Monitoring Distributed Virtual Worlds

Hammad Ullah Khan

Master of Science

School of Computer Science

McGill University
Montreal, Quebec
2012-04-10

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

Copyright © Hammad Ullah Khan, 2012. All right reserved.
ACKNOWLEDGEMENTS

I would like to acknowledge my supervisor, professor Jörg Kienzle, for his help, effort and time. Without his assistance this thesis would not have been possible.

I would like to thank the Mcgill University, the School of Computer Science, and the people involved with Mammoth. They provided me with the opportunity, environment and the support necessary to complete this research.

Finally I would like to thank my parents for their support, their faith in me, and, not to mention, the occasional badgering that played in important role in the completion of this work.
ABSTRACT

Recent years have seen a huge growth in the demand for online virtual worlds. The type of these online systems can range from virtual meeting setups, to a more video game like competitive environment. An equally large number of virtual worlds have been developed to meet this demand, and the competition between these systems is very strong. Developers of such systems can benefit from any edge they can get in terms of technical quality of the system or the enjoyment of the online experience.

We propose that a monitoring system designed especially for virtual worlds will be able to provide that “edge” to the developers. As such, we present, in this Thesis, a flexible real-time monitoring architecture which caters to the specific challenges and requirements of virtual worlds. Handling huge amounts of data present in the worlds is dealt by distributing the data gathering process between multiple nodes. The proposed system modifies the gathered data, into a form more suitable for users to observe in real-time, by filtering it before displaying the final result.

We use Mammoth, a massively multiplayer research framework, as the test-bed for a sample implementation of the proposed architecture. We use the results of experiments conducted on this implementation to validate that the system is indeed suitable for real-time monitoring of virtual worlds.
De nos jours, la demande des mondes virtuels est en plein essor. Ceux-ci vont des sites de rencontre jusqu'aux environnements compétitifs comme par exemple les jeux vidéo en ligne. Afin de satisfaire la demande de mondes virtuels, de nombreux sites ont été mis en place. Du fait de la très grande concurrence présente, les développeurs des services virtuels essayent de bénéficier de tout avantage possible en termes d’avantages techniques ou de la qualité des expériences vécues en ligne.

Nous considérons qu’un système de surveillance des mondes virtuels est en mesure de fournir cet ”avantage” aux développeurs. Ainsi, nous présentons dans notre thèse un système de surveillance en temps réel fait sur mesure afin de faire face aux défis et aux besoins particuliers de chaque monde virtuel. Afin de manipuler toute l’information obtenue des mondes virtuels, le processus d’obtention des données est distribué entre plusieurs nœuds. Le système que nous proposons modifie les données obtenues pour les rendre plus faciles à observer en temps réel. Ceci se fait en filtrant les données avant de déployer les résultats.

Nous utilisons Mammoth, une infrastructure massif de recherche multi-joueurs comme le banc d’essai pour implémenter un échantillon de l’architecture proposée. Nous utilisons les résultats obtenus des expériences réalisées dans cette implémentation pour confirmer que le système est approprié pour surveiller les mondes virtuels en temps réel.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ................................................. ii

ABSTRACT ............................................................ iii

ABRÉGÉ ............................................................... iv

LIST OF FIGURES .................................................... viii

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

1.1 Contributions .................................................... 4

1.2 Thesis Breakdown ............................................... 8

2 Monitoring Architecture ........................................... 9

2.0.1 Background ................................................... 9

2.1 Collecting ....................................................... 11

2.1.1 Collecting information ...................................... 11

2.1.1.1 Using the Servers to Collect Information .............. 11

2.1.1.2 Using Dedicated Collector Nodes ....................... 12

2.1.2 Handling Large Amounts of Information ................. 13

2.2 Filtering ......................................................... 15

2.2.1 Reduction and Accuracy of Information ................. 16

2.2.2 Considering the Observer View ........................... 17

2.3 Presenting ....................................................... 18

2.3.1 Displaying Information .................................... 18

2.3.2 Choosing What to Display ................................. 18

2.4 Other Concerns and Ideas ..................................... 19

2.4.1 Scalability ................................................... 19

2.4.1.1 Multilayered Collectors ................................. 19

2.4.1.2 Controlling traffic to presentation client ............ 20

2.4.1.3 Load Balancing ......................................... 21

2.4.2 Fault Tolerance ............................................. 21
2.4.3 Filtering at the Collector vs. Servers ........................................ 22

3 Monitoring with Replicated Objects ................................................. 24

3.1 Replicated Object Architecture ..................................................... 24
   3.1.1 Replicated Objects ............................................................. 24
   3.1.2 Distribution of Objects ....................................................... 26
   3.1.3 Fault Tolerance ................................................................. 26

3.2 Monitoring Architecture .............................................................. 27
   3.2.1 Collectors ................................................................................ 27
   3.2.2 Filtering ................................................................................ 28
   3.2.3 Presenting .............................................................................. 29

4 Implementation of the Monitoring System in Mammoth ..................... 32

4.1 Background of Mammoth .................................................................. 32
   4.1.1 Replicated Objects .................................................................. 33
   4.1.2 Interest Management ............................................................... 34
   4.1.3 Cells and Servers ................................................................... 35
   4.1.4 Communication Architecture .................................................. 35
   4.1.5 Load Balancing ...................................................................... 36
   4.1.6 Fault Tolerance ...................................................................... 37

4.2 Implementation of the Monitoring System ........................................ 38
   4.2.1 Collectors ................................................................................ 39
      4.2.1.1 Collector Start-Up ............................................................... 41
      4.2.1.2 Load Balancing at the Collectors ....................................... 42
      4.2.1.3 Fault Tolerance at the Collectors ....................................... 42
   4.2.2 Filtering ................................................................................ 44
      4.2.2.1 Filtering Algorithm ............................................................ 45
   4.2.3 Presentation Client ................................................................ 47
      4.2.3.1 Managing MOs ................................................................. 47
      4.2.3.2 Further Merging of the MOs at the God Client .................. 49

5 Experiments ....................................................................................... 52

5.1 Delay due to Redirection ................................................................. 53
5.2 Limiting of Update Messages ......................................................... 54
5.3 Error in Player Position ................................................................. 57
5.4 Error in Player Position Before and After Merging ......................... 60
5.5 Error Due to Load Balancing Delay ................................................ 62
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2–1</td>
<td>Collecting Directly from the Servers</td>
<td>12</td>
</tr>
<tr>
<td>2–2</td>
<td>Dedicated Collector Nodes</td>
<td>13</td>
</tr>
<tr>
<td>2–3</td>
<td>Multiple Layers of Collectors</td>
<td>20</td>
</tr>
<tr>
<td>3–1</td>
<td>Replicated Objects update process</td>
<td>25</td>
</tr>
<tr>
<td>3–2</td>
<td>Filtering Process</td>
<td>29</td>
</tr>
<tr>
<td>3–3</td>
<td>User Feedback and Monitor Object update</td>
<td>30</td>
</tr>
<tr>
<td>4–1</td>
<td>Components of the Mammoth Framework [13]</td>
<td>33</td>
</tr>
<tr>
<td>4–2</td>
<td>Different Network Topologies</td>
<td>36</td>
</tr>
<tr>
<td>4–3</td>
<td>Collector Architecture</td>
<td>39</td>
</tr>
<tr>
<td>4–4</td>
<td>Adding new CO to Collector Ring</td>
<td>42</td>
</tr>
<tr>
<td>4–5</td>
<td>Fault Tolerance using the Collector Ring</td>
<td>43</td>
</tr>
<tr>
<td>4–6</td>
<td>Effects of Merging at the God Client</td>
<td>50</td>
</tr>
<tr>
<td>4–7</td>
<td>Presentation Client</td>
<td>51</td>
</tr>
<tr>
<td>5–1</td>
<td># of Update Messages (Used Bandwidth)</td>
<td>55</td>
</tr>
<tr>
<td>5–2</td>
<td>Position Errors vs. Zoom Level / Configurations</td>
<td>57</td>
</tr>
<tr>
<td>5–3</td>
<td>Relative Error / Visible Area of the World</td>
<td>58</td>
</tr>
<tr>
<td>5–4</td>
<td>Error/Zoom - Multiple Collectors, Merging at the God Client</td>
<td>60</td>
</tr>
<tr>
<td>5–5</td>
<td>Average Error/Time while players are moving to one cell</td>
<td>63</td>
</tr>
</tbody>
</table>
List of Algorithms

1  Update View .......................................................... 46
2  Filtering Algorithm ................................................... 48
Chapter 1
Introduction

Virtual worlds are distributed systems that provide the users with an online setting and a character with which to connect to the world and interact with other users and world objects. Some virtual worlds, such as Second Life[15] provide the users with a platform for online presence, which can be used for conducting events such as online meetings and socializing. Others, such as World of Warcraft[9] provide more complex, game like settings where the users have to traverse through the world competing with other online users in an attempt to complete some predefined goals. Even in the latter, game like, virtual worlds, the goals themselves can vary from command and conquer to a more role playing approach which combines the elements from both the online presence and game based worlds.

Recent years have seen a major growth in all types of virtual worlds, which allow hundreds of thousands of players to connect and control the online characters. With the amount of competition among such services and the huge user base, the developers have to continually improve the virtual worlds both in terms of system efficiency and user experience and at the same time introduce new material to ensure that the current users remain engaged. The persistent nature of the worlds allow the users to go offline and connect at a later time to continue with their character in the same state, which requires that the service has to run without any downtime.
The balance between the different elements of the virtual worlds, such as the story line, the capabilities of the characters, and the type of interactions possible with the world objects, has a major impact on its overall quality and enjoyability. When making improvements or introducing new material, the developers and designers need to ensure that this balance is maintained.

Ideally any new changes that are to be introduced into the virtual world should be well tested before being published for all the users. Unfortunately, the high availability requirement of the service makes it very hard for the developers to conduct such tests with a sufficiently large number of users to replicate a realistic environment.

Another challenge for the developers and maintainers of such high availability services is to monitor the state of the online system to ensure that any problems such as over loading or component failures can be detected early and fixed before the overall world is affected. Besides such extreme cases, the developers also need to collect the data from a running virtual world so that they can find potential problems, or points of improvement. The distributed nature of the world, and its size makes this monitoring task very difficult.

In this thesis we present a solution to the above mentioned problems, related to creation of new content and maintenance of the system, by proposing a distributed monitoring system for such virtual world. While many monitoring systems exist for distributed systems, our proposed system specifically targets virtual worlds, and takes into account the unique properties and challenges associated with them. The monitoring system architecture is flexible enough to allow the developers to observe
any part of the virtual world in real-time or monitor any kind of system statistics such as the load on the servers.

