amza
Department of Electrical and Computer Engineering
University of Toronto, Canada
amza@eecg.toronto.edu

ABSTRACT

Most massively multiplayer game servers employ static partitioning of their game world into distinct mini-worlds that are hosted on separate servers. This limits cross-server interactions between players, and exposes the division of the world to players. We have designed and implemented an architecture in which the partitioning of game regions across servers is transparent to players and interactions are not limited to objects in a single region or server. This allows a finer grain partitioning, which combined with a dynamic load management algorithm enables us to better handle transient crowding by adaptively dispersing or aggregating regions from servers in response to quality of service violations.

Our load balancing algorithm is aware of the spatial locality in the virtual game world. Based on localized information, the algorithm 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 6 compared to a global algorithm that does not consider spatial locality and by up to a factor of 8 compared to static partitioning.

Categories and Subject Descriptors

D.0 [**Software**]: GENERAL

General Terms

Algorithms, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.
Copyright 2005 ACM 1-59593-080-9/05/0006 ...$5.00.

Keywords

adaptive, distributed, load balancing, locality aware, massively multiplayer games

1. INTRODUCTION

The popularity of Massively Multiplayer Online Games (MMOGs) like Lineage [13] and Everquest [14] is on the rise with millions of registered players, and hundreds of thousands of concurrent players. Current state-of-the-art servers, such as, Butterfly.net [3] and TerraZona [15] 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. Newer games, such as, Sims Online and Tabula Rasa [11] feature more dynamic landscapes making static load prediction harder, hence static partitioning infeasible.

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. Most importantly, our flexible server architecture allows us to address server bottlenecks caused by registered MMOG patterns, such as, player flocking.

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. Hence, no static game partitioning algorithm can effectively address this problem.

Dynamic load management through adaptive region to server remapping is an obvious solution to the load management problem. However, any dynamic remapping algorithm must evaluate the potential trade-offs between the two conflicting goals of i) *Balancing the server load in terms of numbers of players* by splitting existing game world partitions across *more* servers and ii) *Decreasing inter-server communication* by maintaining the locality of adjacent regions or aggregating adjacent regions into large partitions to be assigned to *fewer* servers.

In more detail, game hot-spots degrade the quality of service (i.e., the response time) for server(s) hosting the interesting regions to unacceptable levels. On one hand, this naturally leads to remapping some of the regions from the overloaded servers onto underloaded servers. On the other hand, the load rebalancing process itself may result in an increase in the number of partition boundaries, hence possible disruption in region locality. Locality disruption in its turn increases inter-server communication for cross-partition visibility and inter-server player hand-offs as players move between partitions. Finally, since dynamic remapping of regions is complex, involving inter-server migration of the region state, the number of region remaps in any dynamic partitioning algorithm should be kept low.

Few dynamic load balancing techniques in massively multiplayer game servers [9] attempt to address the complex load versus inter-server communication trade-offs. Moreover, existing algorithms [1, 9], for solving the NP-complete load balancing problem typically involve all nodes in the system in global heuristic approximations. In contrast to these global algorithms, which are very compute and communication intensive, we use a localized algorithm. An individual server detecting a local bottleneck causing quality of service violation triggers dynamic region repartitioning. Depending on the bottleneck type, the repartitioning goal is either shedding load from an overloaded server or aggregating partitions of relatively underloaded servers to reduce inter-server communication. In either case, our dynamic partitioning algorithm is *locality aware*. Specifically, our solution preserves region locality in each server partition, while involving only a small portion of the network, usually corresponding to the in-game neighbor servers, in the repartitioning process.

In our evaluation, we use a prototype implementation of the SimMud game server [8] 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 and the costs of inter-server communications. We plot the server's response time curve with increasing number of clients. We measure CPU, bandwidth consumption and delays for basic single-server operations, region and player state migration. We use the measured delays to calibrate a configurable simulator in terms of server operation costs and resource contention curves. We then evaluate various versions of dynamic load management with and without spatial locality versus static load balancing, using trace-driven simulations that involve massive player flocking. We explore the behavior of locality aware dynamic load management through simulation in two server configurations: a centralized local area network server (LAN) as in state-of-the-art game servers [3, 15] and a large-scale wide-area distributed server (WAN) [8].

Our results show that the dynamic load balancers outperform static load balancing substantially in both configurations. Our results further show that 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,

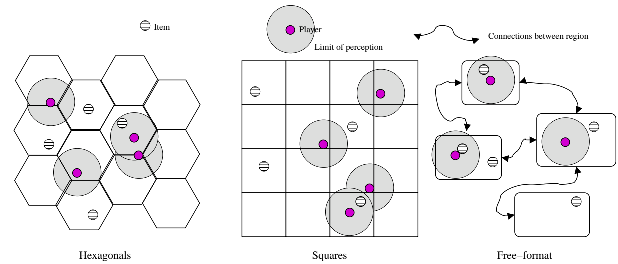

Figure 1: Partitions of the game world.

preserving spatial locality improves performance by up to a factor of 6 compared to a global algorithm that does not consider spatial locality.

The rest of this paper is organized as follows. Section 2 introduces background on game server architectures. Section 3 introduces our dynamic load management solution. We describe our prototype implementation in Section 4, our evaluation methodology in Section 5 and experimental results in Section 6. We investigate the performance of the different static and dynamic partitioners for large numbers of concurrent players through simulation in Section 7. Section 8 discusses related work. Section 9 concludes the paper.

2. GAME SERVER ARCHITECTURE BACKGROUND

The client-server architecture is the predominant paradigm for implementing massively multiplayer online games, in which a centralized server receives server requests, updates the game state and then propagates the new state to clients. 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.

Server clusters are typically used to support thousands of players in strategy or role playing games [3, 15]. Current implementations use static partitioning of the game map, where each partition is assigned to a server. Players switch servers when they move from one region to another. Load balancing is a problem — current games require game designers to carefully plan the load distribution at the design phase.

