Proximity: A Scalable P2P Architecture for Latency Sensitive Massively Multiplayer Online Games

by

Kamran S. Malik

B.Sc., Lahore University of Management Sciences, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University of British Columbia

October, 2005

© Kamran S. Malik, 2005
Abstract

P2P overlays are a natural architecture for scalably supporting Massively Multiplayer Online Games. However, computing consistent game state in a distributed fashion without compromising responsiveness places very hard to meet bandwidth, latency and scalability requirements on the architecture. We present Proximity - a system that utilizes the limited Area of Interest of game entities to arrange player contributed machines in an overlay that optimizes bandwidth consumption, reduces event propagation delays and scales dynamically with an increasing number of players. Network proximity of nodes is also taken into account during the overlay construction process. In this thesis, we describe how to split game state management over unreliable player machines while meeting the fault tolerance, state persistency and consistency requirements of games. We anticipate our system, with its increased scalability and lower latency, opening the possibility of completely new genre of MMOGs with fast paced reflexive action at a fraction of a cost of the more traditional client server architecture.
Contents

Abstract.............................................................................................................................. ii

Contents .......................................................................................................................... iii

List of Tables ................................................................................................................... vii

List of Figures ................................................................................................................ viii

Acknowledgements ....................................................................................................... x

Chapter 1 Introduction ................................................................................................. 1

1.1 Motivation ................................................................................................................ 2

1.2 Thesis Contributions ............................................................................................. 4

1.3 Thesis Organization .............................................................................................. 6

Chapter 2 Background and Related Work .................................................................... 7

2.1 Scalability and Peer Architectures ........................................................................ 7

2.2 Peer-to-Peer Architectures .................................................................................. 8

2.3 Related Work ........................................................................................................ 9

2.3.1 Problems with exiting P2P solutions ............................................................... 10

2.4 Goals of this Architecture ................................................................................... 12

Chapter 3 System Overview ....................................................................................... 13

3.1 Direct Player to Player connections ...................................................................... 13

3.2 Neighbor Discovery ............................................................................................. 14

3.3 Game State ........................................................................................................... 15

3.4 Fault Tolerance .................................................................................................... 16

3.5 Persistence ............................................................................................................ 17
3.6 Coordinator Overlay .................................................. 18
3.7 Overlay Construction .................................................. 18
3.8 Routing ........................................................................... 19
3.9 Transparent Borders ..................................................... 21
3.10 Splitting & Merging ....................................................... 22
3.11 Dynamic Area of Interest (AoI) ......................................... 23
3.12 Overload Detection ....................................................... 24
3.13 Topology Aware Overlay Construction ......................... 24
3.14 Replica Replacement ..................................................... 25

Chapter 4 Overlay Network Management ........................................ 26
  4.1 Routing Table .................................................................. 26
  4.2 Handling Splits .............................................................. 27
  4.3 Intelligent Split .............................................................. 33
  4.4 Bootstrapping the child ................................................... 33
  4.5 Merge Algorithm ........................................................... 34
  4.6 Locking ............................................................................ 35
    4.6.1 Locks and Splits ....................................................... 38
    4.6.2 Locks and Merges .................................................... 38

Chapter 5 Overlay Routing .......................................................... 41
  5.1 Basic Routing Algorithm ............................................... 41
  5.2 Grid .............................................................................. 44
  5.3 Big Hop ...................................................................... 45
    5.3.1 Loop Avoidance Rule .............................................. 48
Chapter 5

5.4 Lies Within .................................................. 49
5.5 Reverse BigHops ............................................. 50
  5.5.1 Reverse Tree Construction ............................. 52
  5.5.2 Routing with Reverse BigHops ......................... 53

Chapter 6 Architecture ........................................... 56

6.1 Player Code.................................................. 56
6.2 Coordinator.................................................. 59
6.3 Central Server .............................................. 60
6.4 Inter-Component communication ........................... 61
6.5 Removing CS Bottleneck ................................... 64
  6.5.1 Secondary Overlay ..................................... 64
  6.5.2 Reusing Primary Overlay ............................... 65
6.6 Comparison with other DHT based Schemes ............. 65

Chapter 7 Performance Evaluation .............................. 67

7.1 System Properties .......................................... 69
7.2 Scalability .................................................. 73
7.3 Routing ..................................................... 75

Chapter 8 Conclusions ........................................... 78

8.1 Future Work: Service Discovery .......................... 78
8.2 Conclusion .................................................. 80

Appendix A Derivations .......................................... 81
  A.1 Expected Average Subscribers per Player ............. 81

Appendix B Bounded Multicast ................................ 83
B.1  Bounded Multicast and Range Queries.......................... 83
B.2  Query Termination.................................................. 86

Bibliography ....................................................................... 88
# List of Tables

Table 3-1 Neighbor Registry and Routing ........................................... 20
Table 4-1 Symbols used in the Locking Algorithm .................................. 36
Table 5-1 Definitions used in Reverse Description ................................. 54
Table 6-1 Message Types ....................................................................... 64
Table 7-1 Routing Comparison .............................................................. 77
List of Figures

Figure 2-1: Kawahara's Node Blindness .................................................11
Figure 3-1 Neighbor Registry and Routing ...........................................20
Figure 3-2 Transparent Boundaries .......................................................21
Figure 4-1 Vertical Split .................................................................28
Figure 4-2 Post Split Configuration ......................................................30
Figure 4-3 Horizontal Split .............................................................31
Figure 4-4 BoundaryMatcher (Region 3) - Horizontal Split .....................31
Figure 4-5 Split with Trespasser ......................................................32
Figure 4-6 Merge - CAN_announceReplace messages ..........................34
Figure 4-7 Inconsistencies caused without Locks .................................35
Figure 4-8 Merges and Deadlocks ......................................................39
Figure 4-9 Why parent should unlock neighbors locked by child ............40
Figure 5-1 Boundary Assignment ......................................................42
Figure 5-3 Zone Origin .................................................................43
Figure 5-4 Effect of origin choice ......................................................44
Figure 5-5 Grid and Loops .............................................................44
Figure 5-6 Relative Location and BigHop Links ..................................46
Figure 5-7 Hopping Subtrees .........................................................47
Figure 5-8 Loop Avoidance Rule ......................................................48
Figure 5-9 Lies Within .................................................................49
Figure 5-10 Subspaces not connected by BigHop .................................50
Acknowledgements

I would like to thank my supervisor, Dr. Son Vuong, for his expert guidance, unending support and encouragement. None of this could have been possible without his backing and sincere advice regarding all aspects of my studies at UBC. I thank Dr. Charles Krasic for always keeping his door open to students like myself seeking advice and feedback. I would also like to express my heartfelt gratitude to Michael Blackstock. He acted as sieve to help me sift through ideas through out the course of the project. A number of proposed improvements to the overlay network were stimulated by the many long discussions with him. I wish him the best of luck with his Doctorate work.

Finally, I thank my family for their understanding and always being there whenever I needed them.

Kamran S. Malik

The University of British Columbia

September 2005
Massively Multiplayer Online Games (MMOGs) are games played over the Internet that involve thousands of concurrent players who interact in real time with other players and entities that share the game world. MMOGs are distinguished from other multiplayer games by the large number of players and the long term persistent storage of the game state. Games like Lineage have recorded 180000 concurrent players and the number of its subscribers exceeds two million [1]. In a MMOG, game entities are always present regardless of the presence of the player that introduced them. Changes made to the game world in a certain area by any player are recorded and must be visible to any player that might pass the same part of the world later even after the player who made the change has left the game. The ability of a player making lasting changes to the world makes the game feel more realistic and appealing. In summary, two properties that differentiate MMOGs from other multiplayer games are the number of players involved and the persistence of game state in large virtual worlds.

Most commercial MMOGs available in the market use the traditional client-server model that limits the scalability of the system and places an enormous financial burden
on the publishing company to furnish enough resources to accommodate the large number of players. In addition, it also restricts the type of games that can be supported. In recent years there has been a push towards P2P architectures because of their inherent scalability. Peer-to-peer architectures allow games to utilize the storage, bandwidth and computing power available on the player machines. However, a number of technical issues are impeding the commercial popularity of this approach.

1.1 Motivation

Traditionally, MMOGs have used a client server architecture where the server is responsible for storing and computing the state of the shared world based on the inputs gathered from all the participating players. The client running on the player’s machine can range from being as a simple text terminal to a complex three dimensional rendering system that shows the player a window into the virtual world. The clients receive input from the user, convert them into commands and pass them along to the server. The server collects the commands from player machines on the network and runs a game world specific simulation algorithm on the gathered commands to generate the next state of the game world. It then sends updates based on the current state of the game world to the clients depending on their area of interest. Since the state of the world is derived from input from all the clients together at a single central location, the players can see the effect of the combination of each other’s actions.

A client may maintain a cache of the state of the virtual world and its objects within the area of interest of a player’s avatar. The server sends them updates only for the objects within this perceivable area. The clients apply these updates to the objects in the cache and update the player’s view of the world.
Since only the server is involved in deciding the state of the entire world it is much simpler to maintain consistency and deal with delayed and lost messages from clients. The game company can also easily control player admission and game state through the central server.

However, as all player machines have to communicate with the server it can quickly become a computing and network bottleneck because of the potentially large number of players involved in MMOGs and the size of the game state. Scalability of the client server model may be improved by employing groups of servers arranged in a cluster [24] or a grid [25]. The game world is split and assigned to different servers on the cluster. A server thus maintains only a part of the virtual world and only players that reside in that portion of the game world connect to that server. As players move through the world they may have to migrate to different servers. Terazone using a cluster of servers that individually could only handle 2000 to 6000 clients was able to support 32000 concurrent players over the cluster [24].

Other schemes divide the server workload by assigning players based on their physical network proximity to different servers running parallel independent worlds. This essentially side steps the problem of scalability as it limits interaction to only players connected to a shared server. Players in different “parallel” worlds cannot interact with one another.

Although these architectures can be made to scale with the number of players by adding more centralized resources, it fundamentally lacks flexibility and the centralized server(s) must be over-provisioned to handle peak loads. To do this, game publishers are forced to estimate in advance the number of players they expect to play their games. To reduce costs providers often under-provision, limiting the growth of appealing games,
which translates into lost revenue for the company. Even if the company is able to predict and provision the right amount of resources, these resources remain under utilized while the game popularity rises to the predicted level. Similarly, when a game’s popularity starts to wane the gaming company is unnecessarily forced to maintain the original level of resources.

Moreover, in all forms of client-server architecture there exists an upper limit on the responsiveness of the user actions as they affect the overall game state since clients have to wait for commands to propagate to a possibly busy server and the resulting updates back to the client. This limits user responsiveness and even prevents certain genres of games like first person shooter games) from becoming massively multiplayer.

1.2 Thesis Contributions

This thesis discusses these impediments and as a solution proposes an architecture that allows player machines to contribute their resources for game state management by becoming a part of an overlay that dynamically distributes the load amongst their machines. The architecture exploits area of interest management to maximize the utilization of the pooled resources. It also describes methods for using existing distributed game state management algorithms with the proposed architecture.

The biggest contribution of this thesis, we hope, will be to revive interest in a genre of overlay networks that allows semantic placement of data on the overlay better than other more popular overlays. We exploit this property to map regions of the game space on the overlay so that close-by entities in the virtual world are co-located on the overlay, as well. To highlight, this thesis:
• Provides a description of a completely distributed dynamic algorithm for assigning game state management responsibilities, to share the load over a set of machines and thus, improve the scalability of the MMOG infrastructure. The division of the game state is completely transparent to the game entities as their perception is not limited by any artificial boundary imposed by the assignment algorithm.

• Reduces infrastructure cost by providing ways to utilize resources available on the participating players’ machine.

• Allows direct communication between the source of an event and all entities affected by it to reduce communication latency and improve response time. It utilizes Area of Interest (Aoi) to transmit an event to only those entities that need to be aware of it.

• Lays out methods for maintaining the consistency of the game state spread over a set of peers and means for guaranteeing state persistence in the face of peer failure.

• Describes distributed algorithms and synchronization issues involved in the topology-aware construction of the overlay. These details are not available in other published work on similar overlays.

• Proposes utilization of the network topology aware construction of the overlay to assign coordinating responsibilities to reduce network traffic and latency.