When monitoring distributed virtual worlds, some of the challenges that we face are:

- The virtual worlds consist of huge amounts of discrete information related to the state of each object within them. This data is also not maintained at any single location, but is spread over a set of distributed nodes in the system. We need to gather this data from all the distributed nodes before it can be presented to the monitoring system user.

- We need to gather this data in real time due to the dynamic nature of the virtual worlds. As the state of the virtual worlds changes quickly, data becomes stale fast. Our gather process needs to ensure that the data gathered is acceptably recent to provide real-time observing for the user. Of course, the time interval during which gathered data remains useful might vary depending on the intended objective of the user of our system.

- The real-time nature of, and the large amount of information gathered from, the virtual world poses a presentation challenge. The size of data combined with the fast paced changes makes it nearly impossible for the users to make sense of it in real-time. We therefore need to filter and aggregate the gathered data to make it more accessible to the user.

Besides addressing the above mentioned challenges, our monitoring system allows the developers fine grained control over the type of information they need to monitor and the level of detail they require. The developers can choose to observe fine detailed
information about a localized area of the virtual world, or they can observe the whole world at a decreased level of detail. The decrease in detail makes the overview more concise and easily understandable, as well as limits the load on the monitoring system to ensure optimal performance.

Since our monitoring system is itself distributed among multiple nodes, we also address the issues related with any distributed system. Problems such as scalability, real-time nature, load balancing, and fault tolerance are catered by having a flexible architecture which can be adjusted to suit the requirements of the developers, and the design of the virtual world.

The next section lists the design of the proposed monitoring system and the contributions that this research has made towards addressing the challenges and problems mentioned above.

1.1 Contributions

There are a number of conceptual, architectural and technical challenges that a developer needs to face when developing a monitoring client for games with potentially thousands of players. In this research I have highlighted these issues, and provide techniques and algorithms that help overcome them. Mammoth - an MMO game - was used as the base for implementing some of the proposed ideas in an attempt to prove their validity and performance. Below I give a brief overview of the major contributions, by listing the difficulties faced, and the solutions that are suggested in this thesis.
Since the monitoring client can not possibly connect to all participating nodes (players/servers) to collect information, we propose a hierarchical, load balancing, fault-tolerant monitoring architecture:

- The virtual world is partitioned into regions of a reasonable size. Each region is assigned a Collector node that observes all relevant information of the region.

- By adjusting the size of the region assigned to a collector node, our architecture is capable of balancing the load between the active collectors to be able to cope with player flocking scenarios or other irregular game load distribution.

- To provide fault tolerance, the collector nodes monitor each other. In case a collector node fails, one of the other collector nodes will continue collecting game information from the region of the failed collector.

- To bound network usage, each collector node aggregates and reduces the observed information before forwarding it to the monitoring client. This mapping takes into consideration the view point of the monitoring client, and adjusts the algorithm accordingly: If the monitoring client wants to observe information from a very specific area of the game, and for a very small number of game objects, it is possible to forward detailed data to the monitoring client without danger of overloading the network. On the other hand, if a larger variety of information is being observed stemming from a larger number of game objects or a bigger part of the virtual
world, the accuracy of the mapping is reduced, or the time intervals at which state updates are sent to the monitoring client are lengthened.

- We provide a design of our monitoring architecture based on Replicated objects. Replicated objects are used as a communication mechanism between the game nodes and the collector nodes, as well as between the collectors and the monitoring client:
  - The collectors register for game object replicas in the region of the virtual world that they are assigned to, and hence receive game state updates whenever the state of the master game object changes.
  - Each collector also hosts a set of special master objects called Monitor Objects. They are used to store the collected aggregated data. Contrary to standard replicated objects, they only broadcast state changes to replicas when a certain time interval has expired. The monitoring client has replicas of all monitor objects, and therefore receives the data gathered by the collectors in periodic intervals.
  - The monitoring client defines a master View Object that stores information about what game data and part of the virtual world the monitoring client is currently interested in. Collectors have replicas of the view object, and can therefore adjust their data gathering strategy whenever the monitoring client’s point of view changes.
  - Each collector node also hosts a master Collector Object that stores the region of the virtual world that the collector is currently assigned to. Adjusting this region enables load balancing. All collector nodes have
replicas of all collector objects. This makes it possible to provide fault
tolerance.

• We demonstrate the applicability of our monitoring framework by designing
specific Monitor Objects that gather player position information. To reduce
the amount of data to be forwarded to the monitoring client, monitor objects
represent groups of players. The mapping algorithm that takes players and
maps them to a group object takes into account the current view of the mon-
itoring client in order to provide the client with the best possible information
accuracy.

• To validate our ideas and conduct real-world experiments, we implemented
the proposed monitoring framework in the context of Mammoth, a massively
multiplayer game research framework. Mammoth allows us to build virtual
worlds, host them on distributed servers, and connect hundreds of human or
computer-controlled players.

• We present performance measurements obtained by running several real-world
experiments on our implementation. The experiments showed that:
  – The delay introduced by using dedicated collector nodes to filter the game
    state updates is minimal.
  – Our approach is capable of bounding the network band-width used by
    each collector node.
  – To achieve the highest data accuracy when observing player positions, the
    ideal ratio of how many monitor objects should be used on each collector

and of how often updates should be sent to the monitoring client depends on the zoom level used on the monitoring client.

1.2 Thesis Breakdown

The rest of the thesis is structured as follows: Chapter 2 describes the general architecture of the monitoring system, without assuming the type of architecture used by the underlying virtual world. Chapter 3 provides an overview of the replicated objects scheme of distributing information and gives additional details about our architecture in context of virtual worlds that utilize such a scheme. Chapter 4 starts with an introduction to Mammoth, an MMO game, and then explains how we implemented the monitoring system for it. Chapter 5 lists the experiments we conducted on our implemented system, and the results we obtained. Chapter 6 presents some of the previous work done on the problem of monitoring large systems, and finally Chapter 7 concludes this thesis and talks about some future work.
Chapter 2
Monitoring Architecture

Any monitoring process consists of three phases: Collecting, Filtering, and Presenting the information. As mentioned in the introduction, monitoring the state of a virtual world is challenging for several reasons: 1) the distributed information must be gathered in real-time without affecting the game itself; 2) the amount of information gathered is so large that it is impossible for a single node to collect and process all of it; 3) the vast information must be filtered according to what the monitoring person wants to focus on, and the point of interest of the monitoring person can change frequently.

2.0.1 Background

Before describing our proposed monitoring architecture it is important that we define some properties of the underlying system, and the terminology that we will employ in this and the subsequent chapters. When talking about any virtual world, we need to differentiate between the different components and the users of the system.

Software Components:

The components are software based parts of the system, which communicate with each other to form the overall distributed application. In this thesis we collectively refer to these software components as nodes. The main nodes that we talk about in this and the following chapters are:
Servers: The virtual world is distributed between many different nodes, and since its users only connect with the system temporarily (i.e. they can connect or disconnect anytime), we need to have some nodes in the system that are able to persistently maintain the state of the virtual world. We use the word server to denote these nodes. These servers collectively possess the complete information about the virtual world. These servers also remain connected to the system permanently in order to provide the required persistence. We must stress the fact that although, we call these nodes servers, they are not necessarily servers in the classical client/server sense. In fact, these nodes are also required in virtual worlds that use a peer-to-peer approach, so that the state can be maintained as user continue to connect and disconnect.

Client: Client is a separate software that provides the users of the system with an interface to connect with the virtual worlds. There may be different types of clients for different platforms (desktops, mobile devices etc.), however, they all have the same general purpose of allowing the users to connect to and use the virtual world.

MonitoringClient: Monitoring client is a special kind of client, that provides its users with a different interface to the virtual world. It allows its users to connect to the service and observe the virtual world without actually being part of its state.

Non-Software Components:
These are other elements of the virtual worlds that are important to the discussion in the thesis.
**Player:** The client provides its users with a virtual character (also called the avatar), which they control in order to interact with the world. Since in terms of the monitoring system we are only concerned with the in-game avatars, we do not differentiate between the human players and the in-game players and use the term *player* interchangeably.

**User (Observer):** The word *user* is reserved solely for the humans that are observing the game through the monitoring client.

### 2.1 Collecting

The first step in monitoring is to collect the information about the state of the virtual world, which is distributed across the servers that are hosting the world. It is important that this collection process is carried out without actually effecting the system itself. This means that, it should not modify the system or its state in any manner, and it should not introduce delays in the system. The real-time nature of the monitoring system also requires that the information collected is acceptably recent so that an up-to-date picture of the system is presented to the users. Collecting information frequently ensures that the information is up to date, however, it is an expensive operation and may also effect the performance of the monitored system.

#### 2.1.1 Collecting information

Collecting information can be achieved at the servers, or separate nodes can be created that act as the data collectors.

1. **2.1.1.1 Using the Servers to Collect Information**

   The existing game servers can be used to collect the information and forward it to the monitoring client. This approach requires some modifications to the servers so
that they can collect the information required and connect to the monitoring client to transfer the information. Since the servers already have the required information, the collection process, in this approach, is fairly simplified. However, collecting the information and sending it to the monitoring client requires computational resources, which limits the performance of the servers for normal operations. Furthermore, such an approach limits the flexibility of the monitoring system by fixing the number of collectors that can be used by the system.

2.1.1.2 Using Dedicated Collector Nodes

The second solution, and the one we adopted, is to designate separate nodes as information collectors. This requires that the information from the servers be forwarded to these collector nodes. Although separate mechanisms can be developed for this process, it is much better and efficient to use the already existing mechanisms for this purpose. Any distributed system will already have some method of
communicating the information stored in the servers to the connected clients. As opposed to developing a separate method of communication, which would again require modification of the servers, using the existing techniques means that collector nodes can present themselves as clients to the servers and obtain the required information.

This approach also has the advantage that the information collected is always as current and up to date as on any of the connected clients, and that it does not entail any additional load on the servers except that of one new client.

2.1.2 Handling Large Amounts of Information

Virtual worlds can be very large, with the number of players ranging in thousands, and therefore, the amount of information that needs to be collected can also be very large. In any system that requires multiple servers to handle all the information of the virtual world, it is also not possible that a single collector node would be
able to gather all that information. We therefore need multiple collectors to gather all the information and process it.

Distributing the information collection between multiple collectors is also important for the purpose of efficiency and load balancing. It would be beneficial if related information was collected by the same collector, while at the same time ensuring that each collector handles about the same amount of data. This division is an application dependent problem, and the optimal solution for different types of application would vary to a high degree. However, in terms of virtual worlds some of the possible criteria for division are listed below:

1. Division by type of data: For instance, the position of players in the world and the objects in possession of the players are separate types of information. One collector can be assigned to collecting position information, while another collector collects all the objects owned by the players.

2. Division by position: In terms of virtual worlds, it makes a lot of sense to divide information gathering according to the location of the information in the virtual world. Each collector is assigned a portion of the world, and it collects all the information about the players and game objects in that part of the world.