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 modeled as airports. This works well for games where the world can be abstracted into separate areas and allows limiting the number of players entering any one partition. Where this is not possible, and the world truly is a single contiguous map with strong spatial relationships, “true” partitioning is, however, required. Figure 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 game avatar has limited movement speed and sensing capabilities. In the figure, the area a player avatar can sense i.e., its area of interest, 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* [12] 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. Thus, inter-server communication is needed only between servers hosting neighbor partitions of the game map for communication of state information to ensure cross-partition visibility within their players' area of interest. Furthermore, as players move between server partitions, they incur player hand-off costs between the servers hosting the affected regions to communicate the avatar state including its position in the world, its abilities, health and possessions.

3. LOCALITY AWARE DYNAMIC LOAD MANAGEMENT

Locality Aware dynamic partitioning is a decentralized algorithm that uses a heuristic approach to i) shed load from an overloaded server while keeping locality in mind and ii) aggregate server partitions in normal load for reducing excessive inter-server communication.

Each server monitors its quality of service (QoS) violations in terms of its average update interval to each of its clients through periodic sampling. The server then determines whether the cause of a perceived QoS violation is client load or inter-server communication and triggers either load shedding or aggregation, respectively.

Locality Aware dynamic partitioning uses locality heuristics in two dimensions: communication based on network proximity in the game and region clustering based on adjacency on the game map. The former means that the distributed load balancing algorithm favors localized communication between servers hosting neighboring partitions for both load shedding and aggregation. The latter means that it attempts to keep adjacent regions together on one server.

Each 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. When overloaded with clients, a server favors shedding load to its neighbors rather than to any other node in its locally maintained lightly loaded set. Conversely, partition aggregation corrects any locality disruption caused by load shedding by inter-neighbor partition aggregation when in normal load.

In the following sections, we detail our formal definitions for quality of service and use of load thresholds (i.e., light load, overload and safe load) in section 3.1, our overall algorithm for shedding load by an overloaded server in section 3.2 and our algorithm for partition aggregation in section 3.3.

3.1 Definitions

Our definition of quality of service is in terms of meeting a service level agreement (SLA), in our case a predefined update interval of each server to its clients. An *SLA violation* occurs when a server exceeds the predefined update interval for 90% of its clients. We define the *overload threshold* ($Overload_th$) as the server load in terms of number of clients for which the SLA is violated. We further define the *safe load threshold* ($Safety_th$) the highest server load for which the SLA is still met for all clients. The overload and safe thresholds are determined by off-line measurements using a stand-alone server (i.e., without inter-server communication). If the number of clients on a server exceeds its *overload threshold*, we say that the server is in *overload*.

An SLA violation while in *overload* triggers load shedding from the overloaded server. On the other hand, to avoid system oscillation, we trigger aggregation more conservatively only for non-overloaded servers that violate double the SLA used for load shedding.

The *safe load threshold* corresponds to the target load after load shedding for an overloaded node. Any load under the *light load threshold* is the load of an optimal candidate for receiving the load shed. Once a safe load threshold has been set, the light load 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 safe load threshold.

$$Light_load_th = 2 * Safety_th - Overload_th$$

Each server node maintains a local load graph as an undirected weighted graph $G(V, E, W)$ where V are vertices, E edges, and W weight of vertices. Every server exchanges load graphs and lightly loaded server lists with neighbors and maintains their load graph in addition to its own. In G, each vertex v represents a region. The weight of a vertex v represents the load of the corresponding region. If two regions are adjacent on the game map and are hosted on the same server, their corresponding vertices are connected by an edge e. Thus, a contiguous region cluster hosted on a server forms a *strongly connected component* in the graph representing that server's partition.

We quantify locality preservation for the purpose of dynamic load management in terms of preserving the number of strongly connected components in the overall game map graph.

3.2 Locality Aware Load Shedding

The constraint of load shedding is to achieve the safe load target on the overloaded node without exceeding the safe load threshold on any of the nodes that it sheds load to. At the same time, load shedding pursues two optimization goals: i) Locality is preserved, i.e., the same number of strongly connected components is maintained after load shedding as before and ii) The minimum number of region migrations occur.

When a server's periodic sampling detects an SLA violation and the server determines that it is due to overload as defined in section 3.1, the overloaded server triggers the ShedLoad algorithm shown in figure 2. In the first phase of the load shedding algorithm, the overloaded server, S_i, attempts to shed load to as few neighbors as possible, then sheds any remaining load remotely as described in the next sections.

3.2.1 Load Shedding to Neighbors

Each candidate neighbor's load should be below the safe threshold load 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 attempting to give each a contiguous region cluster starting with the respective neighbor's boundary regions using a heuristic graph partitioning algorithm (see section 3.2.3). If the safe load level has not been achieved after shedding to neighbors, the remaining excess load is shed to the lightest loaded node known.

3.2.2 Load Shedding to Lightly Loaded Servers

If shedding to remote servers becomes necessary, the algorithm first checks the lightly loaded server set maintained locally. The overloaded server contacts server S_j, the lowest loaded server from its lightly loaded list as the next candidate to receive the load shed. Since the information kept in the lightly loaded server set may not

```
ShedLoad(Si, targetLoad)
{
  //Attempt to shed to neighbors
  leftLoad = ShedtoNeighbors(Si, targetLoad);

  while (leftLoad <= targetLoad) {

    //Find a new lightly loaded candidate
    Sk = SeekUnvisitedLightestServer();

    if (not found) return; //No more resources

    //shed to lightest loaded node
    HeuristicGraphPartition(Si, Sk, targetLoad);
  }
}


ShedtoNeighbors(Si, targetLoad);
{
  while (Si leftLoad > targetLoad and
         neighbor unvisited) {
      Sj = GetLightestLoadedNeighbor(Si);
      HeuristicGraphPartition(Si, Sj, targetLoad);
  }
}
```

Figure 2: Pseudo code for Locality Aware Load Shedding.

be accurate, S_j may reject the request because it is not in lightly loaded status anymore. If the overloaded server cannot get a candidate from its lightly loaded server set, it floods the network with a SeekLightlyLoadedServer request message searching a lightly loaded server. These messages are propagated among neighbor pairs and carry the identity of the overloaded server. If an underloaded server cannot be found, this means that all resources are highly utilized and load shedding is not possible.