• Presents schemes for bounded multicast over well defined subspaces of the overlay coordinate space. Existing multicast algorithm cannot guarantee
non-duplication and only support broadcast of the message to all nodes in the overlay.

- Describes how bounded multicast can be used to make iterative range based queries.

- Explains how a tree can be laid on top of the overlay and used to route messages leading to a reduction of routing cost from $O(N^{0.5})$ to $O(\log N)$.

- Describes an extension of the infrastructure for Service Discovery — another application that can greatly benefit from the semantic placement of information.

1.3 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 makes a case for a P2P based approach, discusses some recently proposed attempts to adopt this technology and lists their shortcomings. In Chapter 3, we provide the high level details of Proximity, our proposed gaming architecture. Chapter 4 presents detailed algorithms for constructing the underlying P2P overlay. In Chapters 5, we describe our proposed improvements to the overlay to allow more efficient routing. The main components of the gaming architecture are presented in Chapter 6. We evaluate our proposals in Chapter 7 and conclude in Chapter 8 with a word on possible extensions of the architecture in the realm of Service Discovery. We present details of the multicast routing and iterative range queries in Appendix B.
Chapter 2

Background and Related Work

This chapter makes a case for a peer-to-peer architecture for MMOGs, introduces some recently proposed peer architectures and delineates their major shortcomings.

2.1 Scalability and Peer Architectures

For a system to be scalable it should allow large number of components to join and leave the system arbitrarily. Resources on any given node are limited and are consumed as components join the system. A system can allow new comers to join only when it has enough spare resource to accommodate the new members. A system can continue doing this as long as resources are not depleted when new nodes join. To prevent resource depletion two strategies can be exploited:

Resource-growing: resources increase at the accepting node with the addition of the new node.
Decentralized end point resource consumption: addition of new members should not use up any centralized resource.

2.2 Peer-to-Peer Architectures

Peer to peer architecture comes naturally to mind as a system which can be built around both these two strategies. A number of structured P2P overlays have been proposed. CAN [13], Chord [14], Pastry [11], and HyperCast [15] being the more popular ones. These networks focus on setting up a distributed hash table that maps values to keys and provides means to route through the network to locate those keys to retrieve the associated values. Recently, there has been considerable research published on building games on top of these DHTs with varied success.

Early multiplayer games and even more recent research and commercial work rely on a simple P2P model. Games like MiMaze [16] and Age of Empires [17] implement a completely decentralized unstructured P2P system where every node receives moves from all the players in the game and decides the game state on its own. These games depend on the underlying network to support broadcasting or multicasting. All nodes are equal and model the entire world based on the messages received from all the other players. Since all nodes run the exact same algorithm on identical sets of inputs, all players see the same world on their screens. Other P2P systems such as AMaze [19] and Mercury [18] utilize multicast groups to send messages only within groups of nearby players to minimize the bandwidth requirements. The applicability of multicast is severely limited as IP multicast is still not supported widely and architectures that try to mimic multicasting by the use of unicast links introduce too much latency.
However, to adapt P2P technology for massively multiplayer gaming, which involves large number of players spread all over the Internet and not necessarily on the same LAN a number of enhancements are necessary. We can no longer assume that (i) messages can be easily broadcasted to all participants (ii) all the intended recipients receive all the messages within guaranteed time constraints. These systems should also be able to support large number of shared mutable environment object in addition to the player entities both of which should be stored persistently.

To remove the need for broadcasting all messages, nodes have to be arranged in a topology where an avatar’s limited area of interest can be exploited to minimize the number of connections between nodes. Sophisticated consistency management techniques must be used to deal with delayed and lost messages.

2.3 Related Work

In recent years a number of researchers have suggested improvements to P2P architectures to meet MMOG requirements. Knutsson et al. [2] propose making all player machines members of a Pastry Overlay [11]. The game map is split into zones and all players within a zone form a group by joining the Scribe [12] multicast tree for the zone. Zone IDs are used to hash onto Pastry nodes to select zone coordinators, which also serve as roots for the multicast trees. Game state is updated by broadcasting the moves on the multicast tree using the coordinator’s ID.

Kawahara et al [7] advocate a fully distributed scheme where all nodes are equal. A player node maintains a list of a fixed number of nearest neighbours and forms direct
connections with the neighbours included in the list. Neighbours periodically exchange their lists to help each other detect new incoming nodes.

In Solipsis [8] nodes attempt to maintain links with a ring of nodes around themselves instead of just connecting to the N nearest nodes. The nodes in the ring act as sentinels and warn the node when an unknown node approaches the central node from any direction.

2.3.1 Problems with exiting P2P solutions

Knutsson et al. [2] describe the most complete system of the three architectures as Solipsis and Kawahara et al. [7] do not consider the issue of game state management. However, Knutsson's approach still has some undesirable properties. The cell sizes do not correspond to the area of interest of the players. As it is, players cannot see across artificial cell boundaries introduced by the game architecture otherwise non-existent in the game world. The authors recommend players with greater visibility requirements to join multiple zones. But this results in a node receiving unnecessary messages from locations beyond its area of interest. More seriously, the scheme seems to go against the very reasons that make the peer approach appealing. Player updates are routed through the coordinator, which is not much different from a client sending updates to a server. Secondly, the location of players and proximity of neighbouring virtual world zones are not used in deciding how to stitch nodes together while constructing the overlay. As a result, messages have to be routed through multiple nodes that may not even be interested in the contents of a message. There may be as many as 50 virtual hops [5] in a decent sized game. To make matters worse, each hop between nodes handling neighbouring virtual world zones translates to multiple hops on the physical network. Moreover, as
Pastry tries to randomize neighbours in the overlay to improve fault tolerance, the messages have to haphazardly bounce all over the Internet before it gets to the destination. All of this adds to the delivery time of updates, adversely affecting the responsive of the application.

The main problem with [2] is that it does not fully utilize the possibility of exploiting direct connections made available by the P2P approach. On the other hand, Kawahara's method maximizes the hop count efficiency by maintaining direct connections between players at the cost of increased network traffic. Moreover, maintaining a list of a fixed number of nearest nodes does not guarantee global connectivity. For instance, if all the N closest nodes crowd together on only one side of the node, the player is rendered completely oblivious to a player approaching from the other side. This situation is depicted in the following diagram (Figure 2-1) where N=3 and A is unable to detect B as it was not connected to C:

![Figure 2-1: Kawahara's Node Blindness](image)

Like Kawahara et al., Solipsis utilizes direct links between neighbours improving the event delivery delay. It also solves the problem of node blindness due to overcrowding on one side, depicted in Figure 2-1, by ensuring that there is always a ring
of neighbours directly connected to the node regardless of their distances. However, proper neighbour discovery is still not guaranteed when players move too fast. Secondly, to maintain global connectivity players have to maintain connections even if they are too far apart to be visible to each other.

2.4 Goals of this Architecture

Keeping all of these considerations in mind, we designed our system with the following objectives:

- Maximize the responsiveness of the game by reducing the amount of time packets take to get to recipients
- Incorporate techniques for maintaining the consistency of the game across all clients
- Minimize the bandwidth and computational requirements for a single client.
- Provide fault tolerance against peers unpredictably leaving the system
- Provide means to manage the game state for player controlled avatars (PC), non-player controlled entities (NPC), immutable and mutable terrain objects.
- Provide distributed persistent storage of game state.
- Where possible ensure the security of the system.
- Allow the game publisher to retain admission control.
- Make no assumption about the existence of a multicast infrastructure.
This chapter presents the high level details of our system called Proximity. We describe how the locality properties of the game are leveraged to distribute game state management over an overlay of player machines without imposing any visible side-effects to the game play. We also discuss how game level semantic information can be utilized to structure the overlay to minimize the management overhead, network traffic and latency.

3.1 Direct Player to Player connections

The design exploits the limited area of interest of a player to directly interconnect small subset of player machines. Each node (or player machine) is responsible for just the avatar of the machine's owner. Avatars have a limited area in which they can bring about changes. This is known as the publish area. An avatar also has a subscription area that conforms to the sensing abilities of the avatar. These areas can have different dimensions
depending on the capability and sensing ability of the player’s avatar. A player can see a player, B, only if its subscription area overlaps the publish area of player B. All players whose areas overlap are connected directly with each other. Every time a player moves it sends a move event to all the connected peers and checks if the areas still overlap. If they do not, the peers drop the connection and are no longer aware of each other.

Using publish/subscribe areas is the most efficient and flexible interest management technique. It ensures that only participants who are interested receive only the events that can possibly affect them.

### 3.2 Neighbor Discovery

As players move around they require some mechanism to discover their new neighbours. In Proximity, some of the player machines are promoted to play the role of a zone coordinator. Zone coordinators are assigned fixed portions of the game world called zones to manage. The coordinators have knowledge of all the avatars currently residing in their zone. They keep a record of their precise location and publish/subscribe areas, so that they can inform players when they come close enough to each other to interact. It is up to the players to decide when to stop communicating with a neighbour.

Zone coordinators form a structured peer-to-peer overlay. The entire game world is split up into non-overlapping rectangular regions/zones. Each coordinator knows about the zone coordinators of bordering zones and maintains direct unicast links with the neighbouring coordinators. Through these neighbouring coordinators it has indirect links to coordinators covering the entire game space. These indirect links can be used to route a message to any point in the virtual world. In this way, coordinators maintain global
connectivity by only keeping a record of a small subset of neighbouring zone coordinators.

3.3 Game State

A successful MMOG middleware should provide support for maintaining consistency and persistence for the game state. This becomes even more important when game state management is delegated to player machines that can arbitrarily join/leave the system. Game state can be broken down into the following classes with interaction between different classes of objects requiring varying level of latency and consistency guarantees:

- Player controlled Character (PC)
- Non-Player Controlled Character (NPC)
- Immutable terrain objects
- Mutable terrain objects

Terrain objects are always located at fixed points on the map. The zone coordinator for that position of the map computes and stores the states of these objects. As an avatar moves through the world the coordinator informs its PlayerMD (the middleware component running on the player's machines) of the positions and states of the all the terrain objects in the avatar's surrounding. The player middleware caches these values and informs the relevant coordinator whenever its avatar interacts with any of the terrain objects. The coordinator resolves any conflict associated with concurrent updates to a terrain object and refreshes the cached value of the shared environment object in all the players in the vicinity.
For more latency sensitive interactions like PC-to-PC interactions, PlayerMDs are allowed to optimistically synchronize with each other, involving the coordinator only when necessary to resolve inconsistencies. For two players to interact they must be in close proximity. As discussed in the previous section these players maintain direct connections between them. Using these links a player informs all nearby players of its actions. The player nodes gather these events and periodically recalculate the new state of the world applying the Trailing State Synchronization algorithm [4]. The same algorithm is run on the coordinator but with a greater lag, which ensures the delivery of more events hence allowing the coordinator’s simulation engine to arrive at a more consistent state some time later.

The state computed at the coordinator is considered to be more authoritative. Whenever an inconsistency is detected between the states maintained by players they revert to the state of their coordinator.

Direct connections between players and the resulting lower consistency delays improve the responsiveness of the clients and allow players in fast paced games to take a series of quick reflex actions. The resulting events are only exchanged between interacting players or transmitted to other nearby players within viewing distance of the action. The higher level of network communications is limited to this small set of players.

3.4 Fault Tolerance

To protect against ungraceful node failures, every coordinator maintains a chain of replicas, which are periodically updated. Only one of the replicas actively exchanges messages with the players and neighbouring coordinators, and computes the states of the
objects located in the zone. At a fixed interval the active coordinator updates one of the replicas, which relays the updates to the remaining replicas. Since the replicas expect to be updated at regular intervals, if a replica does not receive an update for some time it conclude that an upstream replica has gone down and initiate a process to select new PlayerMD to replace the lost replica, which could possibly be an active coordinator.

Replicas save information regarding the players in the region, the active neighbouring coordinators, their replicas and the latest state of the game objects located in the region. When a replica detects a failure of the active coordinator, it informs all the constituent players, neighbouring coordinators and its own replicas that it is taking over the zone from the lost coordinator. The PlayerMDs resend all the events from their cache that the replica might have missed because of the old coordinator failing before updating its replicas.

