Scalable Resource and QoS Brokering Mechanisms for Massively Multiplayer Online Games

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Mehadi Hasan

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Multiplayer online games have become an increasingly integral part of online entertainment. With advances in social media, the number of players of these games is increasing at a very rapid rate, which in some cases has been observed to be exponential. This is when resource[1] becomes a concern. In this thesis, I investigated several challenges in developing and maintaining multiplayer games such as hotspots, genre-specific limitations, unpredictable quality of service and rigidity in resource availability. I showed that these issues can be solved by adopting mechanisms for separation of resource concerns from functional concerns and coordination of resources. To support resource coordination, I divided the ownership of resources among three parties-game owner, resource owner and game player. I developed the CyberOrgs-MMOG API, which supports Massively Multiplayer Online Game (MMOG) platforms capable of resource sharing among multiple peers, through mechanisms for acquiring these resources dynamically. I showed that dynamic acquisition of resources can solve the resource questions mentioned above. The API was evaluated using a 2D game with up to 250 simulated players. I also showed, how the game's responsiveness can be dynamically adjusted in a scalable way. This thesis presents the design and implementation of the CyberOrgs-MMOG API, interfaces provided to the interacting agents representing different parties. I integrated a 2D multiplayer game with the API and evaluated the mechanisms supported by the API.

[1]Here by "resource," I mean computational processor time, memory, network bandwidth, etc.

Acknowledgements

I would like to acknowledge my supervisor Dr. Nadeem Jamali for his suggestions and valuable comments to my research. He not only enlightened me with new ideas but also encouraged me throughout my research work. His friendly supervision and guidance made this challenging research work possible. His views on different areas helped me to carry out my research work successfully as well as helped me gain insight on real life matters.

Many thanks go to Professor Dwight Makaroff and Professor Chanchal Roy, the two professors in my thesis committee, who provided me with valuable feedback. Professor Roy took the time to meet with me a number of times and provided valuable advice before my thesis defence. I would like to thank Professor Ramakrishna Gokaraju, for serving as the external examiner on my thesis committee, and offering very useful feedback at the defense.

Also thanks go to Xinghui Zhao, one of the PhD students in our lab for her ideas and clarifications on CyberOrgs API. I really appreciate the time she spent on me and the knowledge she shared. She always provided valuable feedbacks in the group meetings about my research. I am really grateful to her for her selfless contributions.

I also would like to express my appreciation to my family, specially my parents, who always keep me motivated at every stage of my life. Without their encouragement it would be very difficult for me to complete my research successfully.

Finally, I would like to acknowledge the generous financial support in the form of a Graduate Teaching Fellowship and a Faculty Scholarship from Professor Jamali's NSERC Discovery Grant.

Contents

List of Tables

List of Figures

List of Abbreviations

AA	Actor Architecture
AOI	Area of Interest
API	Application Programming Interface
CDM	Cyberorg Directory Manager
FPS	First-Person Shooter
fps	Frames per second
GDM	Game Directory Manager
GPU	Graphics Processing Unit
MMOFPS	Massively Multiplayer Online First-Person Shooter
MMOG	Massively Multiplayer Online Game
MMORPG	Massively Multiplayer Online Role-Playing Game
MMORTS	Massively Multiplayer Online Real-Time Strategy
NPC	Non-Player Character
QOS	Quality of Service
SALSA	Simple Actor Language System and Architecture

Chapter 1

Introduction

Online gaming has recently become a widely used form of entertainment. In many online multiplayer games, people play in a persistent world. In contrast with computer simulated players (also known as NPC or Non-Player Character), the real time in-game interactions among players combined with the social communications involving these players has made multiplayer games more popular than most other types of games [39]. Currently, many multiplayer games are being supported by a wide range of devices from PCs to dedicated consoles and smartphones. To support such a large number of players, multiplayer games require a lot of resources. These resources could be processor cycles, network or storage; however, little effort has so far been invested for encapsulation and control of these resources. My work aims to fill this gap: I develop solutions motivated by the CyberOrgs model [23], to improve upon the largely manual management of resources in multiplayer games, with specific goals of greater flexibility and scalability. The rest of this chapter is organized as follows: Section 1.1 presents brief introduction to multiplayer games, especially online games with a massive number of players. In this section, online game types (genres) and the growth of popular games is discussed. Section 1.2 and 1.3 present thesis motivations and contributions respectively.

1.1 MMOG

A Massively Multiplayer Online Game, also known as MMOG or MMO, is a video game that involves interactions between a large number of players and hence, need the ability to support a large number of players simultaneously. Most multiplayer

games feature at least one persistent world where the game state continues to evolve.

To play an online multiplayer game, one requires an Internet connection as well as a device capable of rendering and displaying graphical content. These devices include personal computers, game consoles such as the Sony PlayStation 3, Microsoft Xbox 360, Nintendo DS and Wii as well as Android, iOS and Windows Mobile operating systems based mobile devices and smartphones. Additionally, MMO game developers are creating multi-platform games for even greater reach. Web-based multiplayer games such as Zynga Poker[1] can also be accessed from Android and iOS operating system based smartphones.

1.1.1 Genres

MMOGs enable players to cooperate and compete with each other in a large scale virtual world, and allows them to socially interact with people around the world. These games belong in a variety of video game genres. This subsection describes some of the most popular multiplayer online game genres.

Massively multiplayer online *role-playing* games, widely known as MMORPGs, are a very common variant of MMOG. In a *role-playing* game, each player assumes the role of a character often represented by an avatar and takes the control of the character's actions. Usually, an MMORPG contains at least one persistent world that continues to evolve even if a player is offline. Like most other genres, these games are basically designed for thin clients to allow more players to join in a game session: a thin client like a web browser provides cross-platform access to the game. This approach reduces cost and provides a more flexible way to play these games. However, due to the limitation of rendering capability, browser-based games are not always an option for MMORPGs.

Another popular genre is *First-Person Shooter* (FPS), which emphasizes on player's skill on aiming and tactical thinking [36]. Unlike role-playing games, this does not depend heavily on in-game bonuses; rather skill is rewarded. Although

[1]http://www.zynga.com/games/zynga-poker.php, access date = 12/12/2011

there are many FPS games, online FPS games are typically not "massive" in their scale. Popular games such as half-life 2, Battlefield 1943 can support only a few players, in most cases up to 50 players. Neocron is probably the first of the FPS games that was targeted at a massive number of players.[2] However, some say that it is a first-person shooter combined with role-playing mechanics.[3] PlanetSide, published by Sony Online Entertainment Inc. in 2003, is the first MMOFPS claiming to support thousands of players simultaneously.[4]

Massively multiplayer online *real-time strategy* game, also known as MMORTS, is another popular MMO genre where players usually take the role of a leader - often as a king or a general, who leads an army. The virtual resources required by this army often are acquired and maintained by the player. MMORTS usually consists of one or more persistent worlds where the player can gather resources and fight even when offline. Examples of this kind of game include Starcraft and Age of Empires.[5] A variation on real-time strategy is turn-based strategy, where players play based on turns or ticks. Ultracorps, published by Microsoft,[6] is one example of this kind of a game.

These are the most common genres for MMOG. Other genres include simulation games such as for racing and sports.

1.1.2 Growth of MMOG

The popularity of online multiplayer games is steadily increasing. These games are now widely used for both entertainment and educational purposes. The reasons for the increasing popularity of MMOGs include social interactions with other people, portability, co-operation and group-based gameplay, and above all, playing against human beings rather than non-human characters [33]. Most MMOGs provide some interfaces allowing communication with other players in the form of text or even

[2]http://rpgvaultarchive.ign.com/features/previews/neocron.shtml, access date=12/12/2011
[3]http://www.gamespot.com/neocron/previews/neocron-preview-2844808, access date=12/12/2011
[4]http://games.ign.com/articles/400/400835p1.html, access date=12/12/2011
[5]http://pc.ign.com/articles/700/700747p1.html, access date=12/12/2011
[6]http://pc.ign.com/objects/663/663003.html, access date=12/12/2011

voice. The widespread popularity of smartphones also provides opportunities to build MMOGs that run on these devices.

MMOG started gaining popularity in the late 1990s with the debuts of Meridian 59, The Realm Online, Ultima Online and EverQuest. The growth rate was faster than expected. In 1991, Neverwinter Nights was developed with a capability of 50 simultaneous players, a number that grew to 500 by 1995. In 2000, some MMOGs started to serve more than thousands of simultaneous players. In June 2010, Eve Online achieved a new record with 60,453 concurrent accounts logged on to the same server.[7] Most recently, in 2011, Chinese MMO ZT Online 2 claimed to reach 435,000 concurrent users.[8]

Before the arrival of the sixth-generation game consoles, which came with the Internet access, the reach of MMOGs was limited to personal computers. Since then, there have been a number of MMOGs developed for game consoles such as EverQuest Online Adventures for the Sony PlayStation 2 and Final Fantasy XI supported by multiple consoles.

Recently, MMOGs started to break into more personal devices like mobile phones and smartphones. These devices allow gamers to play almost anywhere with the Internet connectivity. Among the first of these is Samurai Romanesque, released in 2001 on NIT DoCoMo's iMode network in Japan.[9] SmartCell Technology developed Shadow of Legend, one of the first MMORPGs, which allows gamers to continue their game on their mobile device when away from their PC. Today, many multiplayer games can be played from more sophisticated mobile devices such as the iPhone or Android-based phones.

In 2003, a study by Castronova *et al.* estimated the monetary value of virtual property in EverQuest, the largest MMOG at that time, at a per-capita GDP of 2,266 USD, which is somewhere between Russia and Bulgaria, and higher than China and India [11]. Currently, World of Warcraft is the most popular MMOG in the world with more than 60% of the subscribing player base and about 11-12 million monthly

[7]http://www.eveonline.com/news.asp?a=single&nid=3934&tid=1, access date=12/12/2011
[8]http://www.gamasutra.com/view/news/37470, access date=12/12/2011
[9]http://www.japaninc.com/article.php?articleID=59, access date=12/12/2011

subscribers worldwide.[11] According to a recent survey made by *newzoo*[10] in the US, there are 145 million active gamers and among them, 43% spends money on games. This report also claims, everyday a total of 25M hours are being spent on MMO games, which is 29% of the total Internet-time spent. According to the same report, 2.6 billion US dollars are spent on MMO games.

The graph in Figure 1.1 presents the growth of some of the mostly played MMOGs.

1.2 Motivations

We believe online gaming has a great potential for both education and entertainment. In most cases, the consumers do not need to have any expertise and are ready to pay for this kind of service. As we can see from Figure 1.2, one key requirement of MMOG is the scalability. The total number of MMOG subscriptions has been increasing at a very fast rate. In some games, exponential growth has been observed [15]. Recent MMOGs are connected to different social networks, which facilitates their growth. On the other hand, developers want to keep as much control over the game as they can. The result is low utilization and less flexibility over computational resources. Additionally, developers are responsible for maintaining their hardware as well. There has not been a lot of research in the area of resource encapsulation for MMOGs. This thesis addresses the tight binding of resource concerns with the functional concerns of MMOG. Separating resource concerns would allow game developers to concentrate only on the functional concerns of the game. I show that a loose coupling of resource and functional concerns provides greater flexibility in scalable game design and execution.