When 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 unlikely 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 lightest loaded candidate for load shedding.

3.2.3 *Heuristic Graph Partitioning Algorithm*

We use the following graph partitioning heuristic algorithm to determine a load shedding schedule given our constraints and locality-based optimization goals. If any acceptable schedule is found for shedding load to either a neighbor or remote node as the case may be, the corresponding scheduled region migrations occur. Otherwise, the load shedding returns an error, by signaling that the load remaining on the node is unchanged. The algorithm proceeds with the next option as shown in the ShedLoad pseudocode.

Our graph partition problem is hence to divide a graph into two or more strongly connected components. The sum of the region weights for each component to be shed should be as close as possible to the target sum of weights to shed for reaching the safe load threshold. In the case of shedding load to a neighbor, S_j, we set the root of our search as one of the regions on the *boundary* with that neighbor. In case of shedding to a remote server, S_r, a random boundary region is chosen. We then use breadth first search (BFS) to explore connected regions by following edges starting with the

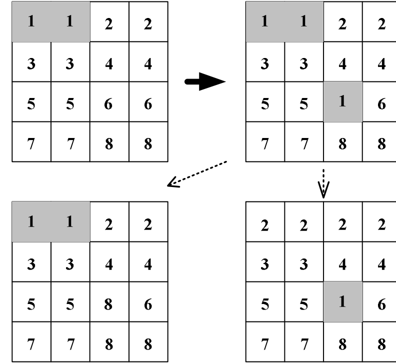

Figure 3: Remote load shedding and subsequent possibilities for merging to correct locality disruption.

root region to obtain a strongly connected region cluster to shed. While the target load is not met on the local server, we continue adding to the cluster to be shed by traversing regions in BFS order. The constraint is to keep the total load of S_j (or S_r) including the weights of the new tentative regions less than the safe load threshold.

3.3 Locality Aware Aggregation

In this section, we introduce a partition aggregation algorithm, orthogonal to load shedding, that is triggered when quality of service violation for a particular server is caused by excessive inter-server communication instead of high client load.

We first give an intuition for our algorithm through a motivating scenario where locality disruption causes excessive inter-server communication in section 3.3.1. We then proceed to describe locality aware aggregation more formally, as a heuristic graph merge algorithm in section 3.3.2.

3.3.1 *Motivating Example*

The example in Figure 3 shows the partition of regions to servers before (top left) and after (top right) a remote load shed, respectively. Initially, blocks of two regions are assigned statically to servers. Next, assume that a hot-spot occurs within server 6's initial partition. As a result, server 6 needs to shed load, but let's say neighbor servers 4,5 and 8 are overloaded themselves with players moving towards the hot-spot location. Hence, server 6 is forced to shed one region to remote server 1.

As we can see, remote load shedding implies locality disruption. This is reflected in an overall increase in the number of adjacent region clusters co-located on the same server. In the initial configuration, each server partition contains exactly one cluster of regions with strong locality (strongly connected component). After the remote load shed, the number of distinct strongly connected components in server 1's partition has increased with the new single region component. In contrast, shedding the region to one of server 6's neighbors would have kept the overall number of strongly connected components unchanged. Shedding load rapidly is, however, the immediate goal under severe hot-spot overload for server 6 and shedding to a remote server is the only option.

The net effect of the observed locality disruption is potentially high inter-server communication. Before receiving the load shed, server 1 has inter-server boundaries only with servers 2 and 3, while afterwards server 1 has several inter-server boundaries with

```
MergewithNeighbors(Si)
{
  while(unvisited neighbor Ni){
    if(Ni boundaries < Si boundaries and
       Ni load < Safe threshold)
      HeuristicMerge(Si, Ni);
  }
}


HeuristicGraphMerge(Si, Ni)
{
  while (unvisited boundary region R){
    //Estimate merging R into Ni's partition
    if(Ni new load > Safe threshold or
       Ni new boundaries > Si new boundaries)
      Discard partial schedule;
    else
      Schedule region migration to Ni;
  }
}
```

Figure 4: Pseudo code for Locality Aware Aggregation.

servers 2,3,4,5,6 and 8. Over time, under random, dispersed player movement after the hot-spot is over, server 1 may experience excessive inter-server bandwidth consumption in the new configuration. Quality of service to clients may thus degrade on server 1 due to high volume inter-server visibility information exchange and/or frequent player hand-offs back-and-forth across its several boundaries. Although the potential performance impact of increased inter-server communication might seem second order compared to the one induced by hot-spot server overload, the problem may persist for a longer time than the hot-spot itself.

Our partition aggregation algorithm corrects such isolated locality disruptions. Server 1 detects that its QoS violation is not induced by overload in terms of its average number of serviced clients, but due to excessive inter-server communication and triggers locality aware aggregation. The intuition behind our aggregation algorithm is that server 1 might be able to merge one of its region clusters into a neighbor server's partition under normal load conditions. For example, server 1 may be able to merge its single region component with the partition of server 8, when server 8's load subsides (bottom left example). Alternatively, server 1 may be able to merge its other two-region cluster into server 3's partition and/or server 2's partition, as in the bottom right example.