3.5 Persistence

To insure tolerance for catastrophic failure and allow a minimum level of persistence there is a lightweight background process, which backs up the state of the different zones of the game world onto the central server. The publishing company can keep a copy of the state of the entire world secure on its own machine. This copy may be out of date by several minutes but it provides a minimum level of protection against a situation where there is a mass exodus of players. In the event of such a catastrophic failure the game will lose the last few minutes of moves. Given the non-critical nature of the application, this is fairly acceptable. We foresee that only in very rare circumstances the game will have to be reset to a slightly outdated state saved at a backup server. Most failures will be sufficiently handled by the replicas of the active zone.
3.6 Coordinator Overlay

Our design of the overlay centers on a 2 dimensional Cartesian coordinate space. The entire coordinate space is dynamically partitioned among the coordinators as needed. Each coordinator is assigned its own independent zone within the overall space. A coordinator discovers and tracks other nodes that coordinate zones bordering its own zone. Using this set of coordinators a node can route messages to any random point on the virtual coordinate space.

3.7 Overlay Construction

The overlay is constructed incrementally with the initial coordinator started by the central server on a computer provided by the game publisher. The first coordinator manages the entire game space and maintains a list of all player machines currently playing the game. It can run on the same or a different machine as the central server depending on the resources available to the game company.

As players join the game the initial coordinator can become overloaded. When that happens, the system dynamically selects one of the player nodes to share the coordinator's load. The overloaded coordinator splits its zone in half and transfers control of one half to the selected player machine while retaining the other half.

This is achieved through the following multi-step process:

1. A coordinator, C, detects an overload locally and sends a distress signal to the central server.
2. The central server chooses from its list of available player machines the best candidate to share C's load and sends it a signal to spawn a coordinator process on its machine.

3. The spawned coordinator, D, informs the overloaded coordinator that it is ready to share the load.

4. The overloaded coordinator decides the orientation of the split and passes the dimensions and coordinates of the newly created zone to the new coordinator, D.

5. C (the original coordinator) informs its neighbours of the split and passes D information about its inherited neighbours. Based on this information new links are established between D and the old neighbours of C.

6. C also passes on to D states of all the players and objects that lie within its zone. These transferred players are asked to establish connections with D.

7. As a last step, D asks the central server to pick player nodes to act as replicas and copies its acquired game object states to all the replicas.

3.8 Routing

Each node divides its neighbours in four sets as shown in Table 3-1. On receiving a message to route to a position (x,y) the node first decides which neighbour set it should search for the next hop.
Table 3-1 Neighbor Registry and Routing

Within a set the neighbours are sorted in increasing order of their origins - with the origin being the coordinates of top left corner of the zone. The node goes through this list looking for a coordinator that has the smallest Euclidean distance to the message’s destination and forwards the message to the selected coordinator.

Figure 3-1 shows an interesting twist to this otherwise simple routing mechanism. Neighbour N3, has its origin coordinates in Region 0 but it crosses into and covers the entire Region 1. If node A, were to follow the simple routing algorithm mentioned above it would return an error because (x,y) lies in region 1 but the coordinator has no neighbour in that region. To counter this situation, the routing mechanism differentiates N3 as a trespassing neighbour for Region 1. So whenever there is a message destined to some region the algorithm checks to see if there is a trespasser for that region and if so, is it the closest coordinator to the message’s destination.
3.9 **Transparent Borders**

The routing mechanism is used most extensively when a new player logs in. The joining player contacts the central server to join the game. At this point the server can refuse the player's entry into the game based on the company's admission control policy. Each player starts off with some starting position, which lies within some coordinator's zone. The server generates a register message for the player with its starting position as the destination of the message and injects the message into the overlay. The message ultimately ends up at the coordinator that oversees the starting position of the player.

The coordinator decides, based on the information in the message, whether the player's publish/subscribe areas encroach into the zones of neighbouring coordinators. It generates a list including all such coordinators and itself. This list is sent to the joining player asking it to register with all the coordinators in the list.

Players forward their actions to all the coordinators they are registered with so that their movements can be tracked. This makes the boundaries fairly transparent. In all other zoning techniques proposed so far the perception of the players is limited by the boundaries of the zone in which its origin lies. In these schemes two nearby players separated by the artificial boundary are oblivious to each other's presence.

![Diagram](image)

*Figure 3-2 Transparent Boundaries*
However, in our scheme these boundaries are made completely transparent through a combination of the publish/subscribe areas and the requirement for a player to register with all coordinators whose space the player’s publish/subscribe areas overlap.

In Figure 3-2, since B’s AoI encroaches into C1’s zone, B sends position updates to C1 as well, even though B itself resides in C0. Based on this information C1 can track B’s location and informs A when B’s subscribe area starts overlapping its own publish area.

### 3.10 Splitting & Merging

A zone is partitioned in half with the split occurring along a line running through the center either horizontally or vertically. When the height and width of a zone are equal it is split vertically resulting in two rectangular zones. If the height and width are unequal the zone is split horizontally. In this way, a coordinator that is repeatedly split alternates between rectangular and square zones. This results in more regularly shaped regions.

Merging is slightly more complicated. The relationship between coordinators can be classified as parent-child. (The parent being the coordinator that split and handed over half of its space to the child coordinator.) Each coordinator stores the ID of its parent and a list of its children in the order in which they are ‘created’. Whenever a coordinator becomes under utilized it sends a merge request to its parent if it does not have any children. Otherwise, it has to send a merge request to the last created child. A child can turn down the request under two circumstances (i) it is overloaded and merging will, in all likelihood, only result in an even more overloaded coordinator (ii) the child has other children. If this is the case, the parent has to wait for its child to become available. Only
once a node has recombined with all of its children in reverse order of creation can it initiate a merge with its parent or accept a parent’s invitation to join.

The architecture is flexible enough to support more complicated splits. For example, instead of just splitting in half, the vertical or horizontal line could be drawn away from the center and the direction of the split could be altered to equally split the number of players. Alternatively, game semantics can also be involved in the decision to coincide the zone boundaries along natural obstructions in the game like walls or mountain ranges.

3.11 Dynamic Area of Interest (AoI)

As the zone split and become progressively smaller to handle higher player densities it becomes increasingly likely that a player’s AoI will overlap multiple regions. To prevent this from happening the AoI of a player is resized proportionally to the dimensions of the zone in which it lies. As a result, a player passing through small zones will have its AoI decreased. This scheme is an acceptable adaptation of the cocktail party effect [2] already well established in interest management systems. Even in real life a person in a crowded space tends to only concentrate on listening to few close-by people and blanks out the background noise. Since smaller zones mean higher number of players in the region, one can get the same effect as above by simply reducing the AoI on entering smaller regions.
3.12 Overload Detection

All along, the system is under pressure to perform its task against hard time constraints so that the latency and responsiveness demands of the application are adequately met. We use this to decide when a coordinator needs to shed some of its workload. The coordinator times how long it takes to perform state computations and dissemination of the updates. If it sees that it is unable to process the state before the next state is due, it deduces that it is overloaded. The processing time also serves as a weak indicator of the load the node is putting on its network connection. As overloading the network forces the network to drop packets some of which have to be retransmitted by the coordinator resulting in higher processing time.

3.13 Topology Aware Overlay Construction

While choosing a child for an overloaded parent the central server considers amongst other factors the network proximity of the child and the parent. Choosing a child which is closest in network proximity to the parent results in lower latency of message exchanges, leading to better consistency and lower aggregate network consumption. Since neighbouring coordinators need to exchange large amounts of data, this can have a considerable effect on performance. Other P2P systems that we have studied do not assign any importance to the underlying network topology. Knutsson et al. [2] for instance uses Pastry, which separates neighbouring coordinators by many overlay hops over the overlay and ignores network topology in deciding the nodes to interconnect to form the overlay. Given the amount of traffic and the latency sensitive nature of the application this is completely unacceptable.
In our scheme, replicas and neighbours are selected to be in close network proximity. To gauge the network proximity we use the idea of network coordinates proposed by [10]. Basically, each node is assigned network coordinates on the basis of ping times to certain landmarks on the network. Two nodes are considered to be close when the Euclidean distance between their network coordinates is small. By selecting children optimally in network terms we make the entire overlay topologically aware and efficient.

3.14 Replica Replacement

This is a fairly simple procedure since the replicas keep a record of their original coordinator's neighbours, constituent players and object states. Based on this information they inform the neighbour coordinators and the players that it is taking over the coordination responsibilities. The players update the new coordinator with any events that it might have missed because the original coordinator died before passing them to the replicas. As there is now one fewer replica, the promoted replica asks the central server to nominate another player node to serve as a replica for that zone.
Chapter 4

Overlay Network Management

This chapter describes a completely distributed algorithm for maintaining the topology of the overlay of coordinators that can dynamically add or remove nodes depending on the load distribution. Using this topology the coordinators maintain global connectivity to the entire game while being aware of only their immediate neighbours.

4.1 Routing Table

Coordinators maintain information about all of its neighbouring zones in a routing table. This information consists of the zone’s origin, dimensions and the IP and port numbers of its coordinator. Whenever a zone’s dimensions are changed only its immediate neighbours need to be notified of the change to maintain a consistent view of how the overlay space is divided amongst the coordinators.

The routing table arranges the neighbours in four sets depending on their location relative to the coordinator’s own zone. These sets correspond to the regions defined
around a zone by superimposing the grid, (described in Section 3.8). A neighbour is inserted into a set if its origin lies in the corresponding region. Within a set, neighbours are sorted by their origin. If the set runs along the y axis (Sets 1 and 3), the neighbours are sorted by the y coordinate of their origin. Otherwise, they are sorted by the x coordinate (Sets 0 and 2).

Arranging neighbours in regions simplifies splitting and merging as the neighbours are arranged in a definite order. Coordinators exploit this order to decide how and when to update neighbours of changes in their zones. This arrangement of neighbours also has the following additional advantages, which are elaborated upon in the indicated sections:

- While forwarding packets a node has to search for the next hop through a smaller set of nodes
- It prevents routing loop (Section 5.2)
- During message multicast it prevents duplicate messages (Appendix B.1).

### 4.2 Handling Splits

The current implementation divides overloaded coordinators in half by splitting zones horizontally or vertically. Both kinds of splits are essentially the same. During a split a node (parent) reduces its chosen dimension to half and assigns the remaining half to another machine, which is referred to as the child. Once the child has been bootstrapped with information about its inherited neighbours, the parent informs its neighbours about the new zone so that they can update their routing tables. The type of update depends on the set of the neighbour. This section discusses the process for a vertical split. The process can be extended by symmetry to horizontal splits.
For a vertical split as in the case of node A shown in Figure 4-1, nodes in:

**Set 3:** need only to be informed about the reduced width of A’s zone. They are not informed about A’s new child B as A separates them from B. Node A sends a CAN_announceReplace (please see Table 6-1 for details of this and other message types) to these neighbours asking its entry to be replaced with almost the same information as the original entry except the chosen dimension which is halved.

![Figure 4-1 Vertical Split](image)

**Set 1:** will no longer abut A after the split. So Node A sends all its neighbours in the set a CAN_announceReplace message to have all entries referring to A replaced by B.

**Sets 0 & 2:** These sets run perpendicular to the direction of the split and updating them is not as straight forward as in the first two cases. The neighbours in these two sets can further be classified into four categories:
NoChange:

These are neighbours like node 1 which lie completely on A's side of the boundary separating A from B. The NoChange nodes only have to be informed about the reduced dimensions of A by using a CAN_announceReplace message.

BoundaryMatchers:

These are neighbours whose boundary neatly coincides with the line splitting A and B. Neighbours 6 and 7 fall in this category for the split shown in Figure 4-1. A sends them a special message, CAN_announceSplit, telling them to reduce the dimensions of A in their routing records and inserting a new neighbour B in their neighbour list. A and B might be recorded in different neighbour sets of a Boundarymatcher.

BoundaryStradlers:

Very similar to boundary matcher, these nodes also lie in sets running perpendicular to the direction of the split. The difference is that the zones of BoundaryStradlers extend on both sides of the split. These nodes are also sent a CAN_announceSplit message by A but in their case both A and B end up in the same neighbour set of the BoundaryStradle. Neighbour 2 in the diagram is an example of a BoundaryStradler.