This thesis also investigates variability of resource use in an MMOG session. These variable loads are often referred as dynamic hotspots [12]. MMOG researchers have been trying to characterize hotspots and offered different approaches [12]. These dynamic loads are often very tricky to determine in advance because of the variability

[10] http://www.newzoo.com/ENG/1589-Infograph_US.html, access date=12/12/2011

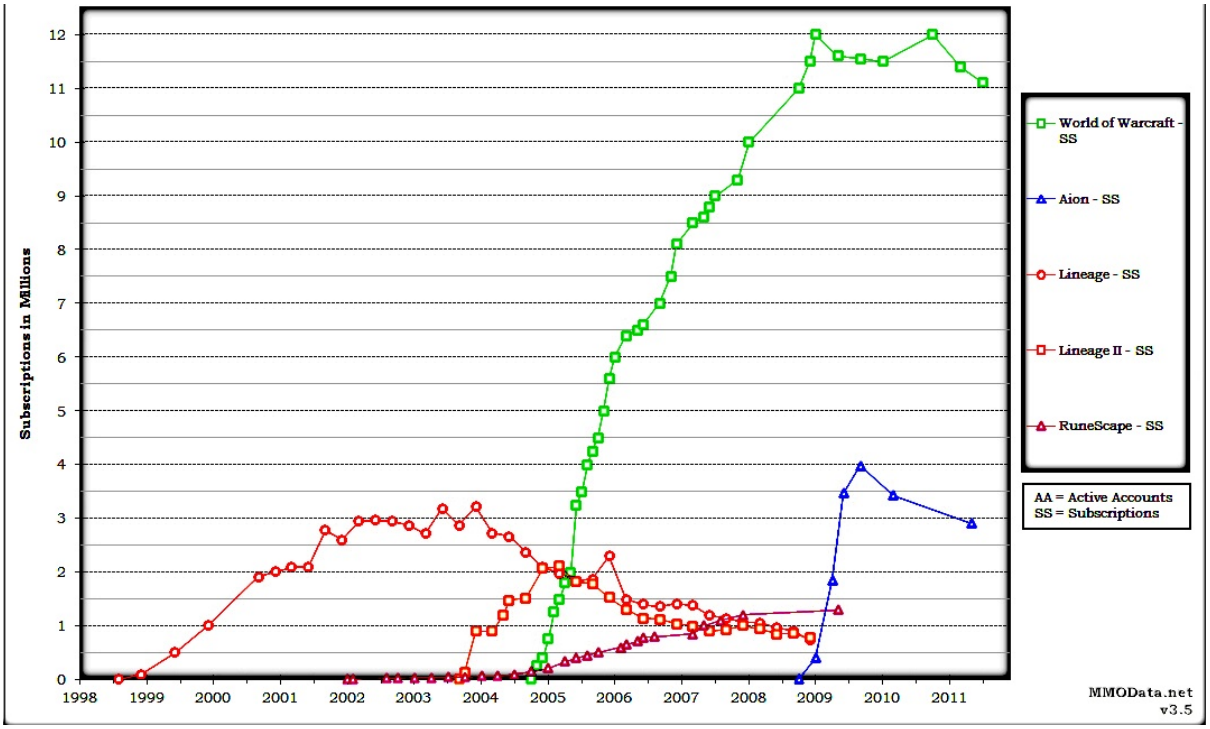


Figure 1.1: Subscriptions and Active Accounts with a peak between 1,000,000 and 12,000,000 [3]

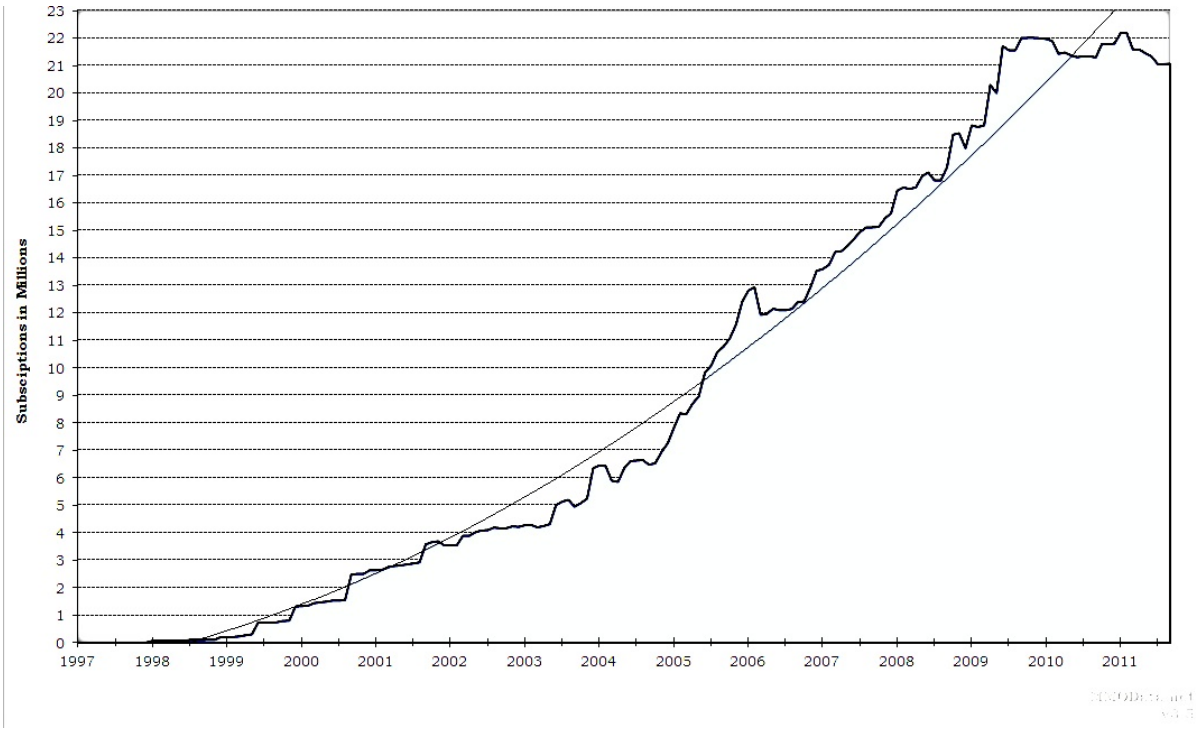


Figure 1.2: Total MMORPG subscriptions and Active Accounts [3]

of players' geographic location, life style, work time etc. Additionally, some multiplayer games allow players to make customized tournaments, which may result in an unexpected load. The random behaviour of hotspots can often affect the overall game performance. Dynamic allocation of resources as needed would maintain the performance as well as it would facilitate better utilization of resources. In this thesis, a novel approach is developed to deal with hotspots by acquiring resources dynamically based on the game's performance parameters.

Another motivation of this research is to investigate the possibility of fine-grained resource control. Although this has not been addressed for MMOGs, work in other domains has delivered some promising results [40]. We believe that players are interested in having greater control over the quality of game experience they receive. For example, a player may like to play in a bigger world with more enemies and new types of missions. This, however, may require more resources, which the player may be happy to pay for. On the other hand, other players may want to stick with the smaller version of the world as this is more affordable for them. Visual experience of the players is a similar concern. In this thesis, I have developed API support to integrate fine-grained resource control with an MMOG.

1.3 Contributions

This thesis addresses resource issues specific to Massively Multiplayer Online Games. The main contributions of this thesis are as follows:

- Development of mechanisms that supports separation of functional concerns from resource concerns and, thus, allows encapsulation of computation in MMOGs. The game developers do not need to be concerned about the resources and can focus on the game development.

- Development of mechanisms to directly control game performance by setting performance thresholds at which more resources are automatically added or released. These mechanisms allow game developers having greater control over the resources and flexibility in managing them.

- Investigation on variability in resource use and development of scalable methodology to deal with load unpredictability by dynamic allocation or release of resources based on pre-defined parameters. Methodology observed in previous studies are adopted to support seamless migration.

- Development of an API that supports the mechanisms mentioned above. A 2D multiplayer game is used to demonstrate the mechanisms supported by the CyberOrgs-MMOG API. The API is evaluated using this demo game, which suggests that the mechanisms can be integrated within a game with negligible impact on the performance of the game.

1.4 Organization

The rest of the thesis is organized as follows. Chapter 2 describes related work providing background knowledge about resource abstraction, control and dynamic load balancing for multiplayer games. Chapter 3 discusses the design and implementation of different components of the system. In Chapter 4, application programming interfaces are provided. Chapter 5 describes hardware and software platform used in the implementation and their limitations. Chapter 6 presents the results on the game performance affected by the API developed and finally, Chapter 7 summarizes the thesis and discusses some possible future directions.

Chapter 2

Related Work

In this Chapter, the existing research work in related areas is reviewed. Section 2.1 presents the Actor model for object-oriented concurrency, which has been used in this work. Actors offer a natural programming framework for implementation of open distributed systems. In Section 2.2, a number of Actor implementations are reviewed. Section 2.3 introduces the CyberOrgs, a model that provides mechanisms for coordinating resources among self-interested peers. Section 2.4 summarizes related work in spatial distribution of game space to build scalable multiplayer games. Works related to genre specific limitations of MMOGs are presented in Section 2.5. In Section 2.6, research related to crowding and different resource provisioning approaches to deal with crowding are reviewed.

2.1 Actors

The concept of Actors was first introduced by Hewitt [19] in his work for PLANNER, a language for proving theorems in Robots. Later, Hewitt *et al.* formalized the Actor model in [20, 18]. His work was carried on by Grief, who developed an abstract model for actors [17] and Clinger, who developed the semantics for actors [13]. Afterwards, Agha extended actors to programming languages [6, 7]. He is also the pioneer in modelling Actors for data abstraction [5] in open distributed systems.

Figure 2.1 shows the structure of an actor. Actors are autonomous computational entities. An actor consists of a state, set of behaviours (methods) and a thread of control. Actors communicate with each other using asynchronous, point-to-point messages. Each actor has a globally unique name, which is used by other Actors to

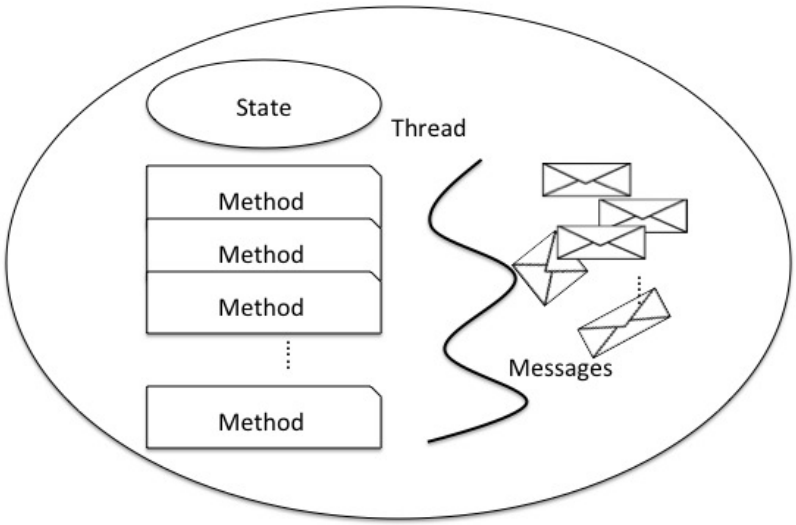


Figure 2.1: Structure of an actor.

send messages. As Actors represent computations they can be distributed over time and space. In other words, it is possible to enable an actor for a specific time at a specific location. Actors store unprocessed messages in a queue and process them one by one according to the order of arrival. The processing of messages, however, solely depends on the scheduling strategy implemented in the underlying Actor System. Three types of *actor primitives* may occur during the processing of a message:

- An actor can send messages to another actor. The name of the destination actor must be known to the sender actor. The messages sent by Actors are guaranteed to eventually be delivered to the destination actors, but the order of arrival is not guaranteed.