3.3.2 Heuristic Graph Merge Algorithm

More formally, Figure 4 shows how our MergewithNeighbors algorithm attempts to create an acceptable schedule of merging the regions of the problem server, S_i, into the partitions of its neighbors. As in the load shedding algorithm presented before, for each neighbor, HeuristicGraphMerge is called in turn for all neighbors. As before, HeuristicGraphMerge traverses regions in BFS order starting from the boundary with the corresponding neighbor in order to create the merge schedule for regions. The optimization goal and constraints are, however, different in this case. The MergewithNeighbors algorithm attempts to optimize the number of inter-server boundaries for the server S_i (S_i_b). The constraint is to maintain the load under the safe threshold and the number of inter-server boundaries under the current S_i_b for each neighbor server involved.

While the ideal merge removes an isolated region cluster from S_i's partition, any partial partition merge that shrinks S_i's partition is considered valid if it decreases S_i's inter-server boundaries, S_i_b. Note that our constraint for safe neighbor load while merging avoids oscillations in the algorithm due to the potentially contradictory goals of shedding load versus decreasing inter-server communication of two different neighbor servers.

3.4 Other Static and Dynamic Partitioning Algorithms Used for Comparison

In this section, we introduce other dynamic load balancing algorithms for comparison with the main Locality Aware dynamic partitioning algorithm. We also introduce a baseline Static partitioning algorithm. 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 remapping 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. 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.

This algorithm, however, 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 smallest difference between the lowest and highest loaded server in the system. 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 adapted to include 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.

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

Lightest is a dynamic load balancing algorithm that attempts to optimize the cost of remapping by prioritizing shedding load to a single server instead of several servers. The algorithm does not take network proximity in the game (i.e., to neighbors) into account. Furthermore, 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_th. Note that our definition of Light_load_th is such that a single such node should be able to accommodate a sufficient load shed from an overloaded node. While this is true in most cases, depending on the actual distribution of players to regions, if some regions are more overloaded than others, a careful load shedding schedule should be constructed to maximize the load to be shed. 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 for 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 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 without exceeding the safe load 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 algorithms for static partitioning of regions to servers are possible: static block partitioning, row-based or column-based static partitioning and cyclic partitioning as well as combinations of these. We have previously shown [5] that no single static partitioning algorithm performs well for all types of scenarios in terms of hot-spot location and server environment. In our previous study, static partitioning algorithms had inconsistent behavior across environments and hot-spot locations and performed extremely poorly in at least one experiment. In this paper, we use static block partitioning as a sufficiently general static load partitioning algorithm for a baseline comparison with our dynamic partitioning algorithms.

3.5 Discussion

None of the heuristics used in our dynamic partitioning algorithms is hard-wired for a particular environment. The same algorithm applies to 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). For instance, individual load thresholds for each server node in heterogeneous server environments could be either measured or approximated by multiplying normalized client load measurements taken on a standard-resource server by the respective server capacity ratios. 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 region 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 Lightest. Shedding to several neighbors may imply a higher remap cost, but may be the best in terms of region clustering type locality and communication cost during load shedding. Shedding to the lightest loaded node as in the Lightest algorithm may involve on the downside some region locality loss and communication penalties such as a search if the local node currently does not know of any such node.

All three algorithms are fast. Lightest and Locality Aware are running in sub-linear time in terms of their number of server participants and the number of regions involved. Spread is the highest complexity algorithm, running in $O(rlogn)$ time, where r is the total number of regions and n is the total number of servers.

4. IMPLEMENTATION

We implement our load balancing algorithms within the context of SimMud [8], a simple game developed at University of Pennsylvania. 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

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.

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

Player objects are the most complex game object in SimMud. Players are able to move, eat, and fight other players. SimMud supports both robotic, AI-controlled players, which we call *robots*, and user-controlled players operated using the keyboard. Our robots use a variant of a simple path finding algorithm [10] to move around the map and avoid roadblocks. Robots are programmed to explore, seek out food, fight weaker opponents and flee from stronger. Each player object has a health attribute which can be increased by eating or decreased as the result of a fight.

Quests are a mechanism for causing robots to flock to one area of the map, thus loading the server hosting the corresponding regions. 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, this overrides all other robot priorities, and the robot moves single-mindedly to the quest goal. Quests are useful in that they allow simulation of the flocking behavior that occurs in real games.

4.2 Server Actions

Player eating or fighting involves sending a request to the region's server, which decreases the quantity of the food and updates the player's health, respectively. 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 to clients in region status updates.

4.3 Multiple Server 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.

The multiple server SimMud, 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

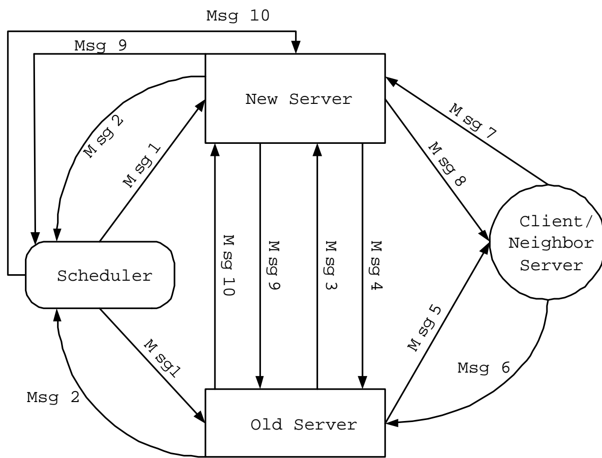

Figure 5: Multiple server SimMud architecture.

scheduler in order to accommodate centralized scheduling algorithms, such as, Spread. Each server keeps track of its own load and reports it to the scheduler periodically. The scheduler makes cross-server region migration decisions. 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.

4.4 Player Migration

Quests or exploration may lead a robot or a human player to try to move outside its server partition. 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 servers 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 on the new server. 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 player handoff across servers.