Each of the criteria mentioned above makes sense, depending on the requirements from the monitoring system. Our implementation (chapter 4) utilizes the second approach for division, however, it is very easy to switch between the two. It is also possible that, for some requirements, we can combine the two approaches to get the benefits of each.
2.2 Filtering

The large size and dynamic nature of virtual worlds mean that the information collected by the collectors can potentially be very large and fast changing. Presenting all this unformatted information is not ideal for the user to understand the state of the world in real-time. We, therefore, need to filter and modify the information to make it easier for users to understand and to meet the requirements of the user. While the collectors as a whole collect information about all the objects in the virtual world, the user might only require information about a smaller portion of the world. We define this portion of the world, for which the user has requested information, along with the amount of detail requested, as the view of the user.

This collected information needs to be modified and filtered to make sure the final information is appropriate, easily understandable and presents a clear picture of the state of the world. Such a process requires that

1. the amount of information is reduced while making sure that it remains accurate to a level acceptable for the purpose of the user.
2. only information pertaining to the user’s current view and requirements is presented.

The second point refers to the fact that a user might be interested in a small region of the world, but require a detailed picture of that region. At another moment, he might be interested in more coarse grained information about the entire or a large portion of the world.

The amount of information can be reduced in a number of ways. One way is to simply discard some of the information so that only a small portion of the
information is presented to the user. This approach, however, results in a falsified and incomplete state of the world being displayed to the user. The user might make the wrong decisions based on this incorrect representation of the state.

Our approach is to summarize related segments of the information and present this summary to the user. This idea especially fits very nicely when considering virtual worlds as the monitored systems. Information about objects that are spatially close in the world can be considered as related information and summarized. Object position is a good example of such a process where the positions of a group of similar objects such as players, can be averaged and a single group presented at that position.

The summarizing in our architecture is achieved by aggregating pieces of related information together - individual player positions in the above mentioned example. The aggregated information is stored in special objects we call Monitor Objects. Besides storing the filtered/aggregated information, these Monitor Objects also transfer the information to the next layers in the architecture, such as the presentation client. The number of monitor objects (MOs) between each collector and the presentation client are fixed at the start of the system.

2.2.1 Reduction and Accuracy of Information

By summarizing the information we reduce the amount of data that needs to be sent to the presentation client in the system. For instance by joining nearby player positions into one group, we only need to communicate the position of that one group instead of individual player positions for many players.

Furthermore, by ensuring that the aggregation of information for each MO is smartly selected, we can ensure that the information presented is acceptably accurate.
In our approach, the accuracy is dependent on the amount of information assigned to each MO; a small amount of information per MO means more accuracy. Since we try to assign an equal number of objects to each MO, the actual division of information between the MOs, and therefore, the accuracy, is dependent on the total amount of information and the number of MOs. The optimal number of MOs required for the accuracy desired can be calculated by running experiments while varying number of MOs. We present these experiments in chapter 5.

2.2.2 Considering the Observer View

The observer view refers to the point of interest of the user, i.e. the area of the virtual world and the subset or type of world state that is of interest. If the user is interested in a large portion of the overall system state, then the amount of information of interest is very large. On the other hand, we only need to handle little information if the user is interested in a very specific subset of the state. In the context of a virtual world, if the user is interested in a small portion of the world, then only information about the few objects in that part of the world needs to be considered. Similarly, if the user wants to observe the whole world then all the objects of the world are of interest. Our approach of mapping information to a fixed number of MOs means, that we are able to present a very accurate picture of the system state when the user view is small. When the user view is much larger, more information needs to be aggregated into each of the MOs, thus reducing the achievable accuracy. This behavior is useful because, when viewing a large part of the system state, details are less important then the overview of the system state.
2.3 Presenting

The presentation layer performs two important functions for the system: 1) it displays the information that was collected and filtered; 2) it lets the user define the requirements about the information filtering process.

2.3.1 Displaying Information

Before the information can be displayed it needs to be retrieved from the collectors. In our architecture this transfer of aggregated information from the collectors to the display component of the system is part of the monitor object design. When a monitor object is updated at the collector, it automatically transmits the new information to the presentation client, which can then display the updated state.

The actual presentation of the information, is dependent on the type of system being monitored and the type of information that needs to displayed. The paper [12] discusses the different methods of displaying information.

2.3.2 Choosing What to Display

Users directly interact with the presentation client, and use its control interface to define the user view. Since the current view affects the filtering process, any updates to the observer view need to be transmitted to the collector nodes where the filtering takes place. With this requirement in mind, it is important that the interface provided to the user for controlling the view is intuitive and easy to use. While the exact interface provided to the user is dependent on the underlying system, with virtual worlds, it is clear that a graphical method of control is required. The interface should allow the user to adjust the size and position of the currently viewable region(zoom and pan the virtual world).
2.4 Other Concerns and Ideas

While the sections above define the basic architecture of each of the three layers in our monitoring system, there are additional properties of the architecture that are important to consider: scalability and fault tolerance. This section presents design decisions related to scalability and fault tolerance that we made and how they affect the architecture of our system.

2.4.1 Scalability

As players connect and disconnect from the virtual world, the size of its state can vary dramatically, and we need to make sure that our monitoring system is capable of adjusting to these changes. It is especially important that as the virtual world grows in size, our monitoring system can handle this increase in size by utilizing the available resources efficiently, and, if required, allocating more resources. This section describes some of the design principles adopted for the monitoring client that help us address this requirement.

2.4.1.1 Multilayered Collectors

In case of the virtual world being very large, the game state, and hence the amount of information that needs to be collected, is also very large. In such cases we also need a large number of collectors to handle the amount of information. However, the presentation client, also needs to communicate with each of the collectors, and when the number of collectors gets very large this communication becomes a problem.

In such cases our architecture is flexible enough to allow multiple layers of collectors. The base level collectors collect information directly from the servers and filter it into MOs. The higher layer collectors can then collect these filtered MOs
from the base level collectors, and possibly apply some more filtering before sending it to the presentation client. The type and amount of filtering done at each of the levels can also be modified to control the efficiency and accuracy of the system.

2.4.1.2 Controlling traffic to presentation client

Although the filtering at the collectors, takes care of reducing the amount of information that is presented to the user, we also need to take care of the amount of data transfer between the collectors, and the presentation client. The filtering process is run periodically, and after each run it updates the Monitored Objects. When ever the MOs are updated, they transfer the new information stored in them to the presentation layer. Since the number of MOs is fixed at start of the system, we can control the number of messages between collectors and presenters by controlling the delay between the updates of MOs.
In-fact, the number of messages is equal to total number of MOs divided by the delay between updates. So increasing the delay will reduce the traffic, while increasing the number of MOs, will increase the traffic. This also presents the interesting possibility of maintaining a fixed rate of data between collectors and the presentation client, by increasing the MOs and the delay at the same time. We present an experiments to observe the results of this variation in the system in the experiments section [Chapter 5].

2.4.1.3 Load Balancing

In cases when we are monitoring a highly dynamic virtual world - where the state of the system changes fairly quickly - it is possible that the division of the state between the collectors can also change to a great degree. This results in the load on the collectors becoming more unbalanced as the time progresses, as most of the information collection is shifted to some of the collectors. Therefore, our architecture also needs to have some amount of load balancing to compensate for this. The way load balancing is achieved is dependent on the properties of the underlying system, the type of information being collected and the method of division between the collectors. The load distribution and balancing mechanisms employed by the underlying system, if any, also affect the eventual solution that is suited for the monitoring system. We give a more detailed description of load balancing in chapter 4.

2.4.2 Fault Tolerance

As in any distributed system, there is a chance that nodes in our monitoring system might fail. In case of such a crash, where some of the nodes might fail, it
is important to ensure that the system can recover from these failures and that the overall monitoring process is not interrupted. Our main concern is the failure and recovery of some of the collectors while the system is running. The collectors need to be aware of each other so that they can check if any of the collectors have failed. The collectors, also need to have knowledge of how the information is divided between the collectors, so that in case a collector fails, one of the other collector can assume responsibility for that portion of information.

2.4.3 Filtering at the Collector vs. Servers

One of the design concerns that we had to address was how to collect the information from the servers.

Servers:

The first possible solution to this problem, was to collect only relevant information from the servers and use it for filtering. In this case, if the user is interested in some part of the system state, then, information only about that part of the system will be collected from the servers. This approach mean that the collectors have less load, especially when the user is concerned with fine grained information about a very small portion. However, some of the filtering still needs to be done at the collectors.

Collectors:

The second solution is to collect the complete system state and then filter it with respect the the user requirements, at the collectors. This requires that the load on the collectors is higher but more uniform, since they are always collecting all the information. The complete filtering is then achieved at the collectors.
In our architecture we adopt the second approach. The major reason for this choice is the reaction speed of the system to changes in user view. In case of the first approach, if the user changes the view, the collectors need to calculate the new information that needs to be gathered, retrieve it from the servers, and then apply the filtering. In the second approach the collectors, already have all the required information and just need to adjust the filtering algorithm to adjust the information being sent to the presentation client.
Chapter 3
Monitoring with Replicated Objects

When describing the monitoring system architecture in the previous chapters, I have attempted to present it in a way that is as generic as possible, and hence, can be applied to a wide variety of applications. However, with this generic architecture, we have to skip many of the finer detail that are of importance to the monitoring system but are heavily dependent on the type of underlying application. In this chapter I present a more detailed picture of the architecture, with reference to a specific type of application, that uses distributed, replicated objects, in order to implement a distributed virtual world.

The first section describes the architecture of the underlying system, before I describe how the proposed monitoring architecture can be adopted to that system, in the subsequent sections.

3.1 Replicated Object Architecture

This section describes how replicated objects can be used in the context of a distributed application to provide an interface to share data and information and isolate the distributed application from the network communication design.

3.1.1 Replicated Objects

In a replicated object scheme of distribution, each aspect of the system that needs to be shared among different components of the system, is abstracted as an object [8]. When a node within the system needs information encapsulated within
such an object, or needs to make changes to the state of the object, a copy of that object is created on that node.

Although the objects can have multiple copies distributed in the system, there is only one master copy for each object. All other copies are only slave objects - also called replicas - and they need to communicate with the master copy when changes are to be made. If a node wants to update the state of an object of which it only owns a replica, the replica sends the update request to the master object. The master object then validates the change, applies it locally and propagates the update to all the slave copies of that replica. The node which has the master copy of the object, is said to “own” the object.
This scheme of updating the objects, ensures that even if concurrent changes are made to the object, they are serialized at the master objects before being applied. Therefore, the state of the any object is always controlled by the master object, and the object remains consistent even if concurrent attempts to make changes are made.