- New actors can be created with predefined characteristics. The creator actor knows the name of the newly created actor.

- An actor can change its own state.

2.2 Actor Implementations

There are several implementations of Actors. In this section, we review some of the existing Actor Systems: *Actor Architecture* [34] , *Actor Foundry* [26] and *SALSA*.[1]

2.2.1 Actor Architecture

Actor Architecture is an actor-based framework implemented in Java. Actor Architecture consists of AA platforms that provide execution environments for actors as well as API support to develop actors. These platforms also allow Actors across distributed systems to execute and communicate with each other through message passing. Each AA platform consists of four service layers:

- *Actor Management Service:*
  The Actor Management Service layer manages the states of the actors and manages all the migrations between the AA platforms.

- *Message Delivery Service:*
  The Message Delivery Service layer handles transportation of all the local messages in the AA platform.

- *Message Transport Service:*
  The Message Transport Service layer provides an interface to communicate between actors of other AA platforms. All messages that are sent to or received from another AA platform pass through the Message Transport Service layer.

- *Advanced Service:*
  The Advanced Service layer provides middleware services, such as matchmaking and brokering, which facilitates look up services to search for a particular actor.

[1] http://www.cs.rpi.edu/research/groups/wwc/salsa/index.html

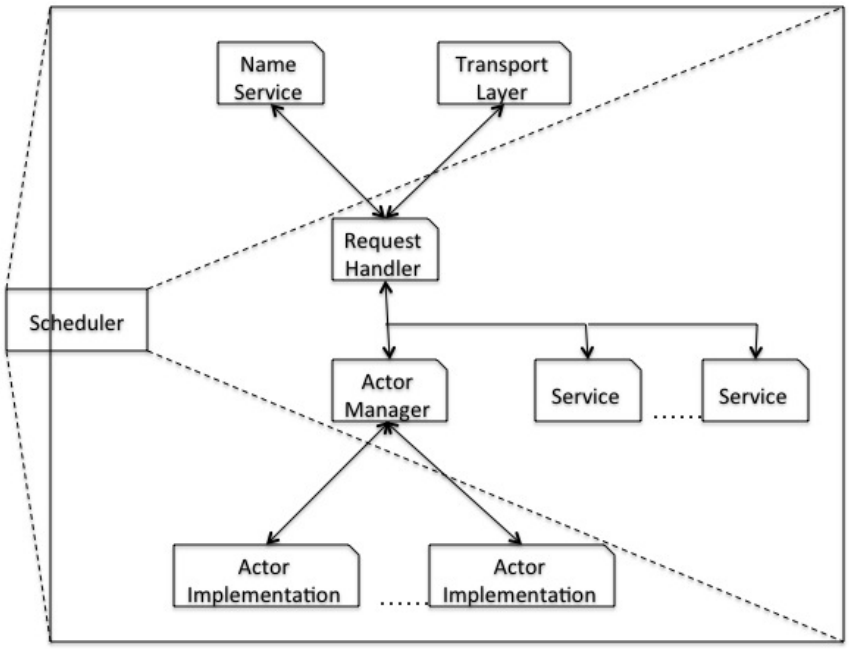


Figure 2.2: Actor Foundry Node Structure [9].

The distributed resource management system that is presented in this thesis has been developed by extending Actor Architecture and the CyberOrgs model. The CyberOrgs [23] model, discussed in Section 2.3 is developed using Actor Architecture.

2.2.2 Actor Foundry

Actor Foundry is another implementation of Actor model developed using Java. Each instance of the Actor Foundry run-time system is called a foundry node. Each foundry node can employ many actor instances and these actors may communicate with each other using asynchronous messages. The message delivery in Actor Foundry is weakly fair; that is, messages are guaranteed to be delivered eventually at the destination. Like Actor Architecture, Actor Foundry also allows programmers to define behaviour of actors through the programming interface provided.

Figure 2.2 shows the structure of a foundry node. As we can see from the figure, each foundry node consists of seven basic components:

- *Actor Manager:*

  The Actor Manager carries out the operations required for actor creation and

begin scheduling. It is also responsible for all intra-node communication and carries out requests from other actors to the request handler.

- *Actor Implementation:*

  Each actor instance is represented by an Actor Implementation. It transfers messages to the Actor Manager, which handles local communication inside a node.

- *Service:*

  The Service Modules provide additional platform specific services to the local actors. Actors communicate with the Actor Manager to access these services.

- *Request Handler:*

  The Request Handler module bridges a local Actor Manager to another Actor Manager on a different foundry node. It provides an interface to communicate other Actor Managers through synchronous or asynchronous remote procedure calls (RPCs). The low-level detail of the message transport services is encapsulated by the Transport Layer. The request handler also provides Name Service to identify an Actor.

- *Name Service:*

  Each actor in the Actor Foundry is given an unique name. This name is generated by the Name Service module. Request Handler, using this module can setup appropriate Remote Procedure Calls.

- *Transport Layer:*

  The Transport Layer deals with the low-level communication protocols. Messages of any size are guaranteed to be transported eventually.

- *Scheduler:*

  The Scheduler module schedules all the threads in a foundry node. Actor Foundry employs fair scheduling strategies.

2.2.3 SALSA

Simple Actor Language System and Architecture (SALSA) is an actor-based programming language designed for developing dynamically reconfigurable open distributed applications. SALSA supports all basic actor primitives such as asynchronous message passing, unbounded concurrency and state encapsulation. Additionally, SALSA provides universal naming, remote communication and migration services to support distributed computing over the Internet [38]. Furthermore, SALSA provides high-level abstractions such as token passing, join and first-class continuations to facilitate concurrent operations.

SALSA syntax is very similar to Java. SALSA compiler translates a SALSA source code to a Java source and then Java compiler produces the final Java bytecode. This facilitates portability across various Java-supported platforms.

2.3 CyberOrgs

The CyberOrgs model [22], introduced by Jamali *et al.* is a model for hierarchical resource coordination between multiple self-interested peers over a network of peer-owned resources. A cyberorg encapsulates an amount of resource as well as a set of computations executed by concurrent actors. Additionally, a cyberorg can host another cyberorg and there exists a contractual relationship between the child cyberorg and the parent cyberorg, potentially the host of a child cyberorg. The resource required by a concurrent activity, executed by an actor is allocated by the containing cyberorg. The contractual relationship between cyberorgs is analogous to a buyer-seller relationship. The virtual currency that flows among cyberorgs is called *eCash*.

Figure 2.3 shows the structure of a cyberorg. In this figure, each ellipse represents a cyberorg and each curved line represents a computation. Each cyberorg contains actors, eCash, messages and may contain one or more cyberorgs. In this model, actors represent computations and a cyberorg can use its eCash to buy re-

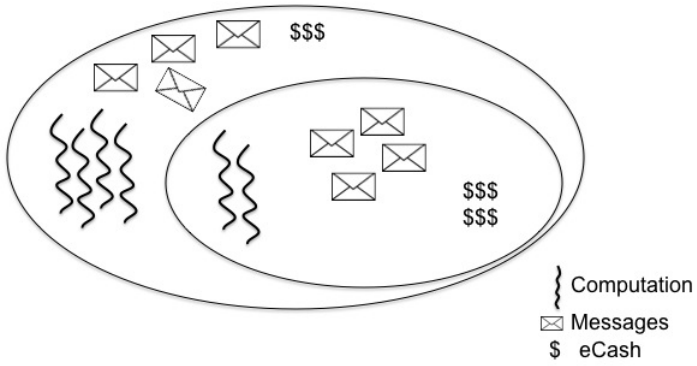


Figure 2.3: Structure of a cyberorg.

sources required for these computations. Actors in the cyberorg communicate using asynchronous message passing. A cyberorg organizes its resources and computations in a hierarchical fashion. A cyberorg can purchase additional resources from another cyberorg according to a contract. This contract is made through a successful negotiation between these two cyberorgs. The contract specifies the types and quantities of resources that will be delivered to the hosted cyberorg. The cost of the resources is also specified in the contract. A cyberorg distributes its resources among the hosted cyberorgs and the local computations according to its own local resource distribution policy.

The CyberOrgs model defines resources (computational and communication for example) in terms of time and space. A resource expires at a particular point in time at a particular location; if it is not used by some computation. In the CyberOrgs, ticks are defined as the unit of consumable resource and defined in time and space. For example- a resource R at location L has N ticks available from time T1 to time T2. Every computation requires a certain number of ticks to complete.

CyberOrgs Primitives

In the CyberOrgs model, three primitive operations are defined.

- *Isolation:* A cyberorg can create a new cyberorg by using Isolate primitive. This mechanism allows a cyberorg to collect some of its actors, eCash and

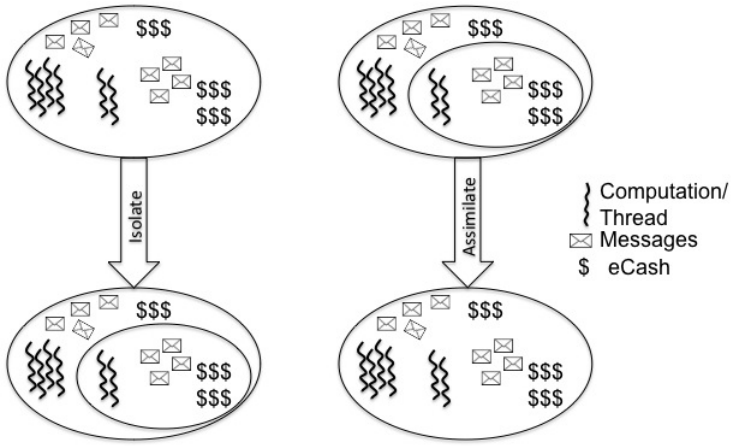


Figure 2.4: Isolation and Assimilation.

allows this new cyberorg to be hosted locally. A contract is also formed between the host cyberorg and the newly created one.

- *Assimilation:* Using Assimilate primitive a cyberorg can relinquish all of its computations, resources, eCash and control to its host cyberorg and disappear.

  Figure 2.4 shows the *isolation* and *assimilation* operation.

- *Migration:* Migration is a little more complex than the two other primitives and plays a very important role in the CyberOrgs model. A cyberorg may realize that its resource requirement has exceeded according to the contract. If the cyberorg requiring resources has enough eCash, it attempts to migrate to another cyberorg where it can buy more resources and process its computations. Before migration, a cyberorg searches for potential hosts that have enough resources available for purchase. Each migration is initiated with a contract between two cyberorgs. A cyberorg that requires resources for the computations it holds can offer other cyberorgs to trade resources for eCash. Both cyberorgs can negotiate about the terms of the contract. When the negotiation is successful and a contract is signed, the new host allows the other cyberorg to migrate and use its resources.

  As shown in Figure 2.5, cyberorg C3, hosted by cyberorg C2 migrates to cyberorg C1 after a successful negotiation.

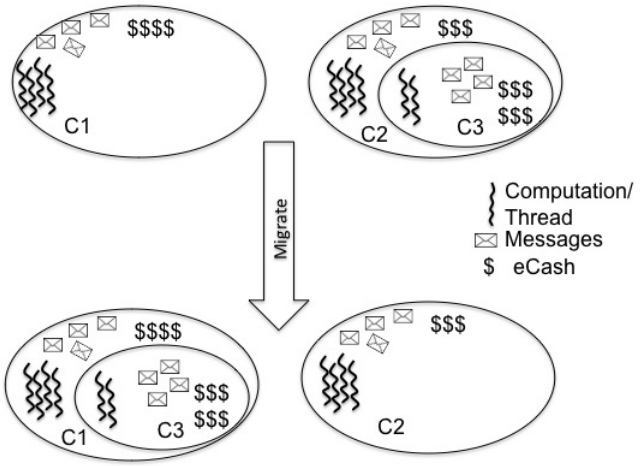


Figure 2.5: Migration.