4.5 Region Migration

Figure 5 illustrates the message sequence of a scheduler-initiated region migration for load management purposes. The messages are numbered by their time line in the process. The odd numbered messages are requests, and the corresponding even numbered messages are replies. Details of the region remap follow:

- Msg1/2: The scheduler sends a *Remap Decision* message to both the original server and the new server of a region to initiate the region migration.
- Msg3/4: The original server sends a *Remap Request* to the new server that contains the region's status and saved incoming messages.
- Msg5/6: Upon a successful remap request, the original server multicasts a *Remap Notify* to its clients and neighboring servers to notify them of the new server's IP address.
- Msg7/8: Upon receiving the new server's address, both clients and neighbor servers contact the new server by sending a *Player Remap Request* or a *Neighbor Remap Request*, respectively.
- Msg9/10: Finally, after all relevant clients and neighbors have contacted the new server, it sends a *Remap Success* message to both the scheduler and the original server to signal the completion of the migration.

5. EVALUATION METHODOLOGY

We use a prototype implementation of the SimMud game server [8] to study limitations of a single server and basic costs for player and region migration with multiple servers under player flocking. Our experimental testbed consists of a 32-node cluster of dual 1GHz Pentium III machines running 2.4.18 Linux, interconnected by a 100Mbps Ethernet. Because of limitations in this testbed configuration (e.g., each client machine can accommodate only 10 robots), we are unable to fully demonstrate the benefit of dynamic load balancing with experiments. Therefore, we measure update interval delays for a stand-alone server with increases in client load and experiment with multiple servers under player flocking in section 6. We then use the experimental results to calibrate a configurable simulator in terms of safe load and overload thresholds to use for dynamic load management, resource contention curves and server operation costs. We generate 6000 SimMud robot traces for a large game map and present trace-driven simulation results with 100 servers and 6000 players in section 7. We choose the load shedding SLA as a 1 second update interval to clients, and a corresponding load aggregation SLA of 2 second update intervals. Hence the goal of our Locality Aware algorithm is to maintain the client update interval under 2 seconds.

6. EXPERIMENTAL RESULTS

This section presents our experimental results with the actual multi-server SimMud implementation. We first present a single server bottleneck analysis and load threshold determination, then we discuss a simple player flocking experiment and measured player and region migration costs.

6.1 Single server experiments

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

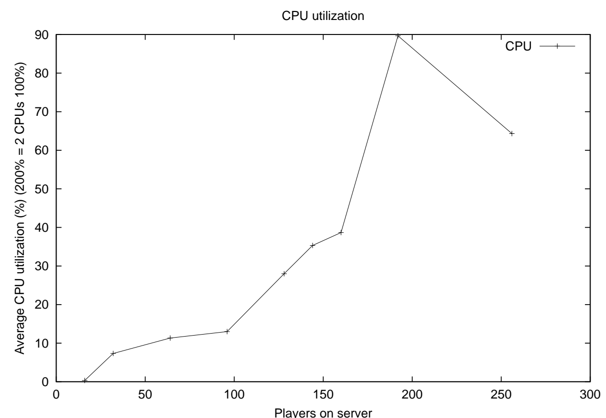

Figure 6: As can be seen, CPU is not a bottleneck. The drop at 256 is due to clients terminating because of network overload.

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 1: Size of server state updates.

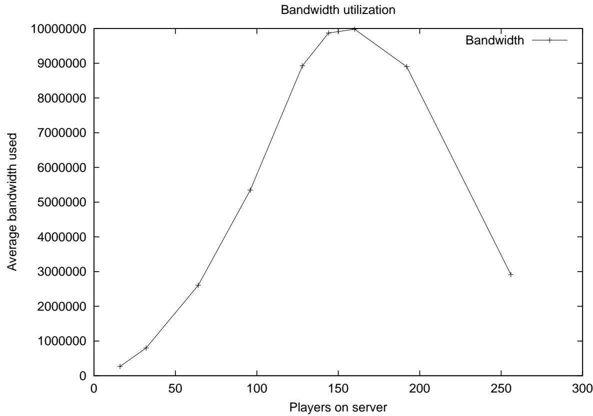

Figure 7: Server bandwidth utilization. When requirements exceed available bandwidth, server update delays skyrocket causing clients to terminate.

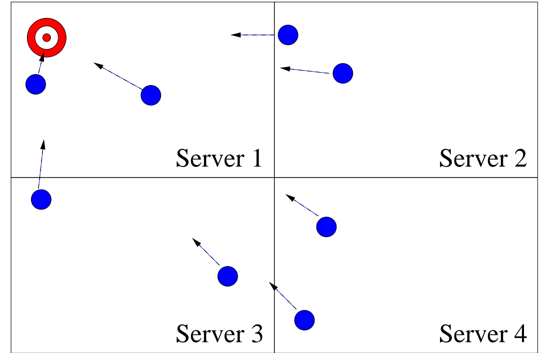

Figure 8: Flocking: Robot players are moving toward a target (quest), in the northwest region (Server 1), many of them crossing server partition boundaries in the process.

In the experiments, 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 sampling intervals, is shown in Figures 6 and 7. The client requests for position updates are approximately 800 bytes each and consist of the entire player object. The size of server updates, shown in Table 1, depends on the number of players — 4.5 KBytes for 4 players up to 60 KBytes for 256 players.

Our experiments show that the server can handle up to 128 players. When increasing the server load further, delays increase sharply — from 214 *milliseconds* for 128 players to 33 *seconds* for 144 players. Figure 7 shows that at 144 players, the bandwidth consumption curve flattens out as the network bandwidth limit is reached, and then falls as clients time-out and terminate due to excessive update delays from the server. The main bottleneck in our environment is thus the server-clients bandwidth. CPU consumption remains low throughout the experiment, with peak consumption being less than half the available cycles from the dual processors.

6.2 Multiple server experiments

A four region world distributed on four servers is set up as shown in Figure 8 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.