ReplaceNeighbor:

These neighbour lie completely on B's side of the split and are required to replace A by B in their records. Neighbours like node 3 which fall in this category receive a CAN_announceReplace message from the splitting node. These nodes replace the parent by its child with half of its parent's zone.
As a last step, after having informed all of its neighbours and bootstrapping a child the splitting node updates its own routing information to reflect the split. Set 3 remains intact with no changes. The neighbour sets 0 and 2 are trimmed to include only NoChange and BoundaryStradlers. Set 1 is reset to include the child and any BoundaryMatchers, like 6. The BoundaryMatcher has to be moved from set 2 to set 1 because it lies in Region 1 of the resized parent node as shown in the following diagram.

![Figure 4-2 Post Split Configuration](image)

For horizontal splits BoundaryMatchers from set 3 are moved to the new set 2 as in the diagram (Figure 4-3):
9 moved from Region 3 to 2

Figure 4-4 BoundaryMatcher (Region 3) - Horizontal Split
Presence of a trespasser\(^1\) complicates splitting to some extent. Figure 4-5 shows a node A with a trespasser node 3 from set 0 going over into region 1. When node A splits horizontally it sends a CAN_announceSplit message to node 1 even though it lies in set 0 which is an exception to the update rules defined above. (According to the update rules only CAN_announceReplace messages can be sent to nodes in sets running parallel to the split.) So we introduce the following exceptions in the update rules to accommodate trespassers:

- In case of a horizontal split send a set 0 trespasser a CAN_announceSplit message if the trespasser’s lower limits extend below or matches the parent-child boundary.

- For vertical splits, a CAN_announceSplit message is sent to a trespasser from set 3 if it overlaps with the child.

---

\(^1\)Trespasser is a neighbour whose zone extends beyond the region of its origin. Please see Section 193.8 for more information on trespassers.
In the case of a horizontal split roles of the set are exchanged. Sets 0 and 2 get CAN_announceReplace messages while neighbours in sets 1 and 3 get CAN_announceReplace or CAN_announceSplit messages depending on what sub-category of neighbours they fall in.

4.3 Intelligent Split

More efficient load balancing can be achieved by dynamically selecting the orientation and location of the split to cut the number of players exactly in half. Also, to reduce the communication cost of moving state from the parent to the child the boundary between the parent and the child can be initially drawn closer to one side and then slowly moved towards the center transferring the state of the objects over a period of time as objects fall over into the child's half. In this way, a parent does not have to move half of its entire state at once to the child. The boundary can also be made fluid by allow it to move in either direction to balance out the workload on the coordinators. The neighbours can be informed of the corresponding changes by a combination of CAN_announceSplit and CAN_announceReplace messages.

4.4 Bootstrapping the child

A parent has to inform its child of the neighbours it will inherit after a split. Using the example of a vertical split of node A from Figure 4-1, node A sends to node B the following neighbours in a NeighborList message:

• All of its set 2 neighbours.
• All neighbours in set 1 and 3 which lie on B’s side of the boundary, BoundaryMatchers and BoundaryStradlers.

• Any trespasser from 0 if it exists and its region crosses the boundary between A and B.

The child is bootstrapped before informing A’s neighbours of the split to ensure that the child does not start receiving overlay messages before it has a complete list of its neighbours.

4.5 Merge Algorithm

When two nodes recombine they send a replace message to all neighbours around them except in the direction in which they combine. The parent advertises itself with the bigger combined sized using CAN_announceReplace messages; whereas the child informs its neighbours to replace it with its parent in their routing tables. The child’s and the parent’s sets that lies along the boundary of the merge are treated differently from other sets as one of the neighbours in the set is always a participant in the merge. In addition to that participant, there can at most be one other node that meets the node at a corner. These are shown below for both horizontal and vertical merges.

![Figure 4-6 Merge - CAN_announceReplace messages](image)
There is guaranteed to be at most only one such neighbour because the parent and the child have the same dimension along the edge that is merged and there is space left for a node to border the node only at the corner.

4.6 Locking

Before merging/splitting nodes ensure that none of their neighbours are also in the process of changing their boundaries. If two neighbours were allowed to split or merge at the same time it would yield inconsistent topologies as in the case of the following diagram:

The exact final configuration depends on whether the parent receives the CAN_announceReplace before or after sending the neighborList to the child. Similar, inconsistencies can arise when two pairs of neighbours merge or one pair merges while

Figure 4-7 Inconsistencies caused without Locks
the other splits at the same time. To counter these situations nodes send a special message
to lock their neighbours before initiating a merge or a split.

The Coordinator, as explained in Chapter 6, has sub-components called the Overlay
Manager (OM) and the Lock Manager (LM). The Overlay Manager is responsible for
maintaining routing table, connections with neighbouring coordinators and routing
messages over the overlay. It also handles the splitting and merging of its own zone and
keeping its routing table updated when its neighbouring zones change. The Overlay
Manager requests a lock from the Lock Manager before initiating a split or a merge. The
Lock Managers working in tandem with Lock Managers in the neighbouring coordinators
ensure that only one split/merge is active at a time in a ‘neighbourhood’. The Lock
Manager utilizes the set of variables shown in the table to ensure this property.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locksGranted</td>
<td>Number of LMs granted a lock by this LM</td>
</tr>
<tr>
<td>requestsSent</td>
<td>Number of pending lock requests sent by this LM</td>
</tr>
<tr>
<td>requestNotSent</td>
<td>Flag that is set when a lock request could not be sent to neighbours</td>
</tr>
<tr>
<td></td>
<td>because the requesting LM has already granted the lock to some other LM</td>
</tr>
<tr>
<td>requestTime</td>
<td>A Lamport timestamp assigned to a lock request to unambiguously establish</td>
</tr>
<tr>
<td></td>
<td>its priority</td>
</tr>
<tr>
<td>requestLock</td>
<td>Message sent to neighbouring LMs to request a lock</td>
</tr>
<tr>
<td>lockGranted</td>
<td>Message sent by an LM to grant a lock to a requesting LM</td>
</tr>
<tr>
<td>blockedNeighbors</td>
<td>List of neighbouring LMs that have requested a lock but have not been</td>
</tr>
<tr>
<td></td>
<td>granted a lock</td>
</tr>
<tr>
<td>grantedBy</td>
<td>List of LMs that have granted this LM a lock</td>
</tr>
<tr>
<td>grantedNeighbors</td>
<td>List of LMs that have been granted a lock by this LM</td>
</tr>
</tbody>
</table>

Table 4-1 Symbols used in the Locking Algorithm
Each Lock Manager maintains two variables locksGranted and requestsSent. LocksGranted corresponds to the number of nodes that have been granted the lock by the node while requestsSents counts the neighbours that have been asked for a lock. To acquire a lock a LM sends a requestLock message to its neighbours incrementing requestsSent once for each node requested. A node can send requestLock only if it has not already granted the lock to another node (i.e. whenever locksGranted is zero). If locksGranted is non-zero the Lock Manager sets requestNotSent and waits for all granted nodes to release their locks before sending its own request for a lock.

On receiving a lock request, a node will block the requester if the requested node currently holds the lock or itself has a pending request with a higher precedence. The precedence of requests is found by first ordering them in decreasing order of requestTimes and then by the requestor's ID. Ids of requestors that are declined a lock are saved by the node in a list of blockedNeighbors. (A node declines a request by not responding to it.) In all other circumstances, a node accepts the lock requests and responds by sending a lockGranted message back to the requestor and incrementing its locksGranted counter. It also adds the granted node's Id to grantedNeighbors list.

Every time a Lock Manager receives a lockGranted message it decrements the requestsSent counter. When the counter hits zero it means that all the requested nodes have granted it the lock and the Lock Manager can invoke the call-back function registered by the Overlay Manager. Again, a node records Ids of all nodes that have granted it the lock in a list named grantedBy. When the Lock Manager's releaseLock() method is called by the OM, the LM sends unlock messages to all neighbours in grantedBy and follows that by sending lockGranted messages to neighbours in the blockedNeighbors list. These are nodes that had requested a lock while the node was
acquiring or using the lock. The nodes from the blockedNeighbors list are moved to the grantedNeighbors list and the requestsGranted is incremented once for each of them.

The unlock message can trigger LMs that had earlier suppressed their lock requests because of non-zero requestsGranted counters at the time of the earlier attempt to restart their suspended lock requests. These nodes will have their requestNotSent set to true by the first call to requestLock function. When nodes with requestNotSent flag set receive an unlock message they make an other attempt to acquire the lock. To prevent starvation the time at the instant of the original (first) requestLock function call is reused as the requestTime.

4.6.1 Locks and Splits

An Overlay Manager detecting an overload sets a flag splitInProgress, requests a lock and registers its processLock() function with the Lock Manager as a call-back. SplitInProgress is used to prevent an OM for making multiple split calls while it waits to receive a lock. The Lock Manager invokes OM’s processLock() on securing the lock. processLock() sees that splitInProgress is set so it concurs that the lock was requested for a split and initiates the aforementioned split algorithm but only if the node is still overloaded. Otherwise, it just lets the lock lapse. Once the split is complete, the node resets splitInProgress and gives up the lock by calling releaseLock().

4.6.2 Locks and Merges

Merging is a bit more involved as two entities- the parent and the child both have to acquire locks from their neighbours before proceeding with the merge. There are some
situations in which this can result in loops of lock request which cause deadlocks as shown in the diagram below:

![Diagram showing Merges and Deadlocks]

Figure 4-8 Merges and Deadlocks

The diagram (Figure 4-8) shows a case where P and C have agreed to merge but node 1 is also in the process of changing its zone and has acquired the lock from C and is waiting for the lock from P. P will not grant 1 the lock until it has merged with C which cannot happen as long as there are outstanding locks granted by C. This results in a deadlock as there is a cycle of dependencies.

In the more complete algorithm a parent and a child decide to merge by exchanging CANMerge and CANMergeConfirmation messages. Once they have decided to merge the parent locks all of its neighbours and sends the child a list of its locked neighbours. The child, in turn, locks all its neighbours not in the list received from the parent and returns to the parent a special message neighborList containing a list of all its neighbours. Once all neighbours have been locked the parent sends CAN_announceReplace messages to the child’s and its own neighbours to announce the merge followed by unlock
messages to all the neighbours locked by itself and the child. The parent unlocks the child's neighbours to protect against the following situation:

![Diagram showing the process](image)

**Figure 4-9 Why parent should unlock neighbors locked by child**

The numbers in Figure 4-9 at the start of an edge show when the message was sent and the number at the end (close to the arrowhead) shows in what sequence it was received. In the above case the neighborList from the child is delayed and is received at time 10 after the split update from node N. When the neighborList finally arrives from the child, it overwrites a more recent update of node N's dimensions. By making the parent unlock its child's neighbour, we ensure that node N is not allowed to split until the parent has received the neighborList from the child.
Chapter 5
Overlay Routing

In this chapter we present detailed algorithms for routing messages over the overlay. We begin by presenting an algorithm inspired by CAN. In the subsequent sections, we introduce incremental improvements to the algorithm which finally yield an algorithm with logarithmic routing cost.

5.1 Basic Routing Algorithm

Irrespective of the routing algorithm, points on zone boundaries have to be assigned deterministically to at most one of the neighbouring coordinators sharing the boundary to unambiguously route messages. A coordinator owns all points on the top and left boundaries of the zone starting from the top left corner to (but not including) top right and bottom left corners, respectively. It does not own any point on the right and the bottom boundaries.
In this manner all boundaries are allocated to only one coordinator. Messages are routed through the overlay by including the coordinates of the destination in the message header. Nodes along the route of a message decide the region for the message’s destination relative to themselves. The following algorithm returns a unique region for a message at a particular node:

```c
Region(mx,my)  
{  
// CNode is the current Coordinator  
if (mx<CNode.x && my< CNode.y+ CNode.height) 
    return 3;  
if(mx>= CNode.x && my< CNode.y) 
    return 0;  
if(mx>= CNode.x+ CNode.width && my>= CNode.y) 
    return 1;  
if(mx< CNode.x+ CNode.width && my> CNode.y + CNode.height) 
    return 2;  
else    
    return -1;  
}
```

Figure 5-2 Region Calculation

Note how the function utilizes the assignment of boundaries to determine the region for message destinations.
As described in Section 4.1 the routing table is divided into four sets. Based on the value returned by the above function a node looks for the next hop in the corresponding routing table set. Within the set it picks the coordinator whose origin is closest in Euclidean space to the destination of the message. If the destination lies within the overlay coordinator’s own space it passes the message on to the application dependent game middleware coordinator. This algorithm is what we refer to as the basic CAN routing algorithm in the thesis.