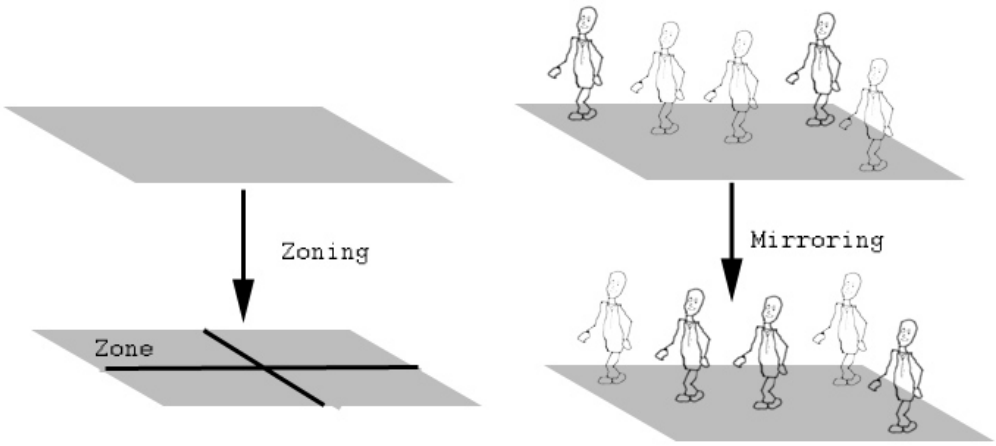


Figure 2.6: Zoning and Mirroring.

2.4 Spatial Scaling of MMOG

This section reviews some of the ideas that are already being used in some MMOGs. In Subsection 2.4.1, three different techniques for spatial scaling are discussed: *zoning*, *mirroring* and *instancing*. Additionally, works related to these techniques are also reviewed. In Subsection 2.4.2, area-of-interest (AOI), a concept being used heavily in MMOGs to reduce communication burden is discussed. This subsection also presents some strategies that facilitate seamless zonal migration.

2.4.1 Zoning, Mirroring and Instancing

Zoning [10] technique used in many games, is based on concepts from data locality in scientific parallel processing. Zones are created to partition the game world into smaller areas. Each zone is handled by a separate host/server. The host is responsible for processing all interactions and requests from the clients that reside in the respective zone. Depending on the implementation of the game, hosts can communicate with each other to share updates and may allow players to migrate form one zone to another. However, this approach alone does not offer much flexibility and scalability in game design nor does it provide any fine-grained control over the resources. Zoning is currently being used in some online adventure games, namely MMORPGs [8].

Another technique called mirroring [30] is used for parallelizing game sessions for densely populated zones and allows users to see only the objects within their area-of-interest. These densely populated areas are also referred as hotspots [35]. This novel approach provides distribution of load by replicating the same game zone over multiple hosts. Each host processes a set of entities called active entities for that host. Other entities that are not processed are called shadow entities. Shadow entities are processed in another host where they are considered as active entities. This approach highly scales because the amount of resource required for data transfer as well as computation for an update of a shadow entity is much less than that of transferring and computing the whole game state. Figure 2.6 shows how zoning and mirroring are used. In the right figure, dark characters represent active entities and grey characters represent shadow entities.

Instancing [29] offers another approach to scale a multiplayer game by distributing the load by creating multiple independent instances of the same zone. Sub-areas that have a very high frequency of access are considered for instancing. Each host responsible for processing a sub-area is called an instance server. The difference between mirroring and instancing is that each instance server processes entities completely independent of each other.

Based on these three techniques, Real-Time Framework (RTF) [16], a Grid-based middleware is developed to scale game sessions. RTF is a multilayered service-oriented architecture that uses the potential of grid computing to provide access to unbounded amount of resources. Based on RTF, a model for computing load for MMOGs is developed [35]. This model also offers opportunities to predict load in advance and necessary steps can be taken to balance the load by real time provisioning of resources. Another load balancing approach is proposed by utilizing the semantics of the simulation executed by the server [14]. In [37], a communication architecture is developed for Networked Virtual Environments that takes advantage of unstructured peer-to-peer (P2P) overlay networks for the distribution of messages.

Some of the above ideas are employed in some MMORPGs. However, MMORPG is a specific genre that allows a slow-paced gaming experience than other online genres like First Person Shooter (FPS) games. A highly interactive, fast-paced game requires a lot more processing to update a player's state than any MMORPG. Also there are no approaches developed yet to provide fine-grained control over the resources for the gamer community. Motivated by above approaches and CyberOrg model, this thesis aims to build a scalable online gaming framework with more fine-grained control over the resources and more flexibility in acquiring resources.

2.4.2 Interest Management and Zonal Migration

Scalability is a critical issue when developing MMOGs or multi-user simulation environments. In most modern MMOGs, scalability is achieved through interest management; in other words by dividing the virtual world into smaller areas or zones where each zone is managed by one server [24]. However, due to the unpredictable nature of hotspot creation in a zone, the zoning approach, alone does not always offer the performance and scalability as required. Besides, a static distribution of these areas makes it hard for clients to migrate from one region to another. In some games, this is done using portals. Portals are gateways used to transport a player from one region to another and manages the lag by presenting the user a loading screen or special effect that does not necessarily require any interaction with the

server. However, this is not often the case for most games and many developers might want to avoid this kind of solution. This approach provides a discrete view to the users as they can not see objects beyond the zonal boundaries. Some MMOGs might need a vast open world without these gateways and require to migrate a gamer from one zone to another seamlessly.

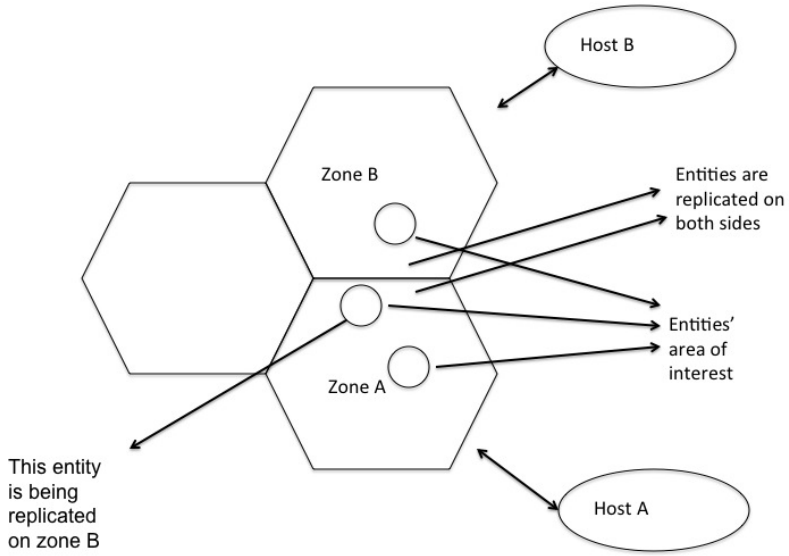
In [27], Lu *et al.* presented a model that facilitates communication among players based on the player behaviour and interactions. In this paper, the author defined the concept of aura, an area enclosed by a sphere for interest management. This behavioural modelling is dynamic in nature and is based on the altitude and viewer range of view. In a nutshell, this model only enables the server to deal with the entities that are in the view radius of the player. Knutsson *et al.*'s P2P Support for Massively Multiplayer Games [25] and Iimura *et al.*'s Zoned Federation of Games Servers [21] proposed a discrete view of the zones; all computation in a zone is handled by a server and has a discrete view of the world.

Figure 2.7a shows distribution of game space into several hexagonal shaped zones. How an entity moves from one zone to another is illustrated by Figure 2.7b. These approaches independently might provide some level of scalability. However, to offer a gamer with the seamless experience of a huge virtual world, a different approach to zonal migration is required. This thesis investigates these approaches, adopts some of these ideas and combines them to offer a continuous seamless experience to the gamer community without compromising game performance and scalability.

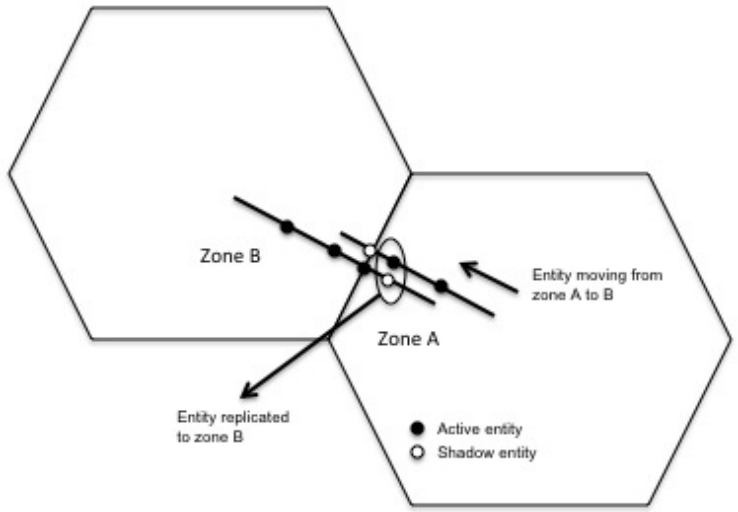
2.5 Genre specific limitations

Most studies and experimental works so far have focused on the design aspects limited to a special category of games, role-playing-games (RPGs) [16]. The other online genres like first-person-shooter (FPS) do not allow "massive" number of players to play at the same time. This could be due to the higher interactivity and fast-paced game-play of an FPS than that of an RPG.

A study by Abdelkhalek *et al.* [4] showed some interesting analysis and presented



(a) Distribution of computation among zones



(b) Migration of an entity from A to B

Figure 2.7: Zonal migration

(a) Screenshot taken from Halflife 2 (non-MMO).

(b) Screenshot taken from WoW Cataclysm (MMORPG).

Figure 2.8: Screenshots of non-MMO and MMO

some attempts to improve the number of concurrent players in Quake 2, an online FPS game. Quake 2 is an open source 3D first person shooter game developed and distributed by *id software*.[2] Quake multiplayer mode follows client-server architecture. All physics simulations, updating state inconsistencies originated from network delays, hardware issues as well as propagation of updated states to clients are handled by a single, centralized server. The first version of Quake allows 32 players to play simultaneously. This study was able to increase the player limit from 32 to 90 simultaneous players. Their experiment concluded that the bottleneck is caused by the lack of CPU resource rather than network bandwidth. In most cases, incoming bandwidth is constant and low. Some parallelization methods were implemented using task decomposition and synchronization techniques to increase the number of simultaneous players.

However, there has not been any significant research in improving the scalability of FPS games. Recent online games like Counter Strike, Half-Life 2 or Battlefield 1942 do not allow more than few tens of players. Another FPS, Battlefield: *Bad Company* (2008) supports up to 24 simultaneous players [3]. Figure 2.8 presents screenshots of an online FPS, Halflife 2 and an MMORPG titled WoW Cataclysm.