Traversal of a region in this case takes about 100 seconds, unless detoured by roadblocks, hence, 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 9, the player density is illustrated as a function of time. Since players that originate from the southeast region must traverse an intermediate region, this intermediate region will be populated longer. At the end of the quest, 123 players have made it to the quest region, with 5 stragglers still on their way.

CPU load and bandwidth consumption (Figures 10 and 11, re-

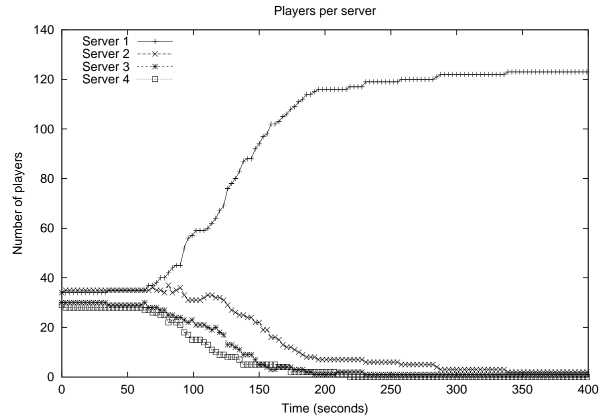

Figure 9: 128 (robot) players distributed over four servers. At time $t \approx 65$ a quest in Server 1's region is created, and players flock there for 300 seconds.

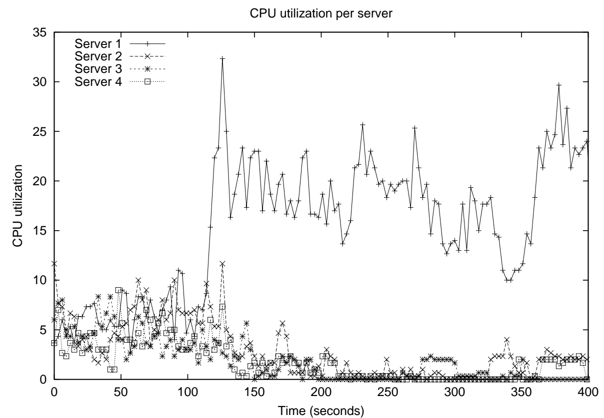

Figure 10: CPU consumption increases on the server hosting the quest (server 1), and declines for other servers.

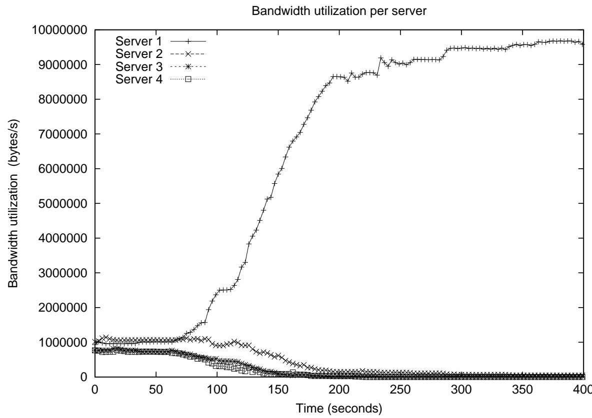

Figure 11: Network bandwidth consumption increases on the server hosting the quest (server 1) to almost the maximum sustainable bandwidth.

spectively) on each server correlate with player density. Server update delays on the server hosting the quest (server 1) increase from 50 ms at quest initiation time by roughly a factor of 4 at the end. On the other hand, since the total number of players on each server does not exceed the overload threshold of 128 players, update delays for all servers are within 250 ms for the duration of the experiment.

During the quest, player migration occurs less than once per second and the 960 byte player handoff messages incur only a minor impact on bandwidth.

6.3 Player and Region Migration Costs

Player migration across a server boundary takes about 100 milliseconds. On the other hand, the cost of region migration is heavily dependent on the number of players in the region. Table 2 shows the migration time as well as the message size of *Remap Request* under different number of players. The migration time is measured from the beginning to the end of the migration. The message size of the *Remap Request* sent from the original server to the new server dominates other migration related message sizes.

7. SIMULATION RESULTS

We use simulation to extrapolate from our experimental results in two different directions: simulating a massively multiplayer game as a state-of-the-art server would support and varying the environment. We have developed a configurable simulator that is powerful enough to model hundreds of servers, thousands of clients and the network between them.

7.1 Simulation Methodology

Each server accepts client requests and performs its normal functions of computing player moves, eat and fight requests which are all simulated in terms of their CPU costs. More importantly, each server sends periodic state updates to all the corresponding players it hosts. In addition, servers send visibility information and other information periodically to their neighbor servers and are able to perform dynamic player and region migrations. The sizes of update, player and region migration messages used in the simulator, are the measured averages in the experiments from section 6, respectively.

The following sections describe our simulation methodology in terms of resource usage, player movement and simulation parameters for different environments.

7.1.1 Resource Usage 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 estimates as measured in the experimental testbed. This calibration of the simulated system against measurement 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 with 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 region of interest.

We measure the periodic update interval experienced by players. We compare the various partitioners through the delays in receiving periodic updates induced by server CPU or network resource contention. The $Overload_th$, $Safety_th$ and $Light_load_th$ are 128, 80 and 32 players respectively, according to the experimental results of section 6.

7.1.2 Player Trace-Driven Simulation

We collect 6000 robot traces by running SimMud robots using the same game configuration of a 100 servers, 400 regions, 10000 by 10000 meter game map with a quest in the center of the game map. Each robot gets a random initial position. The average moving speed of robots is 2 meters per second. The players move towards the hot-spot for a period of time of 1000 seconds, then the hot-spot is removed and players spread out again. The duration of the whole experiment is 2000 seconds. Since our robots stop eating and fighting when a quest is initiated and just move towards the quest, the resulting movement of a robot is independent of others. Therefore, we collect 6000 traces of robots during separate SimMud runs with smaller numbers of concurrent clients, then feed all traces to our simulator to simulate a massively multiplayer game on a server configuration with 100 servers and 400 regions. Initially, regions are allocated to servers using static block partitioning corresponding to a block of four regions per server. All simulations correspond to player traces for the same center hot-spot location that attracts all players. We have varied the time that the players stay in the hot-spot area and the hot-spot location across experiments, and we have determined that our results are largely insensitive to these parameters.