3.1.2 Distribution of Objects

For virtual worlds that are implemented on top of replicated objects, the distribution of the replicas in the system is handled by a separate process which decides which nodes of the system require a copy of the object. This process (called interest management), maintains a list of the clients, and their interests [4]. Clients provide their interest parameters to the IM, which compares each object against these parameters and decides if the client needs to have a copy of that object.

If a client requires a copy of the object the master object is replicated as a slave and sent to the client. Similarly, if the client is no longer interested in a object, its copy is deleted, and updates are no longer sent to that client.

3.1.3 Fault Tolerance

Replicated objects allow us to build fault tolerant systems with relative ease. Since the objects are already distributed among multiple nodes in the system, in case of failure of a node, the objects are not completely lost. If the crashed node owned masters of any objects that are distributed to other nodes as replicas, the current state of the object is preserved in these replicas. In order to recover full functionality of the objects, we can promote one of these replicas to be the new master object.

Since all changes made to the object state need to go through the master object, any requests that were interrupted by the failure will eventually timeout and no
change would be made. Any new update requests will now be routed to the new master object, the nodes making those changes can continue to use the object without even being aware of the change.

3.2 Monitoring Architecture

With the basic design of replication object systems defined we can create a clearer picture of the monitoring architecture that can be developed to work with such systems. Therefore, we, once again present each of the three major components of the monitoring system, but this time with additional details on how these components can be designed using replicated objects.

3.2.1 Collectors

Collectors need to collect all the information about the virtual world from the servers. In the case where the virtual world uses replicated objects, this information is stored at the servers as individual objects. Since the distribution of these objects between the servers and the clients is already achieved using the replicated object scheme, we can design the collectors to present themselves as clients. The collectors connect with the servers and present their interest parameters to the interest management process.

The actual parameters for interest of the collectors depends on the division strategy that is adopted. Considering the case mentioned in the previous chapter, where each collector is responsible for collecting game state information for a specific region; Each collector informs the IM server(s) about the part of the virtual world that it is interested in.
The actual interest in individual objects is taken care of by the IM process running on the server: each object is checked against the region of the collector and if the object is located in that region, a replica is created at the collector. Defining regions of interest, and letting the IM handle the actual objects also allows transfer of objects from one collector to another. If an object, for example a player, moves from the region of one collector to another, no special handling is necessary. The IM will detect that the player object is no longer of interest to the previous collector and delete the copy from that collector. At the same time it will then recognize the object as being of interest to the new collector and create a copy there.

3.2.2 Filtering

The previous chapter [section 2.2] mentioned that the filtering process takes related portions of collected information and summarizes them into a single segment of information which is then assigned to a monitored object. With the replicated object model in mind, we can define these segments of information more clearly as the individual objects present in the system. Since most objects in a virtual world have a position within that world, we can also define sets of related objects to be ones that are spatially close to each other.

Since each segment of information is now an object, we can define the filtering process as mapping a number of world objects to a single monitor object (MO). The information of all the replicated objects that are mapped to a single MO is merged together to form the summarized information. The actual process of the merging of information is still dependent on the type of objects in question. For example, and as mentioned previously, grouping multiple players that are close to each other, can be
as simple as averaging their positions and presenting a single group of players in that location. However, if we consider mapping some player objects and some other types of static objects to the same MO, then this scheme would not make much sense.

3.2.3 Presenting

As mentioned in the previous chapter [section 2.3], the presentation client needs the filtered information from the collectors in order to display it to the users.
the collectors store the filtered information in form of monitor objects, we can use the replicated objects architecture to transfer these objects to the presentation client.

The monitor objects are defined as replicated objects, and their master initialized at the collector that owns them. The presentation client registers its interest in all monitor objects with the IM, which then takes care of creating replicas of all existing and any newly added monitor objects.

Since all the changes made to the monitor objects are done by the algorithms running at the collectors, it is ideal to have the monitor object masters at the collectors as well. This prevents the need to perform expensive remote calls whenever a change needs to be made to the monitor objects.
In order to adjust the filtering algorithms, we need information about the current area of the virtual world the user is interested in, and his preferences in terms of the type of information required. We can use an architecture similar to the monitor objects to achieve this. The presentation client initiates a replicated object called ViewState, which is updated with the portion of the world the user is currently observing, and any other preferences the user might define.

The collectors, express their interest in this ViewState object, and the IM takes care of creating a replica on all the collector nodes, and of updating it whenever the user changes some of the parameters. Similar to the monitor objects, since all the changes made to ViewState are done at the presentation client, it makes sense to have the master object there as well, and create replicas at the collectors.
Chapter 4
Implementation of the Monitoring System in Mammoth

Using the Mammoth MMO game as the testing framework, we have developed an implementation of the proposed monitoring client. We use this implementation for testing the validity and performance of the monitoring system, which is presented in the next chapter. This chapter describes how the system was implemented given the architecture of Mammoth. The first part of the chapter presents an overview of Mammoth and its architecture. Then in section 4.2 we move on to describing how we build our monitoring system on top of Mammoth.

4.1 Background of Mammoth

Mammoth [13] is a massively multiplayer online game, developed in Java, which implements a virtual world. The major purpose behind Mammoth is to provide researcher with a testing framework, which they can use to run experiments. With this goal in mind, the architecture of the game consists of a set of loosely coupled components, each of which provide some of the functionality required by the overall virtual world. This separation of functionality into difference components allows the developers to add, remove, or change any of these components without having any effect on the rest of the game.

In order to provide a consistent interface between the different services, each component is either defined as an engine or a manager. The engines are completely
independent, and hence, can be replaced with an alternative implementation providing the same functionality. Managers handle multiple implementations of the same service, and allow the use of any number of these implementations at the same time.

Mammoth allows hundreds of users to connect to this world using a java client. Each user is assigned a virtual player called an avatar which the user controls to explore the world, and to interact with game objects and other players.

4.1.1 Replicated Objects

Mammoth uses the replicated objects architecture introduced in the previous chapter to encapsulate the state of the game objects and players in the virtual world, and to distribute this state to all game nodes. Each player avatar and other object in the virtual world is a replicated object and each object has a single master object that can reside in any of the nodes in the system.

Any other player interested in that object acquires a replica of that object which can be used to read and change the state of that object. In order to control where
the master objects reside, the game introduces the concept of trusted clients. These are clients which are trusted not to cheat, and only these client can host the master of any of the objects.

4.1.2 Interest Management

In order to handle the replication of the objects to multiple nodes, a separate process called interest management is used. For each of the connected nodes, the interest management (IM) determines the objects that are of interest to them and creates replicas of those objects at those nodes. IM also determine if the nodes have any replicas that are no longer interesting to them and removes them [4].

So that IM can check the interests, each node provides a match function, which can be called with different world object as the parameter. The function returns true if the node is interested in the object passed to it or false otherwise. The IM uses this match function to determine the interest for each player and node in the system.

As an example consider the match function provided by the client used by the players. This match function determines the current position of the player and the distance from the passed object. If the object is within a certain distance from the player then it is considered to be of interest.

The interest management process is handled by the servers, which are separate machines in the system. The use of separate server is made necessary by the fact the none of the clients possess complete knowledge of all the objects in the virtual world, and therefore are not suitable for IM.
4.1.3 Cells and Servers

The virtual worlds in Mammoth are divided into contiguous portions which are called cells [7]. Each cell is assigned to a separate server, and interest management for all the objects within that cell is handled by that server. To allow for smooth transition of players when moving from one cell region to another, each server is also aware of the objects just outside the borders of its cell.

The cells are further divided into smaller regions called tiles. These tiles are used in multiple components of the system, such as, path finding and load balancing [subsection 4.1.5].

4.1.4 Communication Architecture

The separation of the world objects, clients and the interest management allow Mammoth to be flexible in terms of communication architecture used. It can be configured to run with a client/server, p2p or fully connected structure.

- In a client/server structure, all the communications between different components of the game are routed through a central hub, which redirects each call to the proper recipient.
- In the p2p architecture, the communications are forwarded from node to node until it reaches the intended target.
- In the fully connected mode, each node in the system can directly connect with any of the other nodes with which it needs to communicate.

Figure 4–2 shows examples of client/server and peer-to-peer network topologies. Each of these different structures are implemented in the network engine of the game. The network engine provides transparency to the higher functionality of the game,
and therefore, the choice of communication architecture does not effect the running of the game itself.

4.1.5 Load Balancing

Due to the dynamic nature of the virtual world - where players and objects can frequently move from one cell to another - load balancing is very important. Load balancing in mammoth ensures that the load of interest management is divided equally among the servers [7].

During running of the virtual world, the players may move from one cell location to another. This causes the new cell to be heavily loaded with game related objects, while the previous cell has relatively fewer objects. The server handling the new cell also suffers from this increase in the load. In order to transfer some of the load back to other servers, parts of the cells are transferred. This transfer of cells ensures that interest management of any objects within those parts of the cells is also transferred between the servers.

In order to achieve this transfer of parts of cells, the cells are divided into smaller triangles. The load balancing algorithm calculates the total load in the system as
well as the ideal load per server. The threshold load - at which the load balancing
kicks in - is set to be slightly higher than the ideal load. If a server is found to be
above the threshold load, some of the edge tiles are transferred to the adjacent cells.
This process is repeated until the load of all servers is within the threshold limit.

Mammoth provides a flexible interface which can be used to implement different
load balancing algorithms for selecting the tiles for migration, and the rate at which
these tiles are transferred.

4.1.6 Fault Tolerance

When hosting a virtual world, with a large number of active players, failure
in the system need to be handled to ensure continued working. Fault tolerance
mechanisms are used to ensure that in case of such failures no part of the state of
the virtual world is lost.

Since the master objects are spread throughout the different nodes in the system,
a crash may cause some of these to be lost. In such a case the replicas of those objects
also act as the back up, and one of them can be promoted to the new master object.
The replicated object in mammoth are designed with this functionality in mind, and
all replicas hold sufficient information to be converted to master objects.

A crash of one of the interest management servers, has a more pronounced effect
on the running of the world. Any object or player in the region of the crashed server
will instantly lose all IM functionality. However, since the cells in mammoth are also
replicated objects, and all servers keep a replica of all the cells, a failure of one of
the servers can also be handled by promoting one of the other replicas of that cell to
the master. The new master cell is then under the control of the new server, which
will transfer it to other cells through the load balancing algorithms, or merge it with its own cell.

4.2 Implementation of the Monitoring System

This section presents how we have used the Mammoth architecture [section 4.1] to implement the monitoring system. Many of the architectural properties of Mammoth, such as replicated objects, its load balancing and fault tolerance mechanism and Interest Management are utilized to create an easily integratable monitoring system. We use this implementation of the monitoring client as a proof of concept of our proposed architecture, and use it for running experiments [chapter 5].