At first blush the top left corner of a coordinator’s zones appears to be a reasonable choice for the coordinator’s origin. But this choice of the origin results in non-optimal number of routing hops. For instance, in Figure 5-4 the message will be routed through A, B, and C because at each step their origin is closest to the destination. However, had D been picked by S as the next hop, it would have resulted in much fewer hops. So it appears that the dimensions of zones should also be considered while picking the next hop. Priority should be given to a node that will help the message move the farthest distance towards the destination with the minimum number of overlay hops.
To take advantage of bigger zones it's better to use the center of the coordinator's zone as its origin. We refer to this scheme as 'Center Origin' in this thesis. The scheme incorporates into the origin's value a measure of the dimensions of the coordinator's zone yielding fewer hops. The evaluation sections provide empirical evidence for supporting this choice.

5.2 Grid

The existence of the four sets of neighbours/regions plays a very subtle but important role in routing as well. If the nodes were not split into sets (i) search for the next hop would take longer as the search space would be fours times as big; (ii) more crucially it also results in routing loops as depicted in Figure 5-5:
Without the grid A and B would repeatedly exchange the message between themselves because both their origins are closer to the destination compared to C or D. When the grid is laid on top of A to split the space into regions, A is forced to pick a neighbour in the same region as the destination which in the Figure 5-5 would be neighbour C. In other words, the grid forces the message to always move in the direction of its final destination.

5.3 Big Hop

The routing algorithm as is requires \( N^{0.5} \) number of hops in the worst case where \( N \) is the number of nodes connected in the overlay. To allow even better performance each node maintains links called BigHops to its parent and its first two children. The BigHop links inter-connect all the overlay nodes in the form of a tree. Links to only two children are kept to limit the number connections per node. The first two children are selected as they are always the farthest children of a node and maintaining connections with them allows parents a means to forward a message a long distance in fewer hops towards its destination in the overlay space. To route over the BigHop tree the node has to compare distances of its parent and children to the destination. The algorithm uses the top left corner of a zone as its origin for this purpose. The top left corner of a zone always remains the same regardless of the number of times it splits and merges. Any other choice of origin would have required the coordinator changing its zone dimensions to send an update to its parent and children. This not only requires extra communication but is also impossible to accomplish when parents with more than two children split or merge. As explained above, a parent does not maintain links to its children that are created after the first two splits. So it has no way of communicating changing values for its origin.
While routing a message a node checks if any of the nodes in the tree are closer to
the destination compared to its best choice neighbour and if so uses the branch in a tree to
make what is called a "BigHop". This allows messages that have to travel long distances
to take fewer larger hops to get to their destination.

This algorithm just costs three additional links at the most for every node. It also
offers the possibility of a message jumping sub-trees using the normal neighbour to
neighbour links without reaching the common ancestors of the two trees

Without BigHop links the message from the source to destination in Figure 5-6
would have to pass through all the small zones that lie between these two points.
However, with BigHops node A will see that its child B is closer to destination then any
of its neighbours so it will forward the message to B from where it takes only a single
hop to get to its final destination. The message would take the same route if the roles of
the source and destination were reversed.

Figure 5-6 Relative Location and BigHop Links
The depicted space could possibly be divided further. However, the diagram only shows the nodes, links and zones that are important for the sake of this example. The dashed links represent unidirectional connections from a child to a parent. The parent in these cases does not have a direct BigHop link to the child because it already had two other children by the time this child was created. The dotted links represent points where other subtrees can be grafted if the zones where split further.

When C receives a message whose final destination is a point in J’s space it can send it to either its parent in the BigHop tree, node B, or it can send the message to one of its neighbours. If the neighbour is closer than the parent, it will chose the neighbour effectively jumping sub-trees by using one of its neighbour links.

We use the top left corner of the zone as the coordinate to measure distance from while routing within the BigHop tree.

![Diagram of hopping subtrees](image)

Figure 5-7 Hopping Subtrees

In Figure 5-7 since N is closer to DST then S's parent, S will forward the message by using the neighborLink. N will in turn use its neighborLink to forward the message to A which is in the same subtree as DST.
5.3.1 Loop Avoidance Rule

In the case of both parent and child being closer to the destination from each other than their closest neighbour they would indefinitely pass the message between themselves. One way to stop a child from sending a message that has already been to its parent is to add a field to signify that the packet has reached the current node using a link from its parent. This field is set to false whenever a non-child BigHop link is used. But this will not work when a packet is routed to another node before making its way back to the child. Since the parent field would have been cleared by the time packet gets back to the child it has no way of knowing that the packet has already been to its parent. As a result, the message will indefinitely loop between the parent and the child through possibly other intermediary nodes. To prevent this, a node only takes a BigHop if doing so will take the message closer to the destination; otherwise it is forced to pick the closest neighbouring node.

Without Loop Avoidance Loop Rule the message will loop between P and C

Figure 5-8 Loop Avoidance Rule
5.4 Lies Within

A node cannot blindly send a message to the closest node in the same region as the destination. If it did so it would be stuck in a loop as in the case depicted in Figure 5-9, as the origins of both A and B are closer to the destination than the owner of the destination's zone, C. This peculiarity can only arise between coordinators of zones that border the zone in which the destination lies. It is easily avoided by checking if the point lies in one of the neighbour's zone before forwarding it to the closest coordinator.

<table>
<thead>
<tr>
<th>Node</th>
<th>DST in Region R</th>
<th>Neighbors in Region R</th>
<th>Closest Neighbor in Region R</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>C,B</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>C,A</td>
<td>A</td>
</tr>
</tbody>
</table>

Figure 5-9 Lies Within
5.5 Reverse BigHops

![Figure 5-10 Subspaces not connected by BigHop]

Improvements through BigHop routing are significantly diminished when messages are routed in certain directions e.g. when a message is routed from region A to B in Figure 5-10 it will take a number of small hops through immediate neighbours until it reaches region B from where the power of BigHop routing can be fully realized. This happens because of two reasons; (i) coordinators in the two regions are not related by an ancestor-descendant relationship, hence there are no BigHop interconnections between these two regions (ii) the choice of the top left corner as the origin does not make going through the available links available in Primary Tree² attractive enough for the greedy routing algorithm as the message has to be temporarily routed away from the final destination. Figure 5-11 shows an example where S can possibly route a message to DST through three BigHops by choosing a locally non-optimal first step. However, this is disallowed by the greedy routing algorithm.

Both these problems can be solved by overlaying an inverted tree consisting of Reverse BigHop links with its root at the bottom right corner of the overlay space. This Secondary Tree (or Reverse Tree) uses the bottom right corners of the zones it stitches together as their origin in contrast to the top left corner as in the case of the Primary or

² See Table 5-1 for a description of terms used in this section.
BigHop tree. Such a tree does not only relate regions A and B of Figure 5-10 but also allows the greedy algorithm to take longer hops in the opposite direction of BigHop Links.

The Primary coordinator for a zone cannot act as the coordinator in the secondary tree because when its zone splits it would lose authority over the bottom right corner to its new child. This would necessitate informing all children of the coordinator in the secondary tree to point their reverse links to the new child coordinator. However, as a parent only records links to its first two children, it would not be able to update any other children that it might have. Therefore, each zone is assigned a secondary coordinator whose origin lies in the bottom right corner of the zone. When a region does split the secondary coordinator need not be moved. The splitting primary coordinator is assigned a new Primary Tree Partner (PTPartner) for its smaller zone. And the new primary tree coordinator takes over the remaining half with a ‘preinstalled’ secondary coordinator. The new PTPartner of the parent is grafted into the secondary tree as a child of its former PTPartner. Figure 5-12 shows the reverse tree added to Figure 5-11.

![Figure 5-11 Available BigHop Links](image-url)
5.5.1 Reverse Tree Construction

The reverse tree is constructed using the following algorithm:

- When there is only one zone the same MDC acts as the coordinator in both trees.
- On split:
  - the child becomes a secondary coordinator for the parent's half of the zone
  - while the parent's original secondary coordinator is inherited by the child
  - the reverse BigHop links are established between the new and the old secondary tree coordinators as needed

The above algorithm abstracts away some of the important details. A splitting coordinator sends its PTPartner information about its newest child so that PTPartner can associate itself with the new coordinator. The child inherits the parent's old PTPartner and sets itself as the PTPartner of its parent in the primary tree. The child is also added as a secondary coordinator for its parent's new smaller zone. The new secondary coordinator becomes the child of parent's original (pre-split) PTPartner. The salient
feature of the scheme is that all changes in the Primary and Secondary tree remain local to the coordinators directly involved in the split. Other, coordinators need not be informed of the change.

![Diagram of tree construction](image.png)

**Figure 5-13 Secondary Tree Construction**

### 5.5.2 Routing with Reverse BigHops

The complete routing algorithm which incorporates reverse BigHops is given below. A node receiving a message needs to first check if it received the message because of its role in the primary or secondary tree. If the destination lies within its secondary zone or its secondary origin is closer than all BigHop, Reverse BigHop and ordinary neighbours then it forwards the message to its Secondary Tree Partner (STPartner). If the destination does not lie in the STPartner’s zone it can, in turn, send the message using any of its three types of links. If the STPartner chooses one of its Reverse BigHop links the message will continue its journey on the inverted tree.

The secondary coordinator is essentially a form of indirection. It forwards incoming messages to its associated STPartner which has more accurate information of
how its neighbours are arranged. The STPartner can, if required, send a message along the reverse BigHop to another secondary coordinator.

<table>
<thead>
<tr>
<th><strong>Term</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary or BigHop Tree</td>
<td>Tree connecting Primary Coordinator with BigHop Links</td>
</tr>
<tr>
<td>Secondary or Reverse Tree</td>
<td>Tree connecting Secondary Coordinators with Reverse BigHop Links</td>
</tr>
<tr>
<td>Primary Coordinator</td>
<td>Owner of a zone with its origin located at the top left corner of the zone</td>
</tr>
<tr>
<td>Secondary Coordinator</td>
<td>Relays messages on the Reverse Tree to its STPartner and is located at the bottom right corner of its zone</td>
</tr>
<tr>
<td>PTPartner</td>
<td>Is for a Primary coordinator the secondary coordinator assigned to its zone</td>
</tr>
<tr>
<td>STPartner</td>
<td>Is for a Secondary coordinator the Primary coordinator assigned to its zone</td>
</tr>
</tbody>
</table>

Table 5-1 Definitions used in Reverse Description

The figure on the next page shows the complete routing algorithm. It makes a locally optimal choice to pick a message’s next hop from ordinary neighbour-to-neighbour, BigHop and ReverseBigHop links to reach the destination in logarithmic number of hops. The experimental proof for this improvement is furnished in Chapter 7.
Route(mssg){
    minDist=curDist=distance(CNode.origin, mssg.Dst);
    ST Hop=T2Closer=false;
    ST Region=ST.region(mssg.Dst);
    BigHopIndex=-1;
    if(STRegion==-1){ // mssg.Dst lies in STPartner
        ST.Send(mssg);
        return;
    }
    if(ST.distance(mssg.Dst)<minDist){ // mssgDst closer to the STOrigin than PTOrigin
        minDist=ST.distance(mssg.Dst);
    }
    if(PTbigHop.distance(mssg.Dst)<minDist){ //is the Dst Closer to Primary Tree bigHops < minDst
        minDist=PTbigHop.distance(mssg.Dst);
        bigHopIndex=PTbigHop.closest(mssg.Dst);
    }
    if(PTPbigHop.distance(mssg.Dst)<minDist){ //is the Dst closer to Primary Tree Partner’s bigHops < minDst
        ST Hop=true;
        BigHopIndex=PTPbigHop.closest(mssg.Dst);
        minDist=PTPbigHop.distance(mssg.Dst);
    }
    if(Neighbors[region].distance<minDist || curDist==minDist){ // is one of the immediate neighbors or the current coordinator itself
        Neighbors[region].sendToClosestNeighbor(mssg);
        return;
    }
    if(bigHopIndex==-1){ //send to Secondary tree partner as it is closer than BigHops, // ReverseBigHop and ordinary Neighbors
        ST.Send(mssg);
        return;
    }
    if(STHop) { // use reverse BigHop
        PTPbigHop[BigHopIndex].send(mssg);
    } else // using BigHop
        PTPbigHop[BigHopIndex].send(mssg);
}
Chapter 6

Architecture

In this chapter, we elaborate upon Proximity's software architecture by presenting the main components of the game middleware, namely:

- PlayerMD
- Game Console
- Zone Coordinator (Game Middleware & Overlay Coordinator)
- Central Server

6.1 Player Code

The code running on the player machine consists of a game console that sits on top of the middleware, PlayerMD. The console is mostly game specific and is responsible for gathering user input, simulating the effects of user actions and visualization of the virtual world. The game console passes all the events generated by a player to the PlayerMD,
which runs underneath it on each player’s computer and is the representation of the player in the gaming middleware. The PlayerMDs maintain connections with all the players whose area of interest overlaps with its player’s publish area. These players are known as a player’s subscribers. Every time it receives an event from the game console the PlayerMD forwards the event to all of its subscribers and also the zone coordinators with whom it is registered (registered coordinators). PlayerMDs periodically remove subscribers and registered coordinators that no longer overlap with its AoI. It adds new coordinators and subscribers to its lists on being informed by a coordinator of a new overlap. Whenever, a PlayerMD receives an event from another player it updates the player’s location record and passes the event on to the game console.