7.1.3 Environment Simulation Parameters

We explore dynamic versus static partitioning of the game world under two different server configurations: i) a LAN-based centralized server with 100 Mbps inter-server and 100 Mbps server-client bandwidths corresponding to our experimental platform and ii) 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) with a 100 Mbps *shared* network bandwidth towards both its clients and its neighbor servers. In all environments the CPU power and contention is modeled after our experimental environment.

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 2: Cost of region migration.

7.2 Results for Dynamic Load Balancing Algorithms

In Figures 12 and 13 we show a comparison of the three dynamic partitioning algorithms, the main Locality Aware algorithm (Locality), dynamic partitioning that sheds to the lightest loaded server (Lightest), Spread dynamic partitioning and our baseline Static partitioning algorithm in the LAN and WAN distributed environments, respectively. The graphs also contain a variant of our main Locality Aware algorithm without partition merging (Locality w/o Merge), where only load shedding adaptations are used in response to server overload but partition aggregation is disabled. This allows us to separate the relative effects of the two components of our Locality Aware algorithm.

Both figures show the average update interval for players, using a 50 second sampling period, for the least responsive server over that period of time; note that this server could be different for different sampling periods. Overall, from both figures, we see that congestion causes severe degradation in the average update interval to clients in the Static protocol. Furthermore, the dynamic partitioning protocols perform better overall compared to Static partitioning for both platforms.

In the LAN environment (Figure 12), due to the low penalty of all types of inter-server communication, such as, player hand-offs, region migration and area of interest consistency maintenance, all dynamic partitioning algorithms work well. The small latency peak experienced by the two Locality versions and Lightest is due to significant player crowding in the *single region* of the quest, when dynamic load shedding cannot be performed. In contrast, Spread changes region assignments frequently such that no single server holds the quest region for a significant amount of time; hence each individual server's average latency is the best in Spread. On the other hand, while we don't currently use *any* server admission control policy, limiting the *per-region* player density within a server may be reasonable, thus avoiding this problem for our main algorithm.

In contrast to the relative performance graph of the LAN server, in the WAN environment (Figure 13), only the Locality Aware algorithm manages to maintain an acceptable average update interval after triggering adaptation to overload. Because they 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 load-shedding adaptation. We see that this applies to Locality w/o Merge as well, although to a lesser extent. The performance of the Locality Aware algorithm is very similar to the one of Lightest in the initial phases of hot-spot creation because all players are close to the hot-spot location and neighbor servers are over the safe load threshold themselves. Thus, both algorithms shed to remote servers. Both Locality Aware versions outperform Lightest during the second part of the simulation, when the inter-server communication begins to penalize algorithms that have not maintained region clustering on servers. Finally, partition aggregation in this second portion of the graph triggers automatically in our main Locality Aware algorithm, since the quality of service is not met even in normal load due to the increased inter-server communication. We see that partition merging corrects the locality disruptions caused by remote load shedding for the most part, nearly closing the performance gap to the best locality algorithm, static partitioning, at the end of the run.

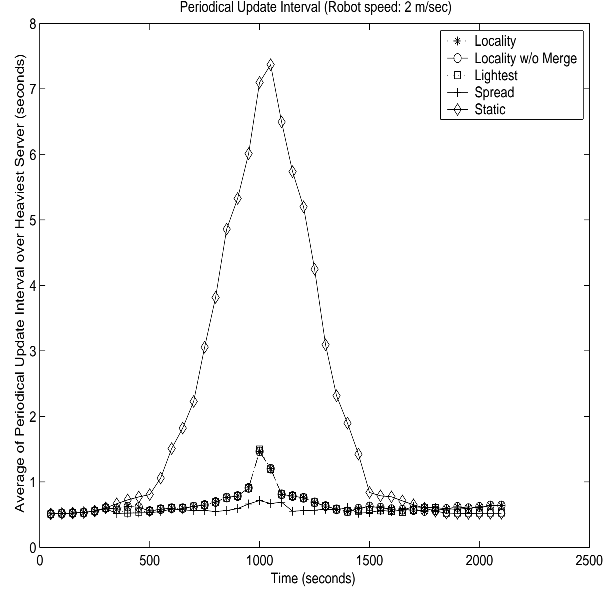

Figure 12: Comparison of Static Partitioning and Dynamic Load Partitioning for centralized LAN-based server, 1000 seconds hot-spot duration. The network to clients bandwidth is the bottleneck during the hot-spot.

Metric	Loc	Loc w/o M	Lightest	Spread
Reg. Mig.	45	34	29	88058
Reg. Clu.	104	108	116	399

Table 3: Comparison of the total number of region migrations and the number of region clusters at the end of simulation in WAN setting.

In order to differentiate the inter-server communication costs further, Table 3 shows a comparison under the WAN environment of all our dynamic partitioning algorithms in terms of total number of region migrations and the number of strongly connected region clusters at the end of simulation. We see that the Locality Aware algorithm (Loc) maintains the best region locality by keeping the number of region clusters close to the initial value of 100 region clusters (corresponding to the initial block partitioning on 100 servers).

Locality w/o Merge (Loc w/o M) migrates fewer regions than our main Locality algorithm since it does not respond to the SLA violations caused by inter-server communication. Its higher degree of locality disruption compared to the Locality Aware algorithm is reflected in a higher number of strongly connected components.