For this implementation we chose to limit the system to monitoring only the player positions and density in the virtual world. Player position was chosen because it forms the very basic set of information present in any MMO game. Using this monitoring system functionality, designers can observe problems with the game world, and understand user behavior patterns. This information allows the designers to understand the preferences of the users, and can be valuable when improving the existing worlds, or introducing new content.

With potentially hundreds of players using the game, tracking all player positions will create a large amount of network traffic between Mammoth and the monitoring system. It is not feasible for a single node to handle all this traffic and, therefore, we need to divide the work of tracking the players between multiple collectors. Displaying all this information to the user, without filtering, is also unsuitable for being observed at real-time. We therefore need to filter this information at the collectors before forwarding it to the presentation client.
4.2.1 Collectors

Since Mammoth already has constructs that allow the clients to get information about the world objects from the servers, the collectors are also designed as a different kind of clients. In fact, to the server and the interest management system, they present themselves as normal clients. The advantage of this approach is that no changes need to be made to the game itself.
Furthermore, since Mammoth already divides the virtual world into separate cells, we use these as the division of information between multiple collectors. All the collectors try to divide the cells in the game as equally as possible, and handle all the players within the cells that they are responsible for. This reuse of cells as divisions between the collectors allow us tap into some of the Mammoth’s already existing functionalities such as load balancing and migrating objects from one collector to another.

To keep track of the division of cells between the collectors, and to allow the interest management to access this information, a new replicated object - called collectorObject - was created. Each collector initiates a new instance of this object, and stores a list of cells that it is monitoring within it. This collector object is then replicated to all the other collectors, and to each of the IM servers.

In-order for interest management to work, a new match function, is also developed. This match function checks the list of cells in the collectorObject for the collector and checks if the given player is within one of those cells. If the player is found to belong to one of the cells, a replica of the player is created at the collector using the normal game procedure. Similarly, if a player is found to have moved out of the region of interest for the current collector, it is removed.

Using the above mentioned mechanism, the collectors are able to gather all the players within the given region, and also update this list of players periodically. At the same time they also filter the results at given time intervals.

Some of other important design and operating properties of the collectors are described below.
4.2.1.1 Collector Start-Up

The first step for the collector start-up is to obtain a replica of the ViewState object by announcing its interest and waiting for IM to create a copy at this node. The start-up procedure can only proceed if the ViewState object is present. This object allows the presentation layer of the system to send user interests to the collectors. The ViewState object is discussed in more detail in the following sections [subsection 4.2.3].

Once theViewState object is found, a new collectorObject is created and registered with the replicationSpace so that it can be replicated on the other collectors and servers. CollectorObjects of other already started collectors and all the cells present in the game are searched, and the number of cells that need to be transferred is calculated. The number of cells is calculated as $\text{cells/collectorObjects}$ rounded down to the last integer. The number of collectorObjects also counts the object created by the starting collector itself. This number of cells are then transferred from other collectors, starting with the one containing the highest number of cells.

All collectors also construct a ring where each collector calls an isAlive method of another collector’s collectorObject. This ring is used in order to provide fault tolerance [subsection 4.2.1.3]. Therefore, it is necessary that the replication is continually search for new collectorObjects and cells, even when the start-up has finished and the normal operation is on going.
4.2.1.2 Load Balancing at the Collectors

Since Mammoth already provides load balancing of the cells by transferring tile between them, the load balancing for the collectors is very simple. In fact, as long as equal number of cells are handled by each collector, the underlying load balancing mechanism will ensure that each cell and therefore each collector has equal load.

4.2.1.3 Fault Tolerance at the Collectors

In order to ensure that the monitoring application can continue working in case of failure of some of the collectors, we introduce fault tolerance mechanisms into our architecture.

All the collectors currently running in the system form a ring, where each collector periodically checks the next collector for failure. The check for the failure is done by making a simple remote call to that collector objects and waiting for a reply. In case the call times-out several times, the collector is assumed to have crashed.
If a collector has crashed, the cells that were handled by that collector need to be taken care of by other collectors. Since the previous collector in the ring already has the replica collector object, it takes those cells, and adds them to its own collector object.

It is also possible that once a collector crash is detected, the collectors, divide the orphaned cells between them to ensure a more balanced operation. However, we have not implemented this functionality, as we assume that the collector crash would be detected and the collector restarted. In this case the redistribution of the cells is significantly more straightforward if only one collector controls them.

When a new collector is started it checks to see if there are any already-running collectors in the system. If it is the first collector then there is no need to make a ring. However, it continues to periodically check if any new collectors are added.
In case there are other collectors already running, the new collector is added in the ring between the last collector to be started and the very first collector in the system.

4.2.2 Filtering

The major information provided by the monitoring system is the player positions in the virtual worlds. However, since the number of players is potentially very large, displaying each player location individually may not be ideal for the user. If there are too many players in the world, the user might not require the exact information about each player, but might be more interested in player density or the overview of the player movements in the area of the world under observation.

Therefore, we provide a filtering algorithm that functions by aggregating the position of the players that are close together into groups of players. In terms of Mammoth, this is equivalent to mapping players that are close to each other to a single monitor object.

The monitor object itself maintains a list of players that are mapped to it, and calculates its center by using the average of all the player positions that are in this list. Since the presentation client, is not interested in the individual players, this list maintained by the MO is only local, and does not get copied to the replicas.

In order to further reduce the number of remote calls made by the MOs, during a single pass of the algorithm, we only update the list by removing or adding players that have moved. Since this is a local operation no remote calls are made. At the end of the iteration the algorithm calculates the average position using this list and makes a single update call to the replicas.
4.2.2.1 Filtering Algorithm

The filtering is achieved in the Mapper class, which stores a list of all available MOs and all the currently mapped players along with the MO they are mapped to in a HashMap. The filtering algorithm works in two phases.

In the first phase, the viewState object is read to determine the visible area of the world, and the maximum visible distance from the center of the view, \(visibleDist\), is calculated. A Set of all the players in this area is computed and stored in a temporary list, which is then compared with the HashMap. If a player is present in the HashMap but not in the list, it has moved out of the visible region and is removed from the HashMap. Similarly any players that are present in the new list but not in the HashMap are newly visible players and need to be mapped. These players are added to the HashMap without being mapped to any MOs. This algorithm is described in 1.

When players are removed from the current mapping it is ensured that they are removed both from the HashMap, and from the MO list. After each removal we also check if the MO list is empty, in which case the MO is marked as inactive. Inactive MOs are utilized in the next phase of the filtering.

After the first phase, our HashMap only contains currently visible players. The second phase iterates over all these players and maps or re-maps them to MOs. This phase is shown in 2. \(maxDist\) determines the maximum distance between a player and the MO for it to be mapped to that MO, and depends on the zoom level of the monitoring node, as well the number of MOs being used in the collector. We
Algorithm 1 Update View

HashMap map

updateView():
    center = ViewState.center
    zoom = ViewState.zoom
    visibleDist = zoom*constant
    for all players p such that:
        p.center - center < visibleDist
        add to list 'lst'
    for each player p in map:
        if not in lst:
            remove p
        else:
            remove player from lst
    for each player p in lst:
        add to map with 'null' MO

experimented with different values of the constants $c1$ and $c2$ to determine a value that yields the best results.

Before mapping the new players to MOs, all the active MOs are compared in pairs to check whether they are close to each other. If the distance of their centres is less than $maxDist/2$ apart, the two groups are combined into one, freeing one of the MOs.

After all possible MOs have been merged, for each player that is already mapped to a MO, the algorithm determines if the player has moved more than $maxDist$ distance away from the group centre. If yes, the player is removed from the group and flagged to be remapped. A player that is not mapped to any MO, was added to the HashMap in the previous phase. Any such players are also flagged for mapping.
Finally, for each player that needs to be mapped, the \textit{mapPlayer()} method is executed. It checks whether the closest MO is less than \textit{maxDist} away, and if so, adds the player to the MO. Otherwise, if there are any inactive MOs, the player is assigned to a new MO, thus starting a new group. In case there are no more inactive MOs, as a last resort, the \textit{mapPlayer} method is called recursively with an increased \textit{maxDist} value to add to the closest MO.

\subsection*{4.2.3 Presentation Client}

In order to provide the user with a good view of the player or group position in the world, our presentation client provides a top-down view of the world, and therefore, we call it the god client. The god client allows the users to pan around the world to choose the area which he wants to observe. The user can also zoom in and out of the region to change the total viewable area.

The zoom level and the current region of the world that the user is observing are the major parameters that affect the filtering algorithm described in the previous section. The god client updates the ViewState object whenever the user pans around the map or changes the zoom level. These updated values are then sent to all the collectors to adjust the filtering.

\subsubsection{4.2.3.1 Managing MOs}

A list of the monitor objects needs to be maintained and updated periodically at the god client. A separate class called CollectorManager handles this operation. When the god client is started, the CollectorManager class looks in the replication space to gather all the currently existing MOs in the system.
Algorithm 2 Filtering Algorithm

HashMap map

doMapping()
    maxDist = (zoom*c1)/(#MOs/c2)
    for each pair of MOs m and n:
        if (m.center – n.center) < maxDist/2:
            join m and n
        list to-be-mapped
    for each player p in map:
        if p is mapped to an MO m:
            if (m.center – p.center) > maxDist:
                remove p from m
                add p to to-be-mapped
            else:
                add p to to-be-mapped
    for each player p in to-be-mapped:
        mapPlayer(p, maxDist)

mapPlayer(p, maxDist):
    find MO m closest to p
    if (m.center – p.center) < maxDist:
        add p to m
    else if free MOs exist:
        add p to a free MO
    else:
        mapPlayer(p, maxDist++)
These MOs are stored in an ArrayList, and the manager checks to see which of these MOs are active. An active MO is defined as one that has at least one player mapped to it. Only these active MOs are displayed to the user.

The manager also periodically checks if new MOs have been added to the system, in case new collectors are started. If any MOs are found those are also added to the list.

4.2.3.2 Further Merging of the MOs at the God Client

Since using more collectors, also increases the number of monitor objects that are used during the filtering, the accuracy of the information sent to the god client is also improved. However, this accuracy is based on the number of collectors and therefore monitor objects that handle the part of the world currently under observation.

If a user is observing a part of the world that is handled by a single collector, the maximum number of monitor objects used during filtering will be equal to the maximum number of monitor objects per collector. If the user was now to move to a part of the world, where the viewable region is divided among several collectors, the maximum number of MOs used would also be higher.

Figure 4–6 illustrates this concept, where the virtual world is divided between two collectors, with 5 MOs each. The dashed box represents the current region the user is viewing, and the circles represent the MOs as displayed by the god client. In figure 4–6a the user is viewing a part of the virtual world only covered by a single collectors, and therefore, only 5 MOs are displayed. Figure 4–6b shows the effect
of moving to a collector boundary without any merging at the god client and figure 4–6c shows the same effect if the merging at the god client is enabled.