The game console and the PlayerMD exchange TCP messages on the loopback address. The game console starts the PlayerMD as a separate process and passes to it as arguments the port number for its socket and its player’s starting coordinates. The PlayerMD connects to the passed port number. This connection is used to exchange messages between the game console and the PlayerMD. Having a socket based interface allows the Middleware to be used with Game Consoles written in any other language. The small size of the message set allows even existing games to be easily modified to work with the proposed architecture.

On start up the PlayerMD sends a register message to the central server. The central server runs on the game publisher’s computer and is used for admission control for the game. The server also keeps a record of all the players currently playing the game. It essentially works as a rendezvous for the peer-to-peer overlay of the coordinators. On receiving a register message from the PlayerMD, the central server injects the register message into the overlay with the player’s starting location as the destination of the
message. The coordinator responsible for the zone covering that part of the map becomes
the primary coordinator for the player. The Primary Coordinator returns the dimensions
of the zone, IP and Ports for itself and any other (Secondary) coordinators whose zones
the player might infringe. The PlayerMD adds the Primary Coordinator to its list of
registered coordinators and subsequently adds all the Secondary Coordinators after
registering with them. Registering with a Secondary Coordinator might reveal more
Secondary Coordinators that the PlayerMD will have to register with. It is the PlayerMDs
responsibility to broadcast the events generated by its Game Console to all its registered
coordinators.

![Figure 6-1 MDC Subcomponents](image-url)
6.2 Coordinator

The coordinators are composed of the subcomponents shown in Figure 6-1. Unlike the PlayerMD-console interface, the coordinator interfaces are more conventional function based interfaces. The Player Manager is the only application specific component while all other components will exist in one form or another for any applications based on the overlay network. The Player Manager performs the game specific activities like registering players, keeping tabs on their location and informing them of overlaps. It utilizes the services of the Overlay Manager which is responsible for maintaining the overlay topology, routing and communicating any changes to its zone to the Overlay Managers of the neighbouring coordinators. Before initiating a split/merge the Overlay Manager has to request a lock from the Lock Manager. The Lock Manager works with neighbouring Lock Managers to ensure that only one Overlay Manager is modifying its zone in the neighbourhood at a time. The network Manager delivers (/receives) messages for various subcomponents to (/from) other coordinators while hiding the networking details. MDC (subcomponent) determines the control flow for the entire coordinator. It periodically issues an Epoch() event to the Player Manager so that it can update certain time dependent information e.g. checking the status of a player that has been inactive for some time. Between 'epochs' the coordinator waits for messages to be delivered to the Network Manager which calls the appropriate functions in the subcomponents based on the type of the incoming messages. The Player Manager following a game specific logic signals to the Overlay Manager when to split or merge. During testing and evaluation we used number of registered players and bandwidth usage for deciding when to merge. However, Section 3.12 discusses a better more practical metric.
6.3 Central Server

In the current implementation the central server helps coordinators locate close-by player machines that do not already have a coordinator running on them. In order to do this the central server has to keep separate lists for PlayerMDs running and not running coordinators. Whenever a child coordinator merges with the parent it informs the central server that its PlayerMD has become available again to spawn another coordinator.

Figure 6-2 Deployment Diagram
6.4 Inter-Component communication

The central server has a single port on which it waits for messages from both coordinators and PlayerMDs. While coordinators have three different ports (i) for communication with registeredPlayers (playerport) (ii) for overlay maintenance and routing (overlayPort) (iii) and for communication between neighboring middleware coordinators (coordinatorPort). A PlayerMD has two other ports in addition to the port used for communication with game console. One is used for inter-player communication while the second port is used to communicate with the coordinating infrastructure which includes both the central server and the coordinators.

The following table shows the different message types used by the middleware.

<table>
<thead>
<tr>
<th>Message</th>
<th>Purpose</th>
<th>Information Contained</th>
<th>From</th>
<th>To</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_announceSplit</td>
<td>Overlay Maintenance</td>
<td>Own and new coordinator’s IP, port and new dimensions</td>
<td>Splitting Coordinator</td>
<td>Neighboring Coordinators that meet the requirements</td>
<td>Update routing table</td>
</tr>
<tr>
<td>CAN_announceReplace</td>
<td>Overlay Maintenance</td>
<td>Own and new coordinator’s IP, port and new dimensions</td>
<td>Splitting and merging coordinator</td>
<td>Neighboring coordinators</td>
<td>Update routing table</td>
</tr>
<tr>
<td>CAN_NeighbourList</td>
<td>Overlay Maintenance</td>
<td>Information about neighbors inherited by a new Coordinator</td>
<td>Splitting parent to child to parent</td>
<td>Newly Spawned Coordinator</td>
<td>Build routing table</td>
</tr>
<tr>
<td>CAN_Route</td>
<td>Overlay Routing</td>
<td>Message to be routed through the Overlay</td>
<td>Created by the source of the message</td>
<td>Passed through the Overlay until it reaches the destination</td>
<td>Destination: Extract payload and deliver to application Intermediate: Find and pass to next hop</td>
</tr>
<tr>
<td>CAN_Split</td>
<td>Overlay maintenance</td>
<td>ID, IP and Port of the new coordinator</td>
<td>New Coordinator</td>
<td>Overloaded Coordinator</td>
<td>Initiate Split and return bootstrap information</td>
</tr>
</tbody>
</table>

61
<table>
<thead>
<tr>
<th>CAN_Dimensions</th>
<th>Overlay Maintenance</th>
<th>Partial information for bootstrapping a new coordinator</th>
<th>Origin and dimensions of the inherited zone</th>
<th>Overloaded Coordinator (parent)</th>
<th>Newly spawned coordinator which inherits half of overloaded coordinator's zone</th>
<th>Initialize Coordinator information</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_Merge</td>
<td>Overlay Maintenance</td>
<td>Request a Merge</td>
<td>Sender's ID</td>
<td>Under-loaded Coordinator</td>
<td>Under-loaded Coordinator's parent/child</td>
<td>Initiate a merge if also under-loaded and Sender's ID matches next expected merge else decline request</td>
</tr>
<tr>
<td>CAN_MergeConfirmation</td>
<td>Overlay Maintenance</td>
<td>Response to Merge Request</td>
<td>Accepted/Declined</td>
<td>Requested Coordinator</td>
<td>Requester</td>
<td>Initiate merge if request accepted</td>
</tr>
<tr>
<td>CAN_NeighbourList Merge</td>
<td>Overlay Maintenance</td>
<td>pass neighbour information during merge</td>
<td>Information about neighbors</td>
<td>Merging child</td>
<td>Merging Parent</td>
<td>Update routing table</td>
</tr>
<tr>
<td>CAN_MergeSubList</td>
<td>Player Management</td>
<td>To pass list of registered player</td>
<td>Information about registered Player</td>
<td>Merging child</td>
<td>Merging Parent</td>
<td>Update Player Registry</td>
</tr>
<tr>
<td>LockRequest</td>
<td>Lock Management</td>
<td>request a lock</td>
<td>Current Lamport timestamp and ID</td>
<td>Requesting Lock Manager</td>
<td>Neighboring Lock Managers</td>
<td>Grant Lock or force to wait</td>
</tr>
<tr>
<td>LockResponse</td>
<td>Lock Management</td>
<td>receive lock grant</td>
<td>Granting Lock Manager's ID</td>
<td>Neighboring Lock Manager</td>
<td>Requesting Lock Manager</td>
<td>Call registered call back function</td>
</tr>
<tr>
<td>Unlock</td>
<td>Lock Management</td>
<td>release locked neighbors</td>
<td>Releasing Lock Manager's ID</td>
<td>Releasing Lock Manager which had formally requested and received a lock</td>
<td>Neighboring Lock Manager</td>
<td>Record release and initiate own request if there is a pending request</td>
</tr>
<tr>
<td>CAN_RequestNeighbourList</td>
<td>Overlay Maintenance</td>
<td>To request NeighborList from Child once the parent has locked its neighbors for a merge</td>
<td>List of neighbors locked by parent</td>
<td>Merging Parent</td>
<td>Merging Child</td>
<td>Requests locks from its neighbors not included in the list received from parent</td>
</tr>
<tr>
<td>CAN_ChangePartner</td>
<td>Overlay Maintenance (Secondary Tree)</td>
<td>Inform Primary Tree Partner to change its Secondary Tree Partner</td>
<td>Information about receivers new secondary tree partner</td>
<td>Splitting/Merging Coordinator</td>
<td>Primary Tree Partner</td>
<td>Update Secondary Tree Partner</td>
</tr>
</tbody>
</table>