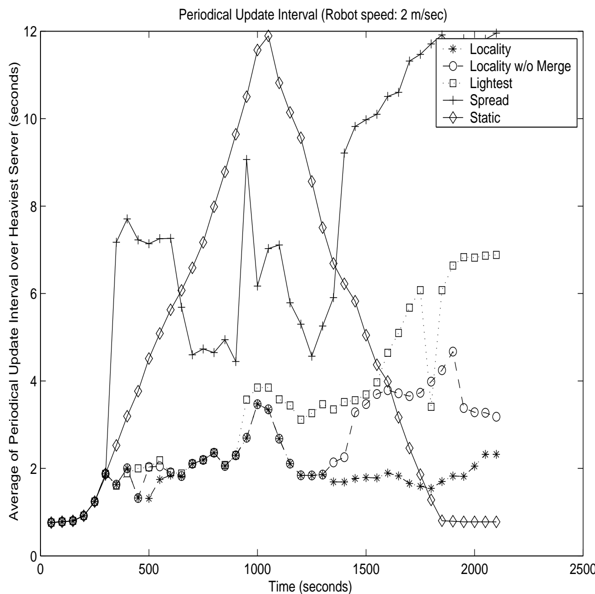

Figure 13: Comparison of Static Partitioning and Dynamic Load Partitioning for WAN server, 1000 seconds hot-spot duration. The network to clients bandwidth is the bottleneck during the hot-spot.

While the absolute difference in the numbers is small, 108 versus 104, these are overall numbers, while the persistent impact on individual servers involved in a remote load shed may be high. Figure 13 captures the highest individual server impact since it shows the update interval of the worst performing server.

The total number of region migrations in Lightest is less than in Locality Aware, as expected, because Lightest always selects lightly loaded nodes to shed load to instead of first considering neighbor nodes. Correspondingly, due to more remote load shedding, the algorithm induces more locality disruption than both Locality Aware versions. On the other hand, compared to Spread, Lightest still attempts to shed regions as connected clusters and involves a single server in load shedding. We can see that Spread does much worse in terms of both metrics because it completely disregards locality maintenance and does not try to minimize the number of region migrations. Hence, its high inter-server communication impacts its performance in the wide-area environment.

8. RELATED WORK

Companies, such as, Butterfly.net [3] and TerraZona [15] develop middleware that provides cluster support for MMOGs. However, these systems provide only static partitioning and newer, more dynamic games like Sims Online and Tabula Rasa can not be effectively handled by them [11]. Each server can support only one region at a time. Updates do not propagate across regions.

Many on-line load management algorithms [1, 9] require global knowledge about the load of all tasks and resources, which is usually not available or prohibitive to obtain in terms of communication in a distributed environment. In particular, Lui and Chan [9] reduce load balancing in distributed virtual environment systems to a graph partition algorithm, and assume global knowledge of load and pairwise communication cost between avatars.

CittaTron [7] 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 our Locality Aware algorithm, but they do not explicitly consider the communication cost resulting from lack of locality.

An online load balancing algorithm using localized adaptations is presented by Ganesan et al. [6] for range-partitioned databases in peer-to-peer systems. Their objective is to minimize the number of tuples moved to achieve a desired constant imbalance ratio, while keeping the tuple range contiguous for any specific partition. Our optimization problem is more complex due to its higher dimensionality, contradictory optimization goals and the need for a fast and efficient solution under severe hot-spot overload.

An orthogonal approach addressing graph partitioning in high dimensions is to use space-filling curves [2], such as, Hilbert curves, for mapping a multidimensional map into a uni-dimension space while preserving some proximity information present in the multi-dimensional space.

Finally, our dynamic load balancing algorithm has similar goals to load management in cellular networks [4]. Just like game players, mobile users may transfer from an area managed by a particular base station to another, incurring hand-off calls. Mobile client crowding in any particular cell may cause the corresponding cell to run out of available channels. Dynamic channel allocation algorithms allow an overloaded cell to borrow channels that do not cause interference from neighbor cells.

9. CONCLUSIONS

This paper presents a load balancing algorithm that enables a seamless game world in which players can freely move and interact with each other across server partition boundaries. Our experiments have shown that both the dynamic approach and the locality aware algorithm are crucial to the performance of game servers. First, the dynamic approach reassigns regions to servers at runtime to accommodate spontaneous flocking to game regions. Second, the load balancing algorithm is aware of spatial locality in the virtual game world and aims to reduce the cross server communication caused by transparent partition boundaries. Our algorithm attempts to collocate adjacent regions onto the same server while maintaining balanced load among servers.

We have designed and implemented the load balancing algorithm and applied it to the SimMud game. Our evaluation is based on a game configuration with 100 servers and 6000 independent robot players distributed over 400 regions. Results show that the locality aware dynamic load balancer improves performance by up to a factor of 8 compared to static partitioning, and by up to a factor of 6 compared to a load balancing algorithm that does not consider spatial locality.

10. REFERENCES

[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. `www.butterfly.net/platform/`.

[4] Guohong Cao. Integrating distributed channel allocation and adaptive handoff management for QoS-sensitive cellular networks. *Wireless Networks*, 9(2):131–142, 2003.

[5] Jin Chen. Locality aware dynamic load management for massively multiplayer games. Master's thesis, University of Toronto, Jan 2005.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Mike Mika and Chris Charla. Simple, cheap pathfinding. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 155–160. Charles River Media, Inc., 2002.

[11] Mitch Ferguson and Michael Ballbach. Product review: Massively multiplayer online game middleware, January 2003. http://www.gamasutra.com/features/20030115/ferguson_01.htm.

[12] Katherine L. Morse. Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27, University of California, Irvine, 1996.

[13] NC Soft. Lineage, 2002. http://www.lineage.com/.

[14] Sony Computer Entertainment Inc. Everquest online adventures, 2002. everquestonlineadventures.station.sony.com/.

[15] Zona Inc. Terrazona: Zona application frame work white paper, 2002. `www.zona.net/whitepaper/Zonawhitepaper.pdf`.