While this increase in the number of MOs provides an increase in the accuracy, the effect of panning the view from region of one collector to another provides an inconsistent accuracy during the move. While the view is in the region of one collector, \( n \) number of MOs are used. As the user moves to the edge of the region,
a higher number of MOs come into play since the viewable area is divided among the collectors. As the view is further panned back into a single collector region, the accuracy drops again. Besides these sudden variations in the accuracy, the increase and decrease in number of MOs displayed also effects the visual consistency for the user.

In order to cater for this behavior, our implementation allows the users to select if there should be another round of filtering done at the god client. If selected this merges active MOs to ensure an equal number of groups displayed irrespective of the number of viewable collector regions.
Mammoth [13] is a MMOG research framework designed for experimentation in an academic setting, which uses replicated objects as the main way of communication between the nodes. Any node in Mammoth can express interest for a replicated object, in which case it will receive a replica of the object from the node that owns the master. Mammoth also offers a modular and flexible infrastructure for the definition of non-player characters with behaviour controlled by complex artificial intelligence algorithms, as well as logging and testing facilities that make it possible to collect detailed measurements when running real-world experiments involving hundreds of machines.

We have implemented our monitoring approach in Mammoth. The experimental results presented in this section have been obtained by running experiments on over 100 lab machines of the School of Computer Science of McGill University. The lab machines run Linux, and are equipped with Intel dual-core 3.06 GHz processors and 4GB of RAM. We used our Wanderer NPC to substitute human players, as it is shown in [4] that measurements obtained with computer-controlled players performing random actions can approximate measurements of games played by real humans, provided that the starting positions of the random players are chosen adequately.

Since we were not able to access enough machines to run thousands of players, the experimental results presented here were obtained with around 100 players and
unless otherwise stated, using a single collector node. Nevertheless, we were able to obtain results that give interesting insight into the performance of the monitoring architecture.

5.1 Delay due to Redirection

The gathered information, in this case player positions, goes through many stages before it reaches the monitoring node. Any change in position on the player node is broadcast to all replicas of the player object, one of which is at the collector node. On the collector, the player position is mapped to a MO, and the group state is updated. That change is then forwarded to the MO replica located on the monitoring node.

This indirection introduces a delay. The delay can be divided into the time taken due to additional communication and the time spent in remapping. While we deal with the latter in the following experiments, this first experiment was conducted to determine the delay caused by the additional communication involved. Concretely, we instrumented a monitoring node to register for a specific player replica. That specific player was tagged on the player machine, and the collector node was instrumented to also tag the group MO the player is mapped to. This makes it possible to measure on the monitoring node the time delay between receiving the original position update from the tagged player replica, which is sent directly from the player node to the monitoring node, and the update received from the tagged group MO replica sent from the collector node. To measure only the delay due to communication, we configured the collector node to propagate position changes immediately.
We repeated the experiment in a world with 1, 30 and 100 players. Table 5–1 presents the results of the tests in milliseconds, and it can be observed that there is a slight increase in the delay as the number of players increases. Since the values presented in the table are the average of 20 samples recorded during the experiment, the variation in network delay is considered to be minimal. The increase in delay is caused by the collector having to execute the mapping algorithm for a higher number of players, and because of network collisions due to the increased number of sent messages.

According to the measurements, the delay introduced by the collector nodes is negligible. It has to be pointed out here, however, that the collector and the monitoring node were located on the same local area network when the measurements were taken. In a WAN setting, the delay introduced by the network connection is expected to increase accordingly.

### 5.2 Limiting of Update Messages

The second experiment was designed to experimentally verify the fact that our architecture limits the network bandwidth required to send updates to the monitoring node. To show this, we ran three separate experiments: one with a monitoring node that is directly connected to all players; one where position updates are filtered by a collector that has 5 MOs, and one where position updates are filtered by a collector

<table>
<thead>
<tr>
<th># of Players:</th>
<th>1</th>
<th>30</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay(ms):</td>
<td>29</td>
<td>35</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 5–1: Delay due to Redirection
with 8 MOs. For the tests involving collectors, we configured the mapping algorithm to run every 0.5 seconds.

Each experiment was started in a virtual world that contains 100 non-moving players. Then, every 10 seconds, a Wanderer NPC would connect and take over the control of one of the immobile players. The movement frequency was set to an average of one change in direction every 5 seconds. Each change in direction results in an update message sent from the node running the Wanderer NPC to the monitoring node (experiment 1) or the collector node (experiment 2 and 3). As the players would start moving, we measured the number of update messages that were received on the monitoring node. The update count was taken every 5 seconds then averaged to compensate for the randomness of the player movements.
The measurements for this experiment are shown in figure 5–1, which plots the number of moving players against the average updates per second. The blue line labelled “Direct Update” represents the results for experiment 1. As expected, the message count on the monitoring node increases almost linearly as more players are added. Furthermore, the number of update messages is approximately 1/5th of the number of moving players, which corresponds to the 5 second average delay between change of direction that was set for the Wanderer NPC.

The results for experiments 2 and 3 are shown in green and orange, respectively. In this case, the received update messages on the monitoring node are coming from the collector, and therefore contain filtered data. The numbers show that in both experiments initially the number of messages received by the monitoring node increases twice as fast as for experiment 1. This is explained as follows: when a moving player that was first mapped to one MO group object is reassigned to a different MO, the states of both MOs change and hence two update messages are sent to the monitoring node in the following time slice. Since the players were distributed uniformly across the world, the maximum number of MOs contain moving players very early on during the experiments. However, as more players start moving, MOs are being shared, which slows down the increase in the number of update messages. Once a sufficiently large number of players are connected (~50 in our experiments), each MO changes state at each iteration of the mapping. However, the maximum number of updates is limited by the number of MOs divided by the update delay between iterations. Since for both experiments the update delay was fixed at 0.5 seconds, the maximum number of update messages that is ever generated is twice
the number of MOs, i.e. 10 messages per second for experiment 2 with 5 MOs, and 16 messages per second for experiment 3 with 8 MOs.

5.3 Error in Player Position

Our third experiment investigates the inaccuracy of player positions reported on the monitoring node due to the mapping of multiple players to a single group MO and due to the time delay. The experiment also shows how to determine the optimal tradeoff between #MOs and update frequency.

We ran several experiments, some on a map with 30 players and some on a map with 100 players, using different zoom levels at the monitoring node, i.e. 15, 9 and
3. For our test map, zoom level of 15 corresponds to a full view of the map, and a zoom level of 3 is very close to the ground. We ran the experiments using different configurations of the collectors, but making sure that the maximum used network bandwidth was constant at 10 messages / second for every experiment. We therefore used: 5 MOs with 0.5s update interval, 10 MOs with 1s update interval, and 20MOs with 2s update interval. We instructed our Random Wanderers to move on average every 3 seconds at a speed of 0.6 units / second.

We measured the average error in player positions, i.e. the difference between the actual player position and the position of the centre of the group to which the
player was mapped. In the case where a player should be visible on screen but was not mapped to a MO yet, the error was set to be the maximum viewable distance.

The results are shown in figure 5–2. The first fact to observe is that difference in the average error in all experiments does not depend on the total number of players in the map. The reason for this is that the size of the MOs in terms of the area of the map covered is roughly the same in both cases, and only the number of players mapped to an MO differs. But since we take the average of all the errors in player positions, the number of players in the MO is not relevant.

We also observe that within each group of measurements for a given configuration, the average error increases as the zoom level increases. The difference in accuracy between the closest zoom level (3) and the full map view (15) is the biggest for the 5 MO configuration. This makes sense, since at the closest zoom level there are often just very few players visible, and hence 5 MOs with fast update yields the best results. In contrast, 20 MOs are not ideal at the closest zoom level, which is clearly shown by the high average error reported at zoom level 3 for the 20 MO configuration. In this case, most MOs are not used, and the reported error stems mostly from the slow update interval. However, having more MOs pays off at higher zoom levels. In our experiment, the 10 MOs with 1s update interval seemed to be performing best at full world view.

Although our experiments show that the error increases as the view is zoomed out, the actual perceived error on the monitoring node is the error compared to the view size. Figure 5–3 plots the ratio $\text{error/visibleArea}$, i.e. the perceived accuracy,
for the 5 MO, 10 MO and 20 MO configurations, with 100 player map. The results clearly show that error ratio in general is lower for higher zoom levels, since the movement of players with respect to the visible area of the world is small. It also shows that for a close zoom level, the shortness of the update interval is most important for achieving high accuracy. For higher zoom levels on the other hand the number of MOs becomes important too. However, 10 MOs and 1s update interval outperforms the 20 MOs and 2s update interval configuration, which suggests that at some point the error resulting from slow updates cancels the benefits of having more MOs.

5.4 Error in Player Position Before and After Merging

Figure 5–4: Error/Zoom - Multiple Collectors, Merging at the God Client
As described in section 4.2.3.2, when using multiple collectors, the user can choose to do further filtering once the data has been received from the collectors. This experiment was conducted to demonstrate that the accuracy achieved after the second pass of filtering is identical to the accuracy achieved by a single collector, provided the number of players in both cases is the same.

For this experiment, we ran four interest management servers, on four separate lab machines. The virtual world was equally divided among the four servers. 100 non-player characters (NPCs) were added to the world, and programmed to move around the world randomly. Two collectors were started, each responsible for two cells, and the filtering algorithm was set to use 10 MOs and run at 1 second intervals. The merging at the god client was designed to reduce the number of MOs to 10. Player position accuracy was checked at each of the collectors, and at the god client. At both locations the accuracy check was done right before running the mapping or merging algorithms to ensure that the worst case for the errors was measured. The error was calculated as the percentage of total visible area.

The experiment was run twice, and each time the readings were taken at zoom levels, 3 (close zoom), 9 (medium zoom), and 15 (high zoom). In the first run of the experiment, it was ensured that the players were almost equally divided between the collectors, and the view of the world visible on the god client covered regions of both the collectors. The results for this run are shown in figure 5-4 with the solid lines. The red and blue lines indicate the percentage error at each of the two collectors, and the yellow line shows the error after merging at the god client. As expected merging MOs increases the percentage error. The error before and after merging for
zoom level 3 remains more or less equal because at such close zoom, very few players are visible and so not all the MOs are active for each of the collectors. Since the number of MOs active are actually less then 10, no merging takes place at the god client, and so there is no increase in error.

For the second run of the experiment, it was ensured that a single collector covered all the 100 players in the world, and the area visible to the user. Since only a single collector was in use, the number of active MOs being used was also reduced to 10 and therefore, no merging was done at the god client. The results for this part of the experiment are shown in figure 5–4 with the Grey dashed line.