[2]http://www.idsoftware.com/
[3]http://badcompany.ea.com/about/

It is noticeable from the screenshots, that the FPS graphics is much more detail than MMORPG one. In this thesis, this question is addressed in terms of resource coordination and sharing. Novel approaches for resource encapsulation, coordination and sharing for MMOGs are developed. This allows a developer to build online games with potentially unbounded resources, of course, as long as the developer pays and thus, can meet the desired QoS and scalability requirement.

2.6 Dynamic Resource Provisioning

This section reviews load variability and approaches that can be used to balance the consequences on resource usage. Subsection 2.6.1 discusses how hotspots are created and their behaviour. In Subsection 2.6.2, different approaches to model loads and some strategies to provision resources dynamically are presented.

2.6.1 Hotspots

Crowding happens when many players move in to the same zone. As each zone has limited resources, it makes the game server perform poorly if the population gets too high for the server to handle. Crowding violates the quality of service and affects gaming performance. The simple distribution model of resources among zones may not work when crowding happens. Therefore, only even distribution of resources among zones may not be the best approach for balancing the load. In most cases, the population distribution caused by crowding is very random in nature and can not be predicted in advance.

In [12], Chen *et al.* pointed out that when many players move into the same zone, the result is "flocking", an MMOG pattern that can not be ignored. The reason behind this could be the zone is more interesting for its rich content. Some games like real-time strategy games and war games may be scheduled for special battles at specific times. Moreover, people are more likely to play at their leisure time. Therefore, games could be less overloaded during work hours and more overloaded during weekends and times when people do not work. Obviously, this also depends

on the timezones and the number of players from those timezones as well.

2.6.2 Dynamic load balancing

To deal with transient crowding problem Chen *et al.* proposed a locality aware dynamic partitioning algorithm. This, decentralized algorithm is based on a heuristic approach that allows the game to i) shed load from an overloaded host considering the locality of the game-entities and ii) merge hosts in normal load condition for reducing excessive inter-server communication due to the partitioning of the hosts. Load shedding is constrained by achieving the safe load target without exceeding the safe load threshold on any nodes to which the overloaded node sheds load. This approach is also aimed at preserving the locality, i.e., the number of strongly connected components must have to be same as before load shedding. Additionally, this strategy also has an optimization goal to keep the number of region migrations incurred due to load shedding minimal.

Another part of Chen *et al*'s work was to aggregate hosts when the quality of service degrades due to excessive inter-server communication instead of high client load. They presented a heuristic graph merging algorithm to merge servers and improve the quality of service.

According to Nae *et al.* [32, 35], current MMOG industry practice is to over-provision resources due to the high variability of resource demand and lack of flexibility in resource renting policies from third parties. This kind of over-provisioning of resources only enables the big companies to enter the MMOG industry. They addressed the issue of high entry and operational costs and proposed a new dynamic resource provisioning method for MMOGs using third party data centers to enable the developers a low cost solution. In this study, Nae *et al.* attempted to identify the type of interactions that cause short-term load variability, which complements the long-term load variability because of the population increase. Based on the player interaction type and the size of the population, a combined processor, network and memory model is presented. This model estimates the MMOG resource demands dynamically and thus, provides opportunities for dynamic resource provisioning.

2.7 Chapter Summary

In this chapter, work in several related areas is reviewed including the CyberOrgs, a model for resource coordination along with Actor Systems and some actor implementations. Some resource distribution approaches such as zoning, mirroring and instancing are briefly discussed. Zonal migration, genre specific limitations as well as load variability issues are addressed as resource coordination problems.

Chapter 3

Design and Implementation

This chapter discusses the methodology used in implementing the CyberOrgs-MMOG platform. The CyberOrgs-MMOG API is extended from the CyberOrgs API, a prototype of the CyberOrgs model [22] implemented by Zhao *et al.* [23]. This implementation builds upon Actor Architecture [31], an implementation of the Actor model [6]; therefore, it is important to understand the structure of the Actor Architecture as well. Section 3.1 summarizes the design and implementation of the Actor Architecture. In Section 3.2, design details of the CyberOrgs-MMOG platform are presented. This section also describes the structure of the CyberOrgs-MMOG platform along with different components of the system, particularly implementation details of support for maintaining resource ownership and coordination, as well as support for zones. Strategies used for incorporating seamless zone migration and load management are also discussed in this section. Design and implementation of the broker and ownerships are discussed in Section 3.3. In Section 3.4, implementation details of Directory Manager, another component of the system, is presented. Section 3.5 summarizes the mechanisms developed to support dynamic resource coordination.

3.1 Actor Architecture

The Actor Architecture (AA) is a middleware system that allows actors (previously introduced in chapter 2) to communicate with each other within a platform and provides actors an execution environment. Each instance of the AA run-time, potentially executing on a separate node, is called a platform. An AA platform has eight components divided into four service layers. Figure 3.1 shows an AA platform

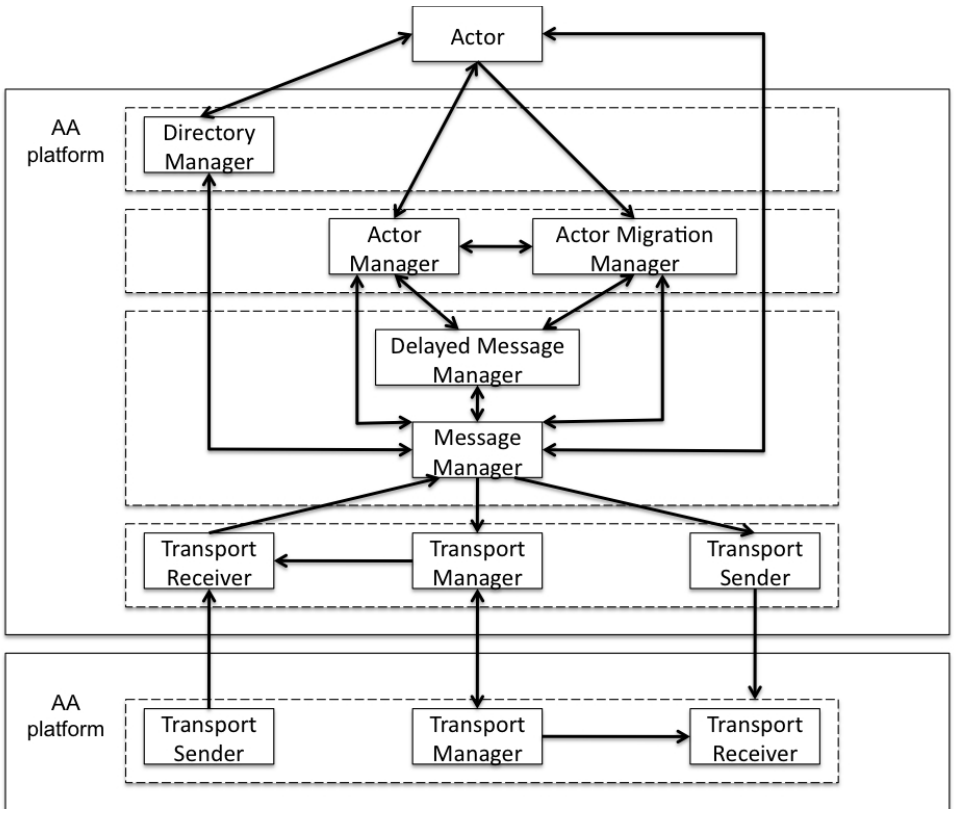


Figure 3.1: Structure of the AA platform [34].

along with its components. The rest of this section discusses different components of an AA platform.

- **Message Transport Service**

  The Message Transport Service layer is responsible for transportation of messages from one AA platform to another. This service layer consists of three components.

  - *Transport Manager*: The Transport Manager is the central component of the transport service layer. It provides the AA platform an interface with other AA platforms. Basically, the Transport Manager creates a communication channel between one or more Transport Managers located in other AA platforms.
  - *Transport Receiver*: The Transport Receiver receives messages from other AA platforms and delivers those messages to the Message Manager of the current AA platform. The Message Manager delivers the message to the destination actor.
  - *Transport Sender*: The Transport Sender receives messages form the Message Manager of the current AA platform and is responsible for sending each of these messages to the Transport Receiver of the destination AA platform. This message sending service must be initiated by a communication channel created by the Transport Manager. If there is no connection available, Transport Sender requests the destination Transport Manager to open up a connection.

- **Message Delivery Service**

  The Message Delivery Service layer provides services to deliver a message to the destination actor. This layer consists of two components.

  - *Message Manager*: Each AA platform contains at least one Message Manager. The Message Manager is responsible for handling all messages in the corresponding AA platform. If the destination actor of a particular

message is inside the local AA platform, the message is delivered to the local actor. On the other hand, if the destination actor is located in a different AA platform, the local Message Manager delivers the message to the Message Manager of the destination AA platform using the Message Transport Services. The remote Message Manager eventually delivers the message to the destination actor.

- *Delayed Message Manager*: Due to various reasons, such as, actor migrations, network delay or network interruption etc. a message intended for an actor could be delayed for delivery. The Delayed Message Manager is responsible for buffering these messages temporarily and delivers them whenever the destination actor is ready to receive the messages.

- **Actor Management Service**

  The Actor Management Service layer provides services to manage the actors in the AA platform. This service layer has two components.

  - *Actor Manager*: The Actor Manager manages states of all actors in the respective AA platform. It also deals with actor operations, such as sending or receiving a message using the Message Delivery Services.

  - *Actor Migration Manager*: The Actor Migration Manager is responsible for the services that allow an actor to migrate from one platform to another. Migrations are also performed using the Message Delivery Services.

- **Advanced Service**

  The Advanced Service layer consists of only the Directory Manager.

  - *Directory Manager*:

    The Directory Manager provides the actor naming services as well as other services such as matchmaking and brokering.

3.2 CyberOrgs-MMOG platform

Figure 3.2 shows a snapshot of the system with the components interacting with each other during an MMOG game session. There are four types of agents in the system: resource owner, game owner, broker and game player. These are discussed in more detail in Section 3.3. A resource owner can own one or more resources, which can be registered to the Directory Manager as available resources through the broker. There can be multiple brokers each encapsulated by a cyberorg. Same is the case with the Directory Managers. As we can see in Figure 3.2, each resource (a CPU for example) hosts a CyberOrgs-MMOG platform, which allows resource coordination and hosting of computations on the fly. The CyberOrgs-MMOG platform is also responsible for monitoring the server load in real time and for making load balancing decisions.

A CyberOrgs-MMOG platform provides a programming interface to developers to build MMOGs in a modular fashion with clear separation of resource concerns from computational concerns. Figure 3.3 shows the core components of a CyberOrgs-MMOG platform with the interactions between the components. A CyberOrgs platform, implemented by Xinghui Zhao [41] manages the underlying low-level details of resource coordination mechanisms. The Zone Manager keeps track of the hosted game zones of a specific game instance and facilitates migration of a zone from one host to another for that particular instance. The Zone Manager also provides feedback to the Load Manager about the population of a zone. The Load Manager is responsible for monitoring the game load and providing feedback to the Zone Manager. There are dedicated Zone Managers and Load Managers for every instance of a game; a CyberOrgs-MMOG platform can host multiple game instances. A broker provides interfaces to players and game/resource owners allowing them to interact with the system. The broker also keeps track of the resources and available games using the Directory Manager, and thus, provides relevant information to the interacting users. Different components of the CyberOrgs-MMOG platform have been discussed in the following subsections.