### PlayerMD to Coordinator

<p>| InitialRegistration | Player registration | Player's coordinates dimensions of the publish &amp; subscribe areas and connection information | Player | Coordinator (player manager) | Add player to list of registered players and return coordinators for other overlapping regions |
|---------------------|---------------------|--------------------------------------------------------|---------------------------------------------|---------------------------------|---------------------------------------------------------------------------------|---------------------------------|
| Move | Player Management | Inform coordinator of a move | New location and player ID | Player | Registered Coordinators | Update Player location |</p>
<table>
<thead>
<tr>
<th>Event</th>
<th>Module</th>
<th>Input/Output Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LostRegistration</td>
<td>Player Management</td>
<td>Re-register with the proper coordinator</td>
</tr>
<tr>
<td></td>
<td>Current Location</td>
<td>Lost Player</td>
</tr>
<tr>
<td></td>
<td>Lost non-overlapping</td>
<td>registered coordinator in the list</td>
</tr>
<tr>
<td></td>
<td>Register player if it lies</td>
<td>within own region else forward through the overlay</td>
</tr>
<tr>
<td>LeaveGame</td>
<td>Player Management</td>
<td>leave game</td>
</tr>
<tr>
<td></td>
<td>Player ID</td>
<td>Player leaving the game</td>
</tr>
<tr>
<td></td>
<td>Registered Coordinators</td>
<td>Remove player from list of registered players</td>
</tr>
<tr>
<td>Coordinator to PlayerMD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AddCoordinatorInit</td>
<td>Player Management</td>
<td>Inform new player about overlapping coordinators</td>
</tr>
<tr>
<td></td>
<td>Coordinator IDs and</td>
<td>Primary Coordinator</td>
</tr>
<tr>
<td></td>
<td>connection information</td>
<td>New Player</td>
</tr>
<tr>
<td></td>
<td>Add primary coordinator and</td>
<td>initiate registration with other coordinators in the list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AddCoordinator</td>
<td>Player Management</td>
<td>Inform player of other overlapping coordinators if a player moves into a neighboring</td>
</tr>
<tr>
<td>(afterRequest)</td>
<td>Coordinator IDs and</td>
<td>zone or in response to initialRegistration</td>
</tr>
<tr>
<td></td>
<td>connection information</td>
<td>Registered Coordinator</td>
</tr>
<tr>
<td></td>
<td>Player</td>
<td>Register with listed coordinators</td>
</tr>
<tr>
<td>AddSubscriber</td>
<td>Neighbor Discovery</td>
<td>Inform publisher of an overlap with another player’s subscription region</td>
</tr>
<tr>
<td></td>
<td>Subscriber's playerID</td>
<td>Registered Coordinator</td>
</tr>
<tr>
<td></td>
<td>and connection information</td>
<td>Player (publisher)</td>
</tr>
<tr>
<td></td>
<td>Add player to subscribers list</td>
<td></td>
</tr>
<tr>
<td>CoordinatorSplit</td>
<td>Inform registered players if</td>
<td>Coordinator’s new dimensions and connection information</td>
</tr>
<tr>
<td></td>
<td>the coordinator splits</td>
<td>Registered Coordinator</td>
</tr>
<tr>
<td></td>
<td>Player</td>
<td>Update Registered Coordinator list</td>
</tr>
<tr>
<td>CoordinatorMerge</td>
<td>Inform registered player if</td>
<td>Parent’s new dimensions, connection information and merged child’s ID</td>
</tr>
<tr>
<td></td>
<td>the coordinator merges</td>
<td>Registered Coordinator</td>
</tr>
<tr>
<td></td>
<td>Player</td>
<td>Update Registered Coordinator list</td>
</tr>
<tr>
<td>ReregisterCoord</td>
<td>Registration information in</td>
<td>Primary coordinator’s and other overlapping coordinator’s IDs, dimensions and</td>
</tr>
<tr>
<td>(after</td>
<td>response for a LostRegistration</td>
<td>connection information</td>
</tr>
<tr>
<td>LostRegistration)</td>
<td>message</td>
<td>Primary Coordinator</td>
</tr>
<tr>
<td></td>
<td>Player</td>
<td>Add Coordinator to Registered Coordinators and initiate with other listed overlapping</td>
</tr>
<tr>
<td></td>
<td>PlayerID</td>
<td>coordinators</td>
</tr>
<tr>
<td>Coordinator to Central Server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overloaded</td>
<td>Overlay Maintenance</td>
<td>ask CS for an available PlayerMD to share load</td>
</tr>
<tr>
<td></td>
<td>ask CS for an available</td>
<td>connection information and network coordinates</td>
</tr>
<tr>
<td></td>
<td>PlayerMD</td>
<td>Overloaded Coordinator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Central Server</td>
</tr>
<tr>
<td></td>
<td>Finds an available PlayerMD</td>
<td>with similar network coordinates</td>
</tr>
<tr>
<td>Merged&amp;available</td>
<td>Overlay Management</td>
<td>Inform of a merge and availability</td>
</tr>
<tr>
<td></td>
<td>PlayerID</td>
<td>Merged Child</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Central Server</td>
</tr>
<tr>
<td></td>
<td>Adds PlayerMD to list of</td>
<td>available Players</td>
</tr>
<tr>
<td></td>
<td>available Players</td>
<td></td>
</tr>
<tr>
<td>RegisterPlayer</td>
<td>Register new Player</td>
<td>PlayerID, publish/subscribe areas and connection information</td>
</tr>
<tr>
<td>(from CS to Coordinator)</td>
<td></td>
<td>Central Server</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Primary Coordinator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>message routed through the overlay</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add player to list of registered players and return coordinators for other overlapping</td>
</tr>
<tr>
<td></td>
<td></td>
<td>regions</td>
</tr>
</tbody>
</table>

63
Table 6-1 Message Types

<table>
<thead>
<tr>
<th>Central Server to PlayerMD</th>
<th></th>
<th></th>
<th></th>
<th>Central server</th>
<th>Available PlayerMD</th>
<th></th>
<th>Central server</th>
<th>Available PlayerMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpawnCoord</td>
<td>Ask PlayerMD to start a Coordinator on its machine</td>
<td>Connection information of overloaded coordinator</td>
<td></td>
<td>Central server</td>
<td>Available PlayerMD</td>
<td>Starts a coordinator which contacts the overloaded coordinator for bootstrap information</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Player to Central Server</th>
<th></th>
<th></th>
<th></th>
<th>Central server</th>
<th>Available PlayerMD</th>
<th></th>
<th>Central server</th>
<th>Available PlayerMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Register a new Player</td>
<td>Player’s coordinates dimensions of the publish &amp; subscribe areas and connection information</td>
<td></td>
<td>New player joining the game</td>
<td></td>
<td>Central Server</td>
<td>Adds player to list of available PlayerMDs and forwards an InitialRegistration message through the Overlay</td>
<td></td>
</tr>
</tbody>
</table>

| PlayerMD to Player MD | | | | Publishing player | | subscribing Player | Publishes event to its simulation engine |
|---|---|---|---|---|---|---|
| Move | Direct player to player event delivery | Event type and related properties | Publishing player | | | | |

6.5 Removing CS Bottleneck

Other than being a rendezvous, the central server’s main responsibility is keeping track of PlayerMDs that are already running a coordinator. There are a number of ways of further distributing this load as well.

6.5.1 Secondary Overlay

The available PlayerMDs can form an overlay similar to the coordinators overlay. The main difference being that all participating nodes are inserted into the overlay on the basis of their network coordinates. The PlayerMDs are arranged in the overlay according to their network coordinates. Whenever, an overloaded coordinator needs another machine it routes a message in this secondary overlay to a point (x, y) where (x, y) are the network coordinates of the overloaded coordinator. The owner of the zone in which (x, y) lies is chosen as the PlayerMD to spawn a new coordinator. The owner is also removed
from the secondary overlay as it is no longer available to spawn a coordinator. Because of the properties of the overlay the chosen node is guaranteed to be close in terms of network coordinates to the overloaded coordinator.

6.5.2 Reusing Primary Overlay

Alternatively, PlayerMD's network coordinates can be mapped onto the overlay of MDCs. A translation from network to game coordinates is required to handle the difference in dimensions of the game map and the limits of the network coordinates. The overlay space is still split according to the loads on the coordinators. A coordinator may store multiple available PlayerMDs whose translated coordinates lie within its zone.

To search for an available PlayerMD a coordinator first checks its own zone to see if there are any available PlayerMDs. If not, it can initiate iterative search queries (using techniques discussed in Appendix B) asking all coordinators within the bounded rectangle to return a PlayerMD closest to the overloaded coordinator's network coordinates from the set of available PlayerMDs registered with them.

As mentioned above the current implementation uses the Central Server to locate available PlayerMDs and it does not support any of the techniques described in Sections 6.5.1 and 6.5.2.

6.6 Comparison with other DHT based Schemes

The salient features of the architecture are distributed dynamic load balancing and more efficient placement of zones on the overlay nodes. There a number of schemes that propose using fixed sized zones/cells. The problem with these systems is that the zone
coordinator for all purposes is very much like a mini-server that can become a bottleneck. Moreover, having unnecessary partitions even in times of low load results in a higher possibility of inconsistency and redundant coordination effort between the zone coordinators.

The DHTs also do not use any game level or network topological information while assigning zones to nodes in the overlay. Therefore, neighboring zones in the game space will usually end up many overlay hops away from each other. Some proposals counter that by adding links in addition to the overlay connections between neighboring coordinators. In contrast, for the architecture proposed in this thesis the overlay links are alone sufficient to route messages efficiently with the minimum possible delay.

Even in the topology aware Pastry, a node only selects the next hop for a message based on the network information. The nodeIDs are still assigned randomly to nodes. Therefore, even if similar IDs were assigned to neighboring coordinators in the game space, the corresponding nodes might still be far away from each other in the network.

It is also not possible for a Pastry based gaming system to dynamically change the dimensions of the zones as these schemes assume that an objects coordinates can be mapped to a cell_id by using a predetermined function. If the zone sizes and boundaries change dynamically a straight forwarding mapping function can no longer determine the correct cell_id. This problem does not arise in the above architecture because a player only needs its own coordinates to locate its primary coordinator or as these schemes would like to call them 'Home Nodes.'
Chapter 7

Performance Evaluation

We have implemented a prototype that allows the coordinators to dynamically split and merge as needed while maintaining a consistent overlay topology. The prototype also defines protocols used between the components of the architecture to handle player registration, player movement, AoI overlap notification and inter-coordinator communication. The current prototype only maintains simple state in the form of player location and their AoIs. We have also implemented all the advanced overlay routing mechanisms discussed in Chapter 5. In this chapter, we discover system properties, study how they are affected by increasing number of players and finally, validate our claim of the routing improvements.

We conducted extensive tests involving up to a 32000 overlay nodes to verify our routing algorithm and topology construction visually using the visualization tool, shown in Figure 7-1.

Simulations were run with 1000, 2000, 3000, 4000 and 5000 players. The purpose of these tests was to see how evenly the workload and number of links required by each entity are spread. The experiments simulated 1 million second of game play with each
player issuing an event every 500 millisecond. The statistics were collected once the simulation had run for the intended period of time and taken as being a sample for the entire duration of the simulation. This section also includes experimental results as evidence supporting routing improvement through the alterations suggested in this thesis.

The GUI displays the Zone Boundaries, Player Locations and Inter-Coordinator and Inter-Player Connections. Also more detailed player and coordinator properties can be viewed (not shown in the diagram)

Figure 7-1 Visualization Tool - Screenshot
7.1 System Properties

Since all coordinators were simulated on the same machine we could not use the automatic load detection algorithm of Section 3.12 instead using weighted averages we measured the bandwidth consumed at each coordinator and set 1 and 3.5 KB/s as the merge and split thresholds, respectively. This led to a very fair distribution of players over the coordinators. Figure 7-2 and Figure 7-3 show that the splitting and merging algorithms are able to effectively limit the load on the coordinators within desired thresholds. The mechanism is quick to respond to any uneven load distribution as there are no coordinators that lie too far outside both the thresholds. The bandwidth requirement for coordinators grows with the number of registered players. Figure 7-3 shows the strong correlation between the two. On an average an additional player registered with a coordinator requires roughly 49 bytes per second.
The expected number of subscribers per player is given by the equation (see appendix A1 for derivation of this formula):

\[
E(\text{Subscribers}) = \frac{N}{W_D^2} \times \lambda \times \left( \frac{\sqrt{2 \times AoI_D^2 + AoI_D}}{2} \right)^2
\]

Where:

Dimensions of the virtual world = \(W_D \times W_D\)

Number of players = \(N\)

Dimensions of AoI = \(AoI_D \times AoI_D\)

For the simulation discussed in this section the values of the variables are \(N=5000, W_D=1500, AoI_D=15\). Inserting these values in the above equation yields an average 2.287 subscribers per player. The average for the experiment was 2.08 connections per
player which is very close to the theoretical estimate derived from the equation. Figure 7-4 shows the distribution of inter-player connections. The graph shows that almost 70 percent of the players required two or less connections while no one was connected to more than 7. This is a marked improvement over Solipsis [8] and Voronoi [26] which require connections to all players around the player for neighbor discovery. The number of player-to-player connections in the case of Voronoi climbs rapidly as the number of players is increased. [26] reports an average as high as 14 connections per player for the Voronoi scheme with a total of only 250 players. The authors of Solipsis did not publish any experimental results but it is clear that it will require much more than 2 connections per player to maintain the convex hull around each player.