The experiment results prove that using multiple collectors, and then merging the MOs from all the collectors, does not impose any increase in error as compared to using a single collector. While the error might be lower if no merging is done at the god client, the number of MOs visible to the user at the god client will not be consistent and the accuracy will vary greatly depending on how many collector regions are currently visible.

5.5 Error Due to Load Balancing Delay

Section 4.1.5 describes how the load balancing works in Mammoth. The monitoring system uses this underlying loadbalancing to balance the number of players between the multiple collectors, as described in section 4.2.1.2. This experiment is designed to validate our assumption that the Mammoth load balancing is sufficient to ensure balancing at the collectors as well. Furthermore, it also shows how merging the MOs at the god client improves visual aspect of the god client, by limiting the number of MOs visible to the user, and the variations in accuracy.
For this experiment, Mammoth was set up to use two interest management servers, and 30 NPCs were added to the world. We were able to control the NPCs externally by sending them text commands such as \textit{move x y} or \textit{flock x y s}, where \textit{x} and \textit{y} are world coordinates. The flock command moves each NPC to a randomly chosen point a maximum of \textit{s} units away from \textit{(x,y)}. Two collectors with 10 MOs each and a god client were started, with each of the collectors handling one cell. The filtering was done every one second.

To start the experiment the players were ordered to move to the border of the two cells with a spread of 3 units, such that an equal number of players were located in each of the cells. Error calculation was started on the god client, and players were ordered to move to a region completely covered by a single cell, while maintaining a spread of 3 units.
Since the move changed the load on each of the cells, load balancing algorithms started transferring tiles away from the over loaded server. We continued calculating the average error in player position until the load on each cell was balanced.

The experiment was done twice, once such that the god client merged the MOs to 10, and once without any merging. The red line in figure 5–5 shows the results with merging enabled and the blue line shows the results without merging.

For the experiment with merging disabled, the starting error is low because all 20 MOs are being used. However, as the players move to a single cell, the error rises sharply, since only 10 MOs are now used. As load balancing moves tiles with some of the players back to the other cell, the error gradually reduces until it is similar to the starting point. These results confirm our hypothesis that load balancing of mammoth will also take care of the load balancing for the collectors.

For the second experiment, we observe, that even while the players have all moved to a single cell and we are waiting for load balancing to finish, there is no significant change in the average error. We also notice that while the players were in the single cell, the error was the same with and without the merging. We use this result to back our claim that while the merging reduces the accuracy, its has much more consistent visual effects.

5.6 Scalability

Due to the limitation of Mammoth the experiments presented above were conducted on a relatively small scale as compared to many of the massively multiplayer virtual worlds. These virtual worlds can experience loads ranging in thousands of players, while our experiments were limited to a fraction of that number. In this
sub-section we analyze these relatively small scale experiments and show how they still give us results that are applicable for realistic virtual worlds, and how we can project the expected performance of our system in the real world.

5.6.1 Delay due to redirection

As we mentioned in the section 5.1 this experiment was conducted to only determine the delay caused in the update of an object on the monitoring client due to the additional communication involved in the redirection of information. Looking at table 5–2 (which is copied from section 5.1) we can observe that the actual number that we are interested in is the delay when only one player is involved. The linearity of the increase in delay in the next two sets of result is only used to strengthen our assertion that the additional delay caused when more players are involved is actually due to the higher number of computations that need to be done at the collectors.

5.6.2 Limiting of Update Messages

In this experiment [section 5.2] we attempt to prove that the number of messages between the collectors and the monitoring client are indeed limited to a fixed higher bound. Our choice of number of players, the number of MOs and the update frequency were selected according to the limit of our set-up. In real world virtual worlds with higher number of players, these variables will need to be adjusted differently. In fact, these number will be dependent on the amount of traffic possible.

<table>
<thead>
<tr>
<th># of Players:</th>
<th>1</th>
<th>30</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay (ms)</td>
<td>29</td>
<td>35</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 5–2: Delay due to Redirection
between the monitoring client and the collectors, and the level of detail that the user of the monitoring client requires.

### 5.6.3 Error in Player Position

In our third experiment [section 5.3], we show how the grouping of players into monitor objects - based on proximity - effects the error between their actual and the reported positions. The three separate charts in figure 5–2 show the results for the different combinations of number of MOs used and the interval between updates to the monitoring client. Within each bar chart, the first two bars show the error for zoom level 3 (close zoom) for 30 and 100 players respectively. The next two bars show the results for 30 and 100 players in the same order for the zoom level 9 (medium zoom), and the last two bars for the zoom level 15 (wide zoom).

From the figure we can observe that the each pair of bars that represents the same zoom level within the chart, the difference between the results of 30 players and 100 players is very small. We use this result to claim that the actual error in the player position will not vary significantly with the number of players involved provided that the number of MOs, and the update interval are kept constant. The fact that the average error remains almost constant can also be seen by the fact that given the number of MOs is the same, they will cover about the same area of the visible map and therefore, the distance between the center of the MO and the furthest player will be similar irrespective of number of players. This, in turn, means that average error for that MO will remain constant.
5.6.4 Other Experiments

The remaining two experiments were conducted to observe the effect of some additional properties and functionality of our system on the average error of player location. The first of these two experiments [section 5.4] observes the change in average error due to the merging of MOs at the monitoring client, and the second experiment [section 5.5] observes the effect of load balancing. Since both these experiments compare the effect of specific but external changes to the system, we claim that the number of players involved does not affect the obtained results. Due to the comparative nature of these experiments we always provide a benchmark result which is obtained in exactly the same testing conditions but without the external effect in question.
Chapter 6
Related Work

Monitoring distributed systems in general is not a new concept. In fact, there have been a large number of proposed ideas for such systems. However, we did not find any work that dealt specifically with monitoring virtual worlds. As such this section lists some of the proposals in the field of monitoring distributed systems that we used as inspiration for several important decisions for our system.

6.1 Hierarchical Structure

The basic architectural design of our monitoring system is based on a hierarchical structure, where each level of hierarchy reduces and aggregates data so that the volume that needs to be processed by higher layers is reduced. Our system also provides a way for the user to control the overall system properties such as what type of information to collect and the level of detail after the filtering. Some other monitoring systems that share some of the designs and technique with our proposed architecture are mentioned in this section. In the last subsection we present a paper that combines these each of these design ideas into a complete monitoring system for distributed and cloud applications.

6.1.1 Hierarchy of Collectors

The Chukwa monitoring system as discussed in [19] was developed as means of collecting and analyzing very large amounts of monitoring data produced by the
hadoop distributed file-system and map-reduce system. The amount of data produced by a large node clusters using hadoop is in the scale of hundreds of Gigabytes everyday. Most of this data is simply stored as log files which are less then an ideal situation for collecting this data from each node and analyzing and displaying it using scripts or programs. Chukwa provides such a solution by acting as a middleware between the data producing nodes and the final collective storage of the information. Its architecture allows for data collection from a very large cluster by using a tree based structure where the nodes of the tree collect data from multiple lower level nodes, combine it and send it to higher level nodes for processing.

Adapters are used as plug-able, data generating components on the nodes which are being monitored and wrap the actual data source such as log files or some Unix command-line tool. With the help of these adapters, the monitoring system can change the matrices that need to be analyzed at run-time, by simply replacing the adapter with another one. Collectors and agents are used as as middle layers between the adapters and the storage to allow these two components to be decoupled. Agents, run on the monitored system and use the adapters to collect data from the individual system and send it to the collectors. Collectors receive data from multiple agents and write it to a big sink-file along with the metadata. These files can later be read for analysis.

A 2000 node cluster of hadoop generates about 6 megabytes of data per second that needs to be stored on a file-system, and then accessed for analysis. The FS should offer scalability and high data read and write speeds. Chukwa uses the hadoop HDFS for this purpose.
This solution, however, only addresses the data collection portion of our requirements. If the users of Chukwa need to change the type of information collected they need to replace the adapters that are being used on each of the distributed node. Furthermore, the collected data is simply unified and stored at a single location. The users of the monitoring system can later access this data and analyses the monitored system. For monitoring virtual worlds, we require a quicker method of control over the type of information collected and a more real-time method of displaying the data for the observer.

6.1.2 Single Point of Control

A better solution for controlling the type of data collected has been presented in the paper “A Scalable SNMP-based Distributed Monitoring System” [20]. This paper talks about implementing a distributed monitoring system using SNMP protocol. SNMP is the standard remote monitoring protocol implemented for most modern day operating systems and can be used to monitor the system health using parameters such as CPU and RAM usage of a process or the network traffic of the system. The author has implemented a stand-alone monitoring system based on SNMP and in this paper he discusses how it can distributed with minimal changes.

The paper introduces a hierarchical structure for the system with three different types of components. Top-level manager(TLM) is the central control for the whole system and the users interact with it to define the type of information that needs to be monitored. TLM controls a number of intermediate-level manager(ILMs) each of which can be assigned a monitoring task and have a number of agents that report to it with the actual data. These agents run on the system nodes that need to be
monitored and collect the appropriate type of data for that node. The TLM defines a task to each of the ILM and sends it an appropriate script (using any language) with which this task can be achieved. When some information is requested from the ILM, it gathers the data from the lower level agents, executes the provided script to generate a metric depending on the request and sends it back to the TLM. The actual data that is gathered and the metric that is calculated is dependent on the script provided to the individual ILM, and can be simply changed by providing it with a different script.

The resulting architecture of the system is a tree, with agents as the leaf nodes, and TLM as the root node, and allows the system to scale depending on the load and available resources. The scripts mentioned in the paper can be used to filter and reduce the data that is received from the agents before sending it to the TLM.

The use of SNMP as the protocol used for collecting data from the system limits the type of information that can be collected. A virtual world has many different types in data that might not be suitable for such a predefined protocol. However, this paper does present us with a solution of controlling the type of data collected - by replacing the scripts that are run at the ILMs. Besides this control of information, the use of a TLM as the single point for managing what is collected and how it is processed is also shared by our system.

6.1.3 Filtering of Collected Data

The above mentioned technique caters for systems that already support SNMP protocol and builds on this protocol to create the monitoring framework. The ganglia monitoring system mentioned in [16] is designed for large scale clusters and grids,
and offers a more flexible set of controls. It also introduces the concept of aggregation points which act as data filters and convert the data into more concise metrics. It also proposes a tree based architecture where each leaf node represents one of the monitored machine in the system, and each higher level node represents an aggregation point. Periodically or when required the aggregation points query multiple nodes to gather information, filters and converts the information into a more concise format and the response is sent as XML over TCP, to higher level nodes.

Ganglia uses three separate programs/daemons to function. One type of daemon (gmond) runs on all the individual machines in the monitored system, gathering performance metrics and sharing it with all the machines in the cluster using multicast. A another program (gmetric) allows the gathering of application specific data using a simple command-line interface. The third type of daemon (gmetad) runs on the aggregation points which request data from gmond and aggregates for forwarding to the higher level, or stores the data.