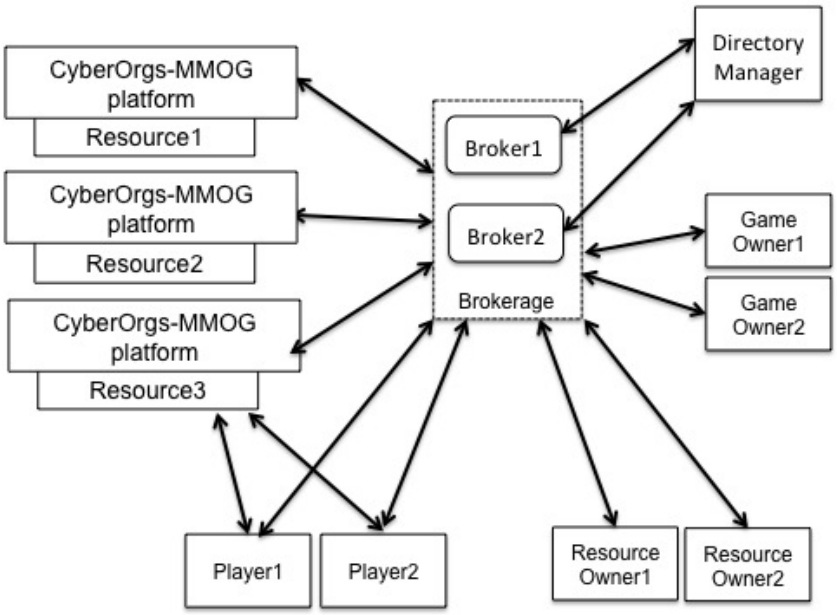


Figure 3.2: Snapshot of interactions during MMOG game sessions using CyberOrgs-MMOG API

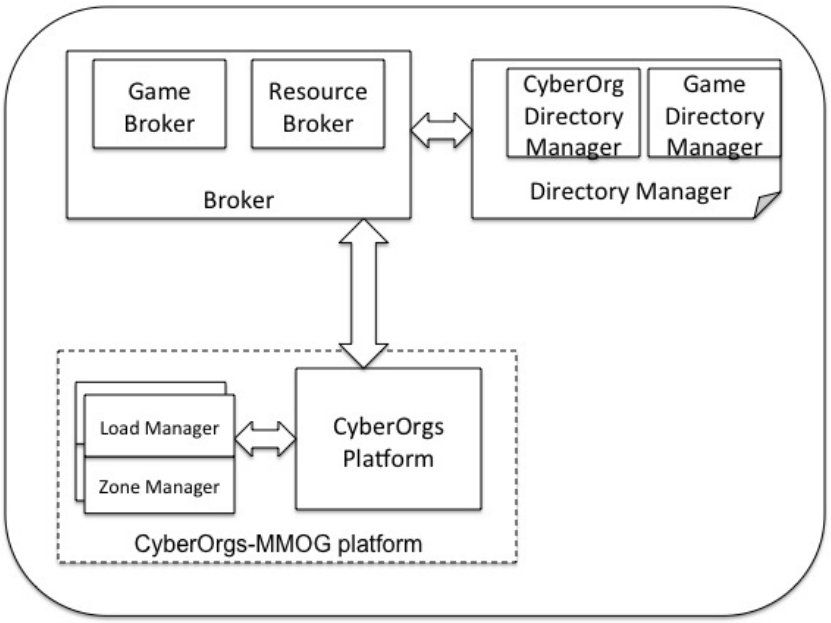


Figure 3.3: Components of the CyberOrgs-MMOG platform, broker and Directory Manager

3.2.1 CyberOrgs Platform

The CyberOrgs API provides programmers with more flexibility in resource acquisition by separating the concerns between computations and resources. Figure 3.4 shows the components of the CyberOrgs implementation. Each CyberOrgs platform contains two additional components along with the Actor Architecture implementation: CyberOrg Manager and Scheduler Manager. The CyberOrgs implementation [41] deals with processor resource allocation and coordination. This work has been modified to support acquisition, coordination and control of resources specific to multiplayer games. The rest of this subsection discusses different services provided by CyberOrg Manager and Scheduler Manager in detail.

- **CyberOrg Manager**

  CyberOrg manager encapsulates computations using actors where each actor represents a computation. The entity that encapsulates a set of computations (using actors) in a resource boundary is called a *cyberorg*.

  Each cyberorg in a CyberOrgs platform is the basic entity of resource acquisition and control. A cyberorg consists of a set of actors, some units of eCash, an amount of resource and a list of hosted cyberorgs. Cyberorgs are organized as a hierarchy in the platform. Actors in the CyberOrgs platform rely on their encapsulating cyberorgs for resource acquisition. Cyberorgs coordinate and exchange resources in a market of resources where eCash is the currency for acquiring these resources. A cyberorg can use its eCash to buy additional resources from its host cyberorg in order to support all of its computational tasks.

  Zhao's CyberOrgs implementation [41] deals with processor time resource. Initially, all resources in the system belong to the root cyberorg of the platform. Other cyberorgs can be created or migrated and hosted at this root cyberorg in a hierarchical fashion. A cyberorg maintains a list of hosted cyberorgs and there exists a contract for each of them. A contract specifies the amount of

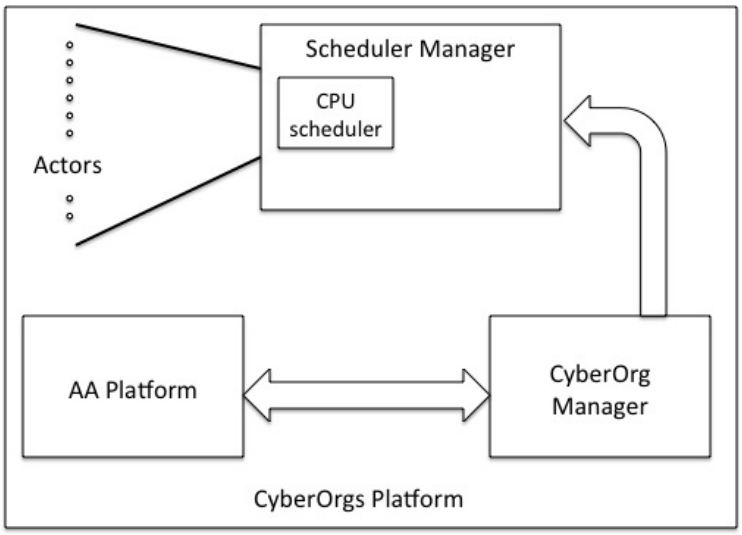


Figure 3.4: A CyberOrgs plaftorm

resource a client cyberorg can own and a certain units of eCash that will be consumed for each allocation of resources.

The CyberOrg Manager is responsible for managing the structural integrity of the cyberorgs as well as supporting execution of *CyberOrgs primitives*. The CyberOrg Manager interacts with the Scheduler Manager in order to control processor time for each cyberorg. Following are the functions of the CyberOrg Manager:

1. *Maintaining the organization of the cyberorgs hierarchy*

   One of the responsibilities of the CyberOrg Manager is to maintain the hierarchical structure of the cyberorgs. The CyberOrg Manager modifies the hierarchy to reflect the change in the structure if a change is required by the invocation of a CyberOrgs primitive.

2. *Carrying out CyberOrgs Primitives*

   The CyberOrg Manager is responsible for carrying out the necessary steps for primitive operations: *Isolate*, *Assimilate* and *Migrate*. Each cyberorg contains a special actor called *facilitator* to invoke administrative operations such as the primitives.

3. *Interacting with Scheduler Manager*

Once the CyberOrg Manager has computed the allocation of resources, it interacts with the Scheduler Manager to schedule the computations. The CyberOrg Manager translates the change in allocation of resources as a result of execution of a *primitive* operation, and updates the Scheduler Manager, which eventually reschedules the cyberorgs.

- **Scheduler Manager**

  The Scheduler Manager schedules threads in a round-robin fashion using Java's suspend and resume primitives. For efficiency reasons, the CyberOrgs implementation [23] uses a flat queue. Each computation is scheduled for a time slice calculated by the CyberOrg Manager according to the contract with the host cyberorg. The CyberOrgs API only deals with processor resource allocation.

3.2.2 Zone Manager

The Zone Manager manages the distribution of zones. A zone is a part of the game world created by spatial subdivision of the game world. This spatial subdivision is required to allow distribution of computation over multiple hosts. In the CyberOrgs-MMOG implementation, zones are abstracted from the hosts. This is facilitated by allowing a cyberorg entity to encapsulate one or more zones. The Zone Manager keeps track of the hosts that are being used for a zone. Like cyberorgs, zones also follow a hierarchical structure. A zone can be divided into multiple sub-zones and the parent zone keeps track of how the division is made. Whenever a game player tries to play in a particular zone, the Zone Manager of the corresponding zone finds the appropriate sub-zone where the player can start playing. The Zone Manager is also responsible for keeping consistent states between zones and allows a gamer to seamlessly migrate from one zone to another.

To facilitate seamless experience, when a gamer moves from one zone to another each zone boundary is extended. As we can see from Figure 3.5a, the extended boundaries allow multiple zones to replicate the same region as well as entities that fall inside that region. In this figure, the thick dotted lines represent the start of

the replication process, black continuous lines are the actual boundary of a zone and light dotted lines are the extended boundary of a zone. Zone A and zone B are encapsulated by cyberorg A and cyberorg B, respectively. The computational tasks required for these zones are executed by the hosts of the encapsulating cyberorgs. The Zone Manager is also responsible for exchanging information between the zones and keeping the events inside the shared region synchronized. To separate the entities in the shared region, each entity is tagged either as an *active entity* or a *shadow entity*. Basically, a *shadow entity* is a replica of an *active entity*. Whenever a player or an entity navigates through the shared region, it becomes an *active entity* to the new zone and becomes a *shadow entity* in the old zone. During this migration, the connection between a game player and a host is also transferred to the host of the new zone.

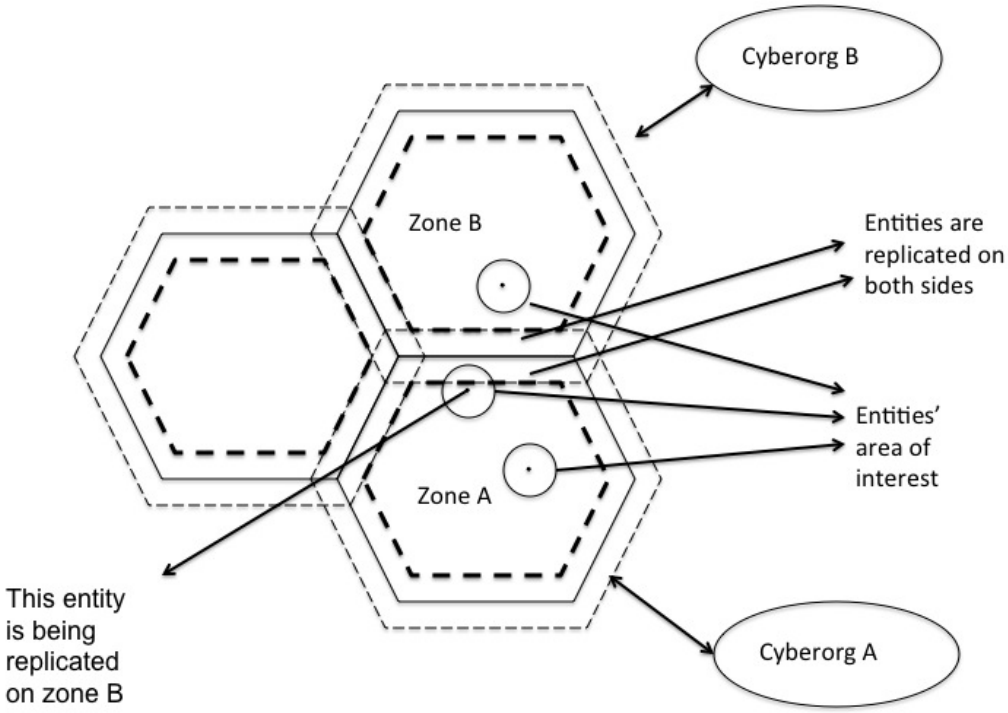
A game player only receives the updates he can see. This is defined by the area of interest (AOI), the visible region around the entity. The AOI contributes to the outgoing bandwidth of a host, in other words, network resource. Therefore, by controlling the size of the visible area around, a game player can control the consumed network resource.