![Figure 7-4 Subscribers per Player](image)

The authors of [27] contend that one of the strengths of their system, MOPAR, is a fixed number of neighboring coordinators. Even though our scheme dynamically and distributedly draws the zones it still yields an average of 7 neighbors per zone. In the distribution of Figure 7-5 82% of the nodes have 7 or less neighbors.
The above figure (Figure 7-6) shows that having less neighbors might not be that important. As players on an average stay within the interior of the zone and establish connections with multiple coordinators only for fleeting periods of time when they cross borders. Almost 80% of the players are connected only to their primary coordinator.
7.2 Scalability

The experiments returned very encouraging results vis-à-vis scalability. The simulation was run with different number of players ranging from 1000 to 5000. The neighbors/coordinator, bandwidth/coordinator, and players/zone stayed relatively constant over the range of players. Figures Figure 7-7, Figure 7-8 and Figure 7-9 show that the scheme is very scalable from the perspective of the coordinating infrastructure.
However, the average bandwidth used by players consistently increases at the rate a constant rate (Figure 7-10). This increase is caused because of the increased communication between players with overlapping AoIs. Since the size of the virtual world was fixed as the number of players is increased their density rises as well leading to a higher likelihood of overlapping publish/subscribe areas. This reasoning is supported by the higher average number of subscribers per node for simulations with higher number of total players as depicted in Figure 7-11. The bandwidth per player grows at a rate slower rate than the average number of subscribers. The bandwidth shown in Figure 7-10 includes the playerMDs’ communication with the coordinating infrastructure in addition to its subscribers. Since bandwidth consumed for publishing events to subscribers is only a portion of the sum, the rate of growth of the Total Bandwidth is slower than the growth rate of the average number of subscribers per player.
7.3 Routing

The original CAN scheme requires $O(N^{0.5})$ hops in the worst case to route a message where $N$ is the number of nodes in the overlay. This is one of the main reasons for researchers to favor Pastry which requires only $O(\log N)$ hops. Pastry however, is
much less flexible in terms of how values are placed on the overlay nodes and it does not allow semantically aware placement. Our improvements to the routing scheme led to considerable reduction in the routing cost. Because of the tree structure superimposed on the overlay by the Reverse BigHop, the overlay can now route messages within $O(\log N)$ hops. The values for Figure 7-12 and Table 7-1 were collected by routing messages between a 100 randomly selected source and destination coordinates in the overlay.

![Figure 7-12 Routing Comparison](image)

<table>
<thead>
<tr>
<th></th>
<th>2K</th>
<th>4K</th>
<th>6K</th>
<th>8K</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Avg</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CAN</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Center</td>
<td>10.28</td>
<td>23</td>
<td>1028</td>
<td>13.8</td>
</tr>
<tr>
<td>BigHop</td>
<td>8.94</td>
<td>22</td>
<td>894</td>
<td>11.8</td>
</tr>
<tr>
<td>Reverse BigHop</td>
<td>8.65</td>
<td>21</td>
<td>865</td>
<td>10.8</td>
</tr>
</tbody>
</table>
Table 7-1 Routing Comparison

<table>
<thead>
<tr>
<th></th>
<th>8K Avg</th>
<th>8K Max</th>
<th>8K Sum</th>
<th>10K Avg</th>
<th>10K Max</th>
<th>10K Sum</th>
<th>12K Avg</th>
<th>12K Max</th>
<th>12K Sum</th>
<th>16K Avg</th>
<th>16K Max</th>
<th>16K Sum</th>
<th>32K Avg</th>
<th>32K Max</th>
<th>32K Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN Original</td>
<td>21.78</td>
<td>51</td>
<td>2178</td>
<td>26.24</td>
<td>56</td>
<td>2624</td>
<td>26</td>
<td>64</td>
<td>2630</td>
<td>32.93</td>
<td>65</td>
<td>3293</td>
<td>51.61</td>
<td>106</td>
<td>5161</td>
</tr>
<tr>
<td>CAN Center</td>
<td>21.04</td>
<td>47</td>
<td>2104</td>
<td>25.73</td>
<td>55</td>
<td>2573</td>
<td>26</td>
<td>61</td>
<td>2595</td>
<td>32.46</td>
<td>65</td>
<td>3246</td>
<td>49.96</td>
<td>105</td>
<td>4996</td>
</tr>
<tr>
<td>BigHop</td>
<td>16.92</td>
<td>46</td>
<td>1692</td>
<td>20.31</td>
<td>45</td>
<td>2031</td>
<td>20</td>
<td>55</td>
<td>1954</td>
<td>25.76</td>
<td>56</td>
<td>2576</td>
<td>35.44</td>
<td>95</td>
<td>3544</td>
</tr>
<tr>
<td>Reverse BigHop</td>
<td>14.41</td>
<td>50</td>
<td>1441</td>
<td>15.99</td>
<td>39</td>
<td>1599</td>
<td>16</td>
<td>56</td>
<td>1604</td>
<td>20.2</td>
<td>53</td>
<td>2020</td>
<td>25.41</td>
<td>103</td>
<td>2541</td>
</tr>
</tbody>
</table>

Figure 7-13 Reverse BigHop - Logarithmic Scale

Figure 7-13 depicts how the number of routing hops increases with the log of the number of nodes in the overlay. The best fit line shows that the routing cost is indeed logarithmic as claimed. In general, our experiments prove that the architecture places an acceptable amount of uniformly distributed load over all components of the system and that the system properties identified in Section 7.1 remain stable as the number of players are increased which augurs well for the scalability of middleware.
Chapter 8

Conclusions

8.1 Future Work: Service Discovery

An entity's position is the most important attribute in a game. Most of the functions are highly correlated to an item's location. We optimized the overlay topology by arranging the nodes according to their location in the virtual world. This also resulted in a simple direct mapping from an entity's location in the game space to the overlays key space and allowed efficient range queries based on the location because of the way overlay nodes are arranged.

Service Discovery [20] is another application that depends on the location of mobile devices and services. Recently, there has been considerable research on incrementally deployable distributed service discovery infrastructure. A number of them propose using P2P overlays [21, 22] that do not exploit any location information in their construction. Moreover, these overlays are highly inefficient in performing range queries,
which are very common in service discovery. Therefore, we feel that our architecture is well suited for such an application.

Our architecture can be easily extended by representing mobile devices and services as players. Mobile service providers would publish services in a 'publication area' and clients will be returned all matching services whose publish area overlaps with the client's 'subscription area'. One difference will be in selecting where to run the coordinator for an area. The choice largely depends on the environment where the application is running and the types of services being supported. In an environment with a large number of mobile devices connected by an ad-hoc network some of the mobile devices themselves will serve as coordinators. A second, more widely accepted model, assumes the presence of coordinating infrastructure. There are two variations of the type of infrastructure assumed to be available. One assumes each point of presence deploys a coordinator [24], while in the other each provider/consumer of a service provides a coordinator [23]. The later requires simple wireless access to the Internet for the mobile devices to connect to the service discovery system; whereas, the former necessitates both an access point and a server to provide service discovery in a given area. In either of these two variations the coordinator could be run on the machines currently performing service matching. Regardless of where the coordinators are run, they will be arranged in a peer-to-peer overlay and assigned ownership of portions of the physical space.

As future work, we plan to adapt the architecture for scalable global service discovery. We are currently leaning towards the model that assumes wireless access to the Internet for mobile devices where service matching is performed by a group of peers. Nonetheless, our architecture is flexible enough to be useful in all types of environments.
8.2 Conclusion

This thesis has provided a general overview of MMOG, its requirements and various architectures that have been used to meet these requirements. We also proposed an architecture that utilizes the players' machines to allow gaming companies to scalably reach out to larger audiences without a huge investment in the infrastructure. The proposed ideas open the possibility of bringing MMOG to a completely new genre of games involving fast paced action that have very strict latency requirements. By adopting the publish-subscribe mechanism we guarantee near optimal area of interest management at the same time as partitioning the workload dynamically without introducing unnatural, in-transparent boundaries in the virtual world. We also show how the tried and tested Trailing State Consistency protocol nicely fits on to the architecture to provide persistent and fault tolerant management of players and terrain objects and propose an extension of the architecture for service discovery.

We hope that the improvements in the routing algorithm and the addition of support for bounded multicast to the underlying overlay will encourage numerous exciting P2P applications.
Appendix A
Derivations

A.1 Expected Average Subscribers per Player

Dimensions of the virtual world $= W_D \times W_D$

Number of players $= N$

Density of Players $= D = \frac{N}{W_D^2}$

Dimensions of AoI $= AoI_D \times AoI_D$

Maximum possible distance, $A$, between two players with overlapping AoIs

$= \text{diagonal of the AoI} = \sqrt{2 \times AoI_D^2}$

(when the line joining two players runs along their diagonals)

Minimum Distance, $B$, between two non-overlapping player $= AoI_D$

(when the players AoIs align perfectly horizontally and vertically)

Average Distance, $R = \frac{A+B}{2} = \sqrt{\frac{2 \times AoI_D^2}{2} + AoI_D}$
Since a player is connected to at most all players that lie within a circular space of radius $R$ around itself the average number of subscribers per player = expected number of players with this circular space.

Expected number of players within this space = Player Density $\times$ Area of Space =

$$D \times \lambda \times R^2 = \frac{N}{W_d^2} \times \lambda \times \left( \frac{\sqrt{2 \times AoI_d^2 + AoI_P^2}}{2} \right)^2$$

* assuming square dimensions for the virtual world and the Aol
Appendix B

Bounded Multicast

B.1 Bounded Multicast and Range Queries

The overlay provides the facility to broadcast a message to a bounded rectangular region within the coordinate overlay space. A message can be broadcasted to all coordinators whose zones overlap a rectangle (x, y, x+width, y+height) by sending a special broadcast message to the broadcast-origin coordinator, BOCoordinator. This is the coordinator that owns the point (x,y), the top left corner of the broadcast rectangle. The broadcast message also contains the xlimit (x+width) and the ylimit (y+width). The BOCoordinator sends a broadcast_X message to all neighbors in its Set 1 and any trespasser from Set 0 whose zone overlaps with the broadcast rectangle. Nodes receiving a broadcast_X message forward it to all their neighbors in Set 1 that overlap with the broadcast rectangle. A node forwards a broadcast_X message to its Set 0 trespasser only if the node's zone lies or crosses the boundary of the broadcast rectangle. If it does not then there is guaranteed to be another neighbor above it, which will forward the message to the trespasser. This rule prevents the transmission of duplicate messages to the trespassers and the nodes downstream from them. The nodes keep forwarding the broadcast_X message until it reaches a node whose zone crosses the xLimit of the broadcast rectangle.
1. Sends to 5 even though it's a trespasser because its own zone lies on the boundary of the broadcast rectangle.
2. Sends Broadcast_Y to only its first neighbor in its region 2 that overlaps the broadcast rectangle.
3. Does not forward Broadcast_X message to 4 because it does not lie on the top boundary of the bounding rectangle.
4. Forwards Broadcast_X message to multiple nodes as they overlap with the bounding rectangle.

Figure B-1 Bounded Broadcast

The BOCoordinator also sends a broadcast_Y message to the first node in its Set 2. A node receiving a broadcast_Y message in addition to propagating the message to its first neighbor in Set 2, generates a broadcast_X message for nodes in its Set 1. The propagation of broadcast_X and broadcast_Y stops when xLimit and yLimit are reached, respectively.

This algorithm guarantees that no node receives a duplicate message. This is achieved without keeping any message histories and the need for generating and maintaining any sequence numbers. The approach is also more resilient to faults as there is no risk of losing sequence number records due to faulty nodes. It also allows bounded multicast. None of the known schemes [28] for similar overlays provide any of these benefits.
Bounded multicasts can be used to search for values stored in sections of the overlay. It can support range queries like ‘find all objects with a property X that lie within a bounded rectangle.’ For a gaming middleware, this facility can be used for multicasting an event like a nuclear detonation to all the players/objects within the area affected by the explosion.

There are applications that search for objects within some area and expand the search space when the smaller area fails to return any useful results. More often than not the expanded area includes the area that has already been explored in the first failed query. So optimization can be made to prevent redundant searches in an already explored area.

![Figure B-2 Redundant Queries Prevention](image)

Querying the expanded area would require searching the shaded area A, again. To prevent that we split the expanded search space as show below:
Four separate bounded broadcast messages are sent to BOCoordinators for 1, 2, 3 and 4 with the corresponding rectangular bounds. The four smaller rectangles together with A make up the whole area of the expanded search space. This also results in faster query processing time as the query is computed in parallel in all the rectangles. If the expanded search space does not return any satisfactory results, the search space is expanded again around the old expanded search spaces. This provides a mechanism for iteratively expanding search spaces without redundant computation and messages which is important for a number of applications including service discovery.

**B.2 Query Termination**

Another non-trivial problem associated with range queries is determining query termination at the querier. Every node forwarding a broadcast message sends to the querier a list of IDs of all the coordinators to which it is forwarding the message. Based on this list the querier can keep track of all nodes that have not responded. Keeping simple counts of nodes receiving the multicast would have been enough to determine when a query is complete as each node is guaranteed to receive only one broadcast
message/range query. Having richer information (ID and coordinates) of the involved
nodes provides alternatives in case of nodes not returning any results for the query. Using
the coordinate information the query can be restarted from where it failed.
Bibliography