Another mode of filtering information is presented in [2]. The monitoring system presented in this paper consists of monitor agents which gather information from the application process and forward it to the distributed system monitor (DSM) agent. Both the monitor agent as well as the DSM agent run on the same node as the application process, and the DSM agents distribute the gathered information to all other DSM agents running on remote machines.

Since the DSM agents need to send the collected data to remote DSM agents each time a new value is collected at the local node, a large network overhead is created. Therefore, instead of filtering after gathering the data from multiple nodes,
the paper proposes a filtering mechanism which takes place at the application nodes before the data is transmitted over the network.

The filtering mechanism itself that is adopted in the paper is also very interesting. The DSM checks each new update to determine if it is significantly different from the last update. In case the change in value is small enough to be ignored, no new update is sent to other DSM agents.

6.1.4 A Combination

REMO is a distributed monitoring system presented in [17] which combines the design ideas presented in the previous subsections - i.e., a hierarchical structure, central control and processing of results - to create a complete distributed monitoring system for distributed applications. The focus of this monitoring system is on providing the ability to perform multiple monitoring tasks in parallel while minimizing the resource usage.

A monitoring node is used as the primary data gathering component, and these monitoring nodes are arranged into a tree structure. A central management core is used to arrange these nodes into trees at run-time. This management core is resource aware and creates a separate tree for each of the monitoring tasks. This results in the same monitoring node appearing in different positions of the tree for different monitoring tasks. So the same node might be the root of a tree for one monitoring task while being the leaf node for a different monitoring task. This dynamic arrangement ensures that the resource usage is minimized.

A universal data collector gathers information from the root nodes of each tree and then forwards it to a result processor. The result processor then performs the
required actions on the gathered data such as filtering and aggregation before the final results are presented to the user of the system.

While this monitoring system uses all the same ideas as we require in our virtual world monitoring system, it has been designed for monitoring performance of the applications that are running on a distributed system such as cloud based applications. It is, therefore, missing many of the capabilities and properties, such as the rapidly changing requirements of the users, that are unique to and are required for monitoring of the virtual worlds. For example, if a user of the monitoring system changes the area of the map that is being monitored or the zoom level of the view, management core needs to readjust the tree structure of the nodes to optimize the resource usage. This introduces an additional delay before the up to date information is visible to the user.

6.2 Information Visualization

Displaying the final, filtered information to the user can be achieved in multiple ways. [12] presents a comparison of the two major options that are available for the display. The first option uses plain text based traces to display the interactions and events in the monitored system. The overall effect of such a presentation of information is very similar to a collective log for all the machines in the distributed system, however, because all the information is available at one point the output can be filtered and only the relevant information displayed.

The second approach to presenting information is to use a graphical display, which shows the current events, and interactions that are taking place. Filters similar to the text based approach can also be used. Using a graphical representation can
provide a much easier and concise overview of the system in a more human-readable manner. However, unlike text based representation, graphical display does not retain any history of previously completed events. When using textual representation the user can simple scroll up to review what happened at an earlier point in time. So overall, the choice of display depends on what kind of information is being displayed, and if a historical view of previous events is required.

The Chukwa monitoring system [19] that we discussed in the previous section also presents a portal-style user interface, where the user can add widgets that are like plug-ins and analyze or display some of the matrices. Since the complete information of the system is available in storage, these widgets act as information filters, and display relevant information depending on the requirements of the user. These can also format the filtered information so that it is more easy to understand and presents a clear overview of the system as required.

This solution presents a hybrid approach to presenting data by allowing the retention of older information for later perusal as well as providing a visual output depending on the filtering parameters defined by the user. However, the author of this system argues that the reaction time of administrators of the monitored system, in case of a problem, will be a matter of minutes and not seconds. This assumption, while suitable for many types of distributed systems, is not valid for monitoring virtual worlds. In fact, one of the foremost requirements of our system is the ability of its users to observe the world in real-time.
Chapter 7
Conclusion

The ability of monitoring online virtual worlds can be very helpful for its developers and designers. Such a monitoring system allows its users to observe the world in real-time while the real-world users are connected to it and are interacting with the objects. Some of the benefits of observing the virtual world, using such a monitoring system are:

- Developers can figure out potential problems or failures in the system and respond in a timely manner, before the whole world is effected.
- Developers can also monitor any newly introduced changes to the system to ensure that they are working as expected.
- World designers can observe the game and find out user behaviour patterns, which can be helpful in figuring out new improvements that can be done to the world.
- These user patterns are also helpful to the designers in ensuring that any new virtual world content introduced is in accordance with the user preferences and liking, and hence, it will be well received by the users.

This thesis proposed a monitoring system for the virtual worlds, that helps the developers and designers to achieve the above mentioned goals. Since the architecture of the underlying virtual worlds can vary to a great degree, the general architecture of the monitoring system has been designed with flexibility in mind. Our system
can be adapted for use with different types of virtual worlds, irrespective of their architecture or design.

Besides being flexible, the monitoring system architecture also addresses some of the challenges and requirements of monitoring that are unique to virtual worlds. The major challenges associated with virtual worlds are:

- The amount of information present in virtual worlds is very large.
- This large amount of information needs to be collected in real-time.
- Information needs to be filtered and formatted into a form that makes it possible for users to observe and interpret the data in real-time.
- The users must be able to select the portion of virtual, and type of information, and the level of detail they want to monitor.

In order to cater for the large volume of information, which needs to be collected by the monitoring system, we utilize multiple collectors. These collectors divide the virtual world into multiple portions and each collector gathers information from that part of the world. This division allows each collector to gather information without being overloaded by its overall volume and to remain up to date as the state of the virtual world changes.

These collectors also provide the filtering, which is required, in order to reduce the amount of information that is forwarded to the presentation client and displayed to the user. The presentation client not only displays this information for the user, but also provides an interface for the users to select the location, and level of detail they want to observes. These user preference are communicated back to the collector for use in the filtering algorithm.
As means of testing our proposed system, we developed a sample implementation of the monitoring client on top the Mammoth MMO game. We use this implementation to run different types of experiments to validate our proposal, and to ensure that the system is suitable for real-time monitoring.

With the experiments we were able to confirm that:

- The delay created due to the additional redirection of information through the collectors, does not create a significant delay between the actual and the observe state of the virtual world.
- We can, indeed, control and limit the network traffic between the collectors and the presentation client, by varying the delay between updates and the number of monitoring objects per collector.
- Although the filtering of the information introduces some error in the observed player positions, this error can be controlled by varying different variable in the filtering system.

The filtering system variables present a compromise between the error in the reported state and the computational resources required for the filtering. In order to tune the filtering system for low error we require more processing and network resources and vise-versa.

Overall, our proposed system caters for many of the requirements of the developers and designers of the virtual world, as well as address the challenges faced while monitoring such virtual worlds.
7.1 Future Work

We proposed a general architecture for monitoring systems which can be adapted to many different types of virtual worlds. However, our implementation, focused solely on virtual worlds that were based on the replicated objects, and focused on a smaller subset of the types of information that can be monitored. This section list some further possibilities of research that can extend the work described in the previous chapters.

Multi-Stage Collection and Filtering

While we proposed using a multi-layered hierarchy of collectors to provide scalability to our system, our implementation was limited to using a single layer of collectors. In the future we can research using multi-stage collecting and filtering to determine the effects this has on the final results.

Some of the effects that promise to be of interest include:

- The lag between the actual updating of an object and the update being received at the presentation client;
- The error due to multiple stages of filtering; and
- The controlling of the filtering algorithm depending on the user preferences.

Besides, multi-staged collectors, there is also the possibility of investigating the effects of using separate collecting and filtering nodes.

Alternate Filtering Algorithms

Chapter 2 describes the basic filtering scheme that we have employed for our implementation. The filtering consists of aggregating similar type of data into single monitor objects as means of limiting the network traffic between the collectors and
the presentation client and also to provide the user with the appropriate level of
detail.

Further research can be done in order to come up with other schemes for filtering
information and comparing the results with those of our experiments. The resulting
error between the actual and the displayed location of the world object would be of
prime interest.

**Individual Load Balancing Scheme**

Our current system relies on Mammoth to handle the load balancing of the
servers, which in turn allow the collectors to be balanced as well. However, this
approach limits the maximum number of collectors to the number of servers running.

Furthermore, the load balancing in Mammoth also takes into consideration a
few other parameters such as the load of the master objects the are owned by each
of the servers. This can result in a difference between the load at the servers and the
load experienced at the collectors. With individual load balancing mechanism, we
can increase the number of collectors to be more then the servers and ensure that
the collectors are perfectly balanced.

With individual load balancing, the monitoring system will also be able to func-
tion with virtual worlds where the load balancing is either non existent or unusable.

**Monitoring Additional Types of World Objects**

Our current implementation can be extended to support monitoring of other
system parameters and types of world objects. Currently our monitoring system
only handles the player positions in the world, however, these additional options can
provide the user with a more complete picture of the system. Monitoring additional
types of objects, can provide us a better insight into the performance and adaptability of the system.

Allowing the user to observe system health parameters such as resource usage at the servers, can help detect malfunctioning of the virtual world, and highlight performance issues that need to be improved.
Appendix A
Glossary

**Avatar:** Mostly refers to the graphical representation of the player.

**Cells:** A group of tiles that belong to the same server in Mammoth.

**Client:** The implementation of the user interface that runs on the user machine and allows him to interact with the system.

**Collectors:** A component of the monitoring system which collects the information from different nodes of the monitored system.

**Interest Management(IM):** A mechanism used virtual worlds to determine what each of the player might be interested in and is therefore able to observe.

**Load Balancing:** A technique used in distributed system to ensure that the work load is fairly divided between all the nodes in the system.

**Map:** Same as a world.

**MMO:** Massively Multiplayer Online.

**Monitoring Client/God Client:** The user interface for the monitoring system we developed for Mammoth.

**Monitor Objects(MO):** A special replicated object that is used by the Mammoth monitoring system to transfer data from the collectors to the god client.

**Node:** A single physical and virtual machine which has a component of the distributed system running on it.
**Player:** The representation of the players inside the virtual world. We reserve the word 'user' for the actual human using the virtual world.

** Replica:** A single instance of the replicated object.

**Replicated Objects:** A distributed object scheme which automatically replicates object to different components of the system.

**Server:** A central component of the virtual world system which provides many services such as client initialization, and a more persistent store for the objects in the system. There can be multiple servers which communicate with each other to ensure consistency of the virtual world. Multiple servers allow more users to use the system simultaneously.

**Tiles:** Mammoth divides the map into multiple small divisions called tiles that are used in the IM and load balancing operations.

**User:** The actual human who is using the system.

**World:** The virtual world that contains and objects and players. The players can interact with this world and the objects within this world.
Bibliography