Figure 3.5b illustrates a typical migration of an entity from zone A to zone B. For a smooth transition from one zone to another the area near the boundaries are replicated on both servers and mostly created with static, non-movable objects. Following steps are taken to facilitate a seamless zone migration.

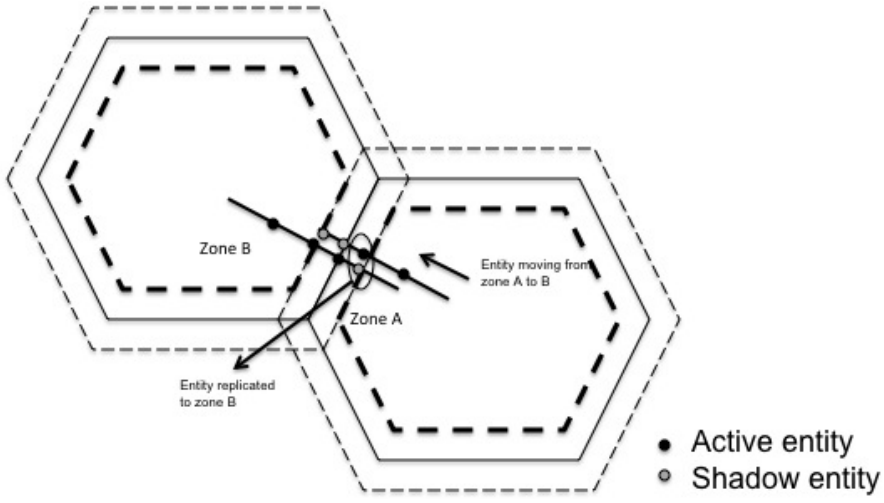
- *Step 1:* The entity within zone A moves into the overlap region and is replicated as a shadow entity on zone B. Here, zone A and B are encapsulated by cyberorgs A and B, respectively. The handoff process starts as soon as the entity's area of interest falls into the overlapped region.

- *Step 2:* After the entity passes the half way of the overlap, cyberorg A automatically changes its status in A from active to shadow and vice versa for cyberorg B.

- *Step 3:* As soon as the entity leaves the overlap and enters completely into



(a) Zones with associated resources



(b) Migration of an entity from zone A to B

Figure 3.5: Seamless migration between zones

zone B, cyberorg A removes the entity from zone A.

If the entity is a player's avatar, the connection between the player and the host of cyberorg A is transferred to the host of cyberorg B during step 2. In the CyberOrgs-MMOG implementation, the width of the overlapping zone is equal to the diameter of the area of interest (the view range surrounding the player in 2D case) of the player. This allows the involving zones to have enough time for a seamless zone migration. The width of the shared region and a player's area of interest are adjustable system parameters and game owners can adjust these parameters according to their quality of service requirement. The seamless migration can be further facilitated by incorporating caching mechanisms on game client during the connection transfer phase.

3.2.3 Load Manager

The Load Manager is responsible for monitoring the load and makes load balancing decisions. Based on the decisions made by the Load Manager, the Zone Manager divides the zones into sub-zones and takes appropriate actions to keep the load balanced. To facilitate load management, each zone is monitored using an actor that keeps track of the size of the zone, population, total number of interacting entities and the average server response time. Based on the information the monitoring agent provides, the Load Manager points out the location and shape of a possible hotspot (previously mentioned in Chapter 2). The Zone Manager, using the feedback from the Load Manager divides the zone into sub-zones. Performance parameters such as *response time*, *average outgoing bandwidth* etc. can be predefined, which allows the Load Manager to make decisions on when to look for new resources. Each of the sub-zones are encapsulated by one or more cyberorgs. CyberOrgs' *isolate* primitive is used for this purpose.

On the other hand, an underloaded zone can be aggregated with another underloaded zone. The aggregated zones do not necessarily have to be neighbours as this is not a spatial aggregation. In both cases, the parent zone must be informed about

any kind of division or aggregation.

3.3 Broker

The broker provides resource coordination services to game players, resource owners and game owners, and hence, allows them to buy and sell resources in an open market. The broker is also responsible for providing separate graphical interfaces to the interacting participants for any game instance. Each of these participants is a software client working on behalf of the users. These participating users are divided into three categories based on the ownership of resources and computations:

1. *Resource owner*
2. *Game owner*
3. *Game player*

A resource owner registers its resources through the broker and the information about available resources is stored in the Directory Manager (discussed in the following section). Both the broker and the Directory Manager are distributed components. A broker can search in multiple Directory Managers and a Directory Manager can provide information to multiple brokers. A game owner registers its game following the same procedure. It can also search for available resources and can negotiate for acquiring resources, which is done through the broker. Both a resource owner and a game owner can negotiate on the terms of the contract. Once both parties agree on the contract, resource is allocated for the game and a certain amount of eCash is consumed by a resource owner according to the terms of the contract. A game owner can start the server on the allocated host according to the terms of the contract.

The services provided by the broker include registering and search services for the resource and game owner, payment methods, default policies for resource allocation, customization options and storing policies to a repository etc. The broker interacts with the Directory Manager to search for available resources, and helps a game owner to find out appropriate resources to accomplish his goals.

A game player can look for available games or different versions of the same game. Different versions of the same game can be made available by a game owner at varying costs, because they may involve varying resource consumption. Therefore, a player also gets a certain level of resource control. In other words, a game player controls their game experience by controlling the payment it makes. Once a game player decides on a particular game and pays for it, he can connect with the appropriate host and a game session is started. It should be mentioned that after joining a game session all in-game communication is done between the host and a game player.

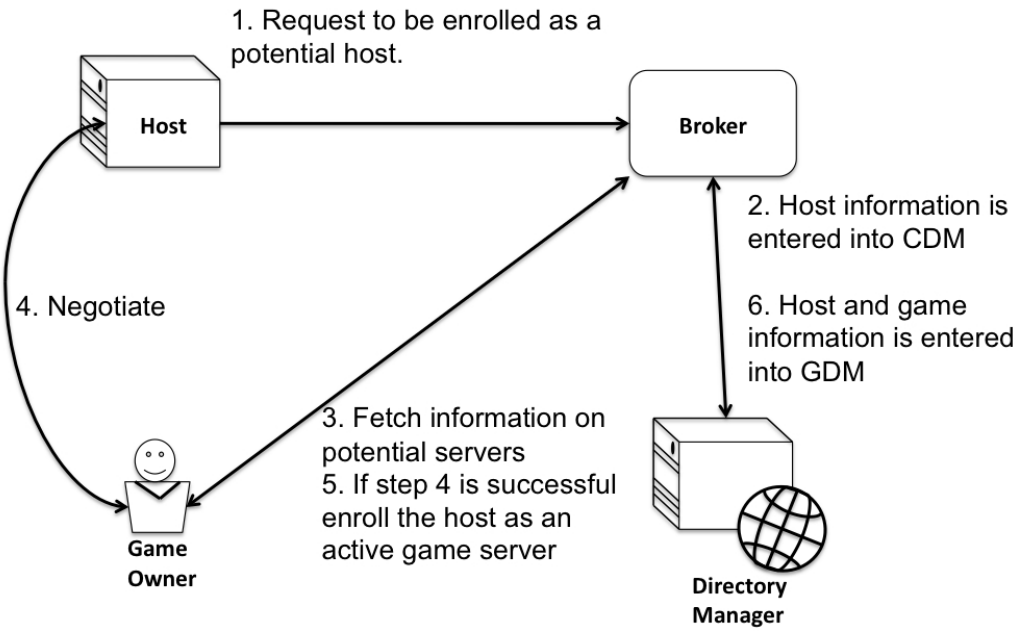
Figure 3.6a and 3.6b show the step-by-step flow of information among different parties. One crucial part of these communications is to maintain a consistent game state for multiple hosts. This is the case that might appear when the game world becomes large and population increases to a point where a single server can not serve the purpose. We adopt a combination of *zoning* and *replication* [16, 28, 29] and exploit the advantages of limited visibility (Area of Interest) to keep the game servers consistent. The approaches used for consistency management are discussed in one of the following sections.

3.4 Directory Manager

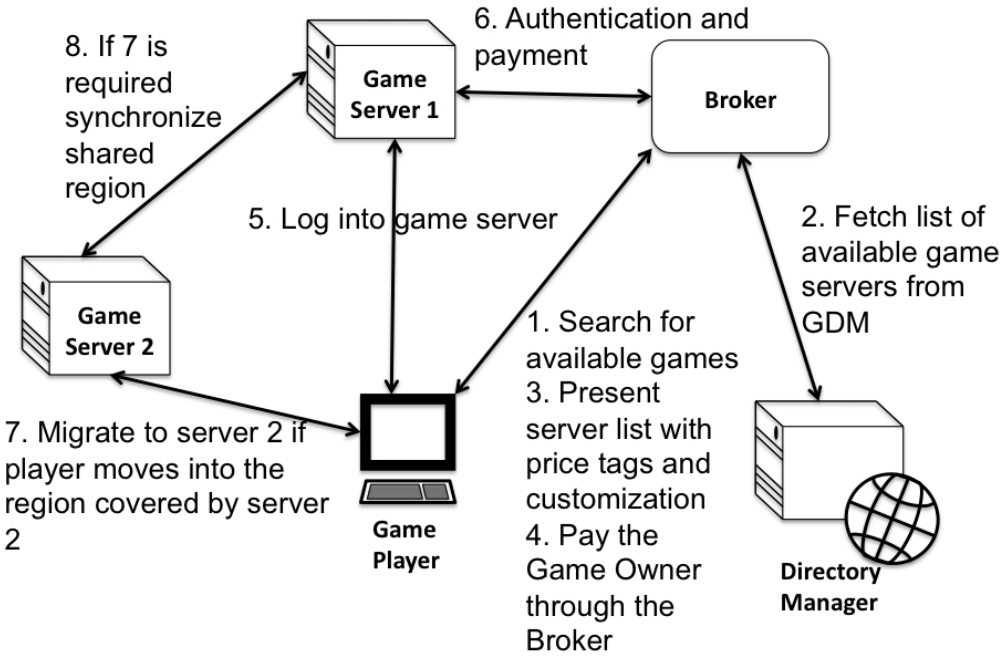
The Directory Manager provides yellow pages services to the broker. These services are mostly in response to the requests for available resources, games and hosts. The responses are processed and forwarded to the appropriate individuals in a presentable format. The Directory Manager is divided into two sub-components: *Cyberorg Directory Manager* (CDM) and *Game Directory Manager* (GDM).

The CDM keeps track of the available resources and contracts associated with them. Each of the available resources is encapsulated by a cyberorg entity. The CDM stores the name and location of the cyberorgs along with the hosting cost.

On the other hand, the GDM stores information on the available games, available hosts along with the final contracts between a resource owner and a game owner. This sub-component stores the name of the games as well as the name and location



(a) Flow diagram for buy/sell of resources



(b) Flow diagram of a game session start-up

Figure 3.6: Flow diagrams

of the cyberorgs hosting these games.

3.5 Dynamic Resource Coordination Mechanisms

The mechanisms that are developed to support dynamic resource coordination can be divided into three parts- *Encapsulation of Zones*, *Load Detection* and *Migration*.

- *Encapsulation of Computation*

  In the CyberOrgs-MMOG platform, the computation is defined in terms of zones. Previously, in Section 3.2.2 zones are introduced. The CyberOrgs-MMOG platform encapsulates zones in cyberorg entities. A cyberorg hosting a zone is responsible for carrying out all computation required by the players or entities inside that zone.

- *Load Detection*

  The CyberOrgs-MMOG platform monitors the number of players, average response time experienced by the players and the average bandwidth among the resources. Monitoring is done by a separate actor in the platform. The API allows game developers to control load detection by defining thresholds for these parameters. If one of these parameters goes beyond the defined limit an overload condition is detected. This mechanism is carried out by the load manager as discussed in Subsection 3.2.3.

- *Migration*

  Migration of a zone happens when an over-load or under-load condition is detected by the load manager. The CyberOrgs platform, discussed in 3.2.1 is responsible for carrying out the migration process in a seamless fashion. The zone manager divides the over-loaded zone into two or more sub-zones by identifying the load characteristics. These sub-zones are encapsulated within new cyberorg entities and migrated to available resources. Similarly, when under-load condition is detected zones may be merged to increase resource utilization. Detail discussion on the CyberOrgs platform can be found in 3.2.1.

3.6 Chapter Summary

In this chapter, we described the implementation of the CyberOrgs-MMOG API. The design of the system and implementation details are also presented. The implementation is constructed on CyberOrgs API and Actor Architecture, which is an implementation of Actors. This chapter also discusses other actors in the system such as the broker and the Directory Manager along with their respective responsibilities. This chapter also presents the flow of information among different clients. Approaches used in the CyberOrgs-MMOG API to divide zones, support for seamless migration of an entity and migration of an entire zone are also discussed thoroughly.

Chapter 4

Application Programming Interface

The CyberOrgs-MMOG implementation provides application programming interfaces for creating cyberorgs, and allows encapsulation of resources and computations. Moreover, the API facilitates resource coordination between computations in an open market of resources. The API offers an interface for the game developers to customize load balancing operations and allows them to exercise fine-grained control on the distribution of zones. In the following sections, these interfaces are discussed in detail. Most of these interfaces, implemented as methods encapsulated in actors can be accessed by sending messages to the appropriate actors. Actors can be extended to employ resource coordination mechanisms based on the game experience. These are discussed in this chapter to demonstrate the flexibility CyberOrgs-MMOG API offers for coordinating resources.

4.1 APIs for cyberorgs creation and primitives

Like a CyberOrgs platform, a CyberOrgs-MMOG platform supports two types of creation: cyberorg creation and actor creation. The first cyberorg (root cyberorg) is created at the startup of the platform. It can also be created using the available graphical user interface or from a user program. Subsequent cyberorg creations result from invocation of the *isolate* primitive. Actor creation can be triggered by other actors in a platform or a *facilitator* actor in case of a cyberorg. A *facilitator* actor is a special actor that performs administrative tasks for a cyberorg [23]. The administrative tasks include invoking CyberOrgs primitives such as creation of a new cyberorg, migration of a cyberorg and assimilate to a parent cyberorg. Param-

eters that are used to make a load management decision- such as response time and bandwidth- can be configured through a configuration file, which the developer may use to enforce certain performance goals.

CyberOrg creation :

```
CyberOrg createCyberOrg(long ticks, long ticksrate, long eCash,
String facilitatorClass, Object[] args)
```

where *ticksrate* is the rate of processor time that the new cyberorg would receive with respect to the cyberorg in one scheduling cycle; *eCash* is the number of eCash units that are used for buying processor resource from host cyberorg; *facilitatorClass* and *args* identify the facilitator actor class for the cyberorg and the arguments for creating such a facilitator actor.

Actor creation:

Each actor in the CyberOrgs-MMOG platform represents a computation. Actors can be created either by another actor or a cyberorg. Both of the variations are discussed below:

- `ActorName createActor(ActorName creator, String actorClass, Object[] args)`

  where *creator* is the unique actor name of the creator; *actorClass* and *args* specify the class of the actor being created and the arguments used in the actor constructor. This method is used by one actor to create another actor.

- `ActorName createActor(CyberOrg host, String facilitatorClass, Object[] args)`

  where *host* identifies the creating cyberorg; *facilitatorClass* identifies the actor class of the facilitator and *args* specify the arguments to be used in constructing the facilitator actor. This is used by the cyberorg constructor at the time of cyberorg creation.

CyberOrgs Primitives:

Primitive operations are called by the facilitator actor of a cyberorg.

- **Isolation:**

  ```
  CyberOrg isolate(long eCash, ActorName[] actors, Contract newContract)
  ```

  where *eCash* is the amount of eCash that is given to the newly created cyberorg; *actors* is a list of existing actors that will be isolated into the new cyberorg; *newContract* is the contract imposed on the new cyberorg and its host cyberorg, which specifies the ticks, ticks rate of processor as well as total length of data and data rate (for network resource control) that the new cyberorg receives, as well as the cost of the resources in terms of eCash payments to be made.

- **Assimilation:**

  ```
  CyberOrg assimilate()
  ```

  This primitive will cause assimilation of the cyberorg into its host. Invocation of this primitive results in releasing all resources, eCash and computations to the parent cyberorg.

- **Migration:**

  ```
  void migrate(ActorName facActorOfDestCyberOrg, Contract newContract)
  ```

  where *facActorofDestCyberOrg* is the name of facilitator actor in the destination cyberorg, which serves as the cyberorg's name; *newContract* is the negotiated contract between the migrating cyberorg and the intended host.

Negotiation

Every migration starts with a negotiation between a cyberorg and a prospective host cyberorg. The negotiation is invoked by the facilitator actor of the migrating cyberorg.

```
Contract negotiate(ActorName destFacilitatorActor)
```

Here *destFacilitatorActor* is the facilitator actor of the prospective future host CyberOrg. A *Contract* object holds 3 attributes -

- *Ticks*- The amount of resources that are given to the child cyberorg in one scheduler cycle.
- *Ticksrate*- The rate of resources that the child cyberorg would get with respect to the root cyberorg.
- *Price*- The virtual price of the resource in terms of *eCash*.

4.2 API for Resource Owner

The following invocations allow registering and deregistering of potential cyberorgs that have available resources :

```
void register(CyberOrgTuple p_ctTuple)
void deregister(CyberOrgTuple p_ctTuple)
```

Here *p_ctTuple* is an instance of CyberOrgTuple that encapsulates necessary information such as the IP address, associated contract and the facilitator actor of a cyberorg. A resource owner can send a request to register its resources to CyberOrg Directory Manager through a broker. A Broker provides "registerToCDM" method for this purpose.

```
void registerToCDM(CyberOrgTuple cyberOrgTuple, ActorName sender)
```

Here, *sender* is the name of the actor that sends the message on behalf of a resource owner; *cyberOrgTuple* encapsulates the information required to access the resource.

4.3 API for Game Owner

Game owners are the publishers or developers of the game. Initially, a game owner sends a request to a broker to find out what are the potential resources available.

the broker receives this request, in response it returns a list of cyberorgs that are currently registered in CDM. Upon receiving this list, a game owner proposes a contract to each of the cyberorg from the list and keeps doing this until a successful negotiation is made. Once a contract is signed, the cyberorg is inserted into GDM and removed from CDM. Following are the interfaces provided by a broker to facilitate these services.

```
void getNewResource(Contract contract, String gameName,
ActorName sender)
```

Here, *contract* represents the search criteria for potential resources; *gameName* is the name of the game and *sender* represents the actor responsible for making requests on behalf of a game owner.

```
void negotiate(CyberOrgTuple cyberOrgTuple, Contract contract,
ActorName sender)
```

This method provides an interface to the negotiation process. Here, *cyberOrgTuple* identifies the potential resource with a proposed *contract*.

```
void registerToGDM(CyberOrgTuple cyberOrgTuple)
```

This method registers a cyberorg, identified by *cyberOrgTuple* with additional game related information into the GDM.

4.4 API for Game Player

A game player initially requests a broker for the available games that are provided by different game owners. The broker then searches in the GDM for resource that matches the price tag mentioned in the contract. This search result is presented to the player as a list of games. Once a game is chosen, a list of available hosts is presented. Now game player can select a particular host from this list and get connected to the chosen host. During this connection phase, the player pays the respective game owner. Following interfaces are provided to a player.

```
void getGameList(Contract contract, ActorName sender)
```

This method searches for appropriate games according to the *contract* where *sender* represents the player actor. The contract object facilitates controlling of game experience by controlling the resources it consumes.

```
Object[] getServerList(String gameName, Contract contract)
```

This method returns a list of available servers for a particular *gameName* according to the specified *contract*.

4.5 APIs for Zone Division and Load Management

This part of the API provides interfaces to access and update zone related information such as population of the zone, size of the zone etc. It also consists of an algorithm to divide zones in smaller sub-zones. This algorithm currently creates rectangular zones. Developers can implement their own algorithm based on their needs. These methods are implemented in a static class `ZoneDivider`. In this section, some of these interfaces are discussed.

```
ZoneBox createZoneBox(String zoneName, String ip, int x, int y,
int width, int height)
```

This method creates a `ZoneBox` object, which stores metadata of a zone such as size, IP address of the cyberorg encapsulating a zone. *zoneName* represents the name of the zone, where *x*, *y*, *width* and *height* represents the rectangle that defines the size of the zone. IP address can be obtained as an available resource by accessing broker services. It should be noted that the ZoneBox class does not represent the actual zone. It comprises of data and methods required by the zone division algorithm.

```
void incrementPopulation(RPObject rPObject)
```

This method allows updating the population of the zone and is called when a new player joins a zone. This event is detected by an actor that keeps track of the changes occurring in every turn. Here, *rPObject* is an instance of RPObject representing the new player entity.

`void divideZoneStrategy()`

This method defines the zone division strategy. The cutoff performance paramenters such as population limit, bandwidth and response time can be defined in the configuration file. In the following section the format of the configuration file is shown.

`void resetPopulation()`

The population of a zone can be reset to zero by invoking this method. This is mostly used during a new zone creation.

Moreover, `ZoneDivider` class implements methods to detect if an entity representing a player is in the region shared by two zones, to define the extension of the boundary of a zone that is replicated from another zone, to identify a player's zone etc. Load Manager detects a load condition by accessing these methods.

4.6 Configuration File

A file named "server.ini" provides easy access to configure the system-wide properties. Following is an example of a typical configuration file. The maximum number of players for each zone can be specified in this file. The parallel connection limit poses a restriction on how many simultaneous connections can be made from the same client machine. For the purpose of simulation, the limit is set to 1000 as most of the players are simulated clients from the same machine. In real world gaming, this has to be set to a much smaller number to prevent flooding or cheating. Zone size defines the number of players in a zone that initiates a load balancing task. Same is the case with repose time cutoff and bandwidth cutoff. However, only one of these parameters can be set.

server.ini

```
max_number_of_players=500
parallel_connection_limit=1000
zone_size=150
reponse_time_cutoff=60ms
bandwidth_cutoff=80KB/s
jdbc_url=jdbc:mysql://localhost:3306/marauroa
jdbc_class=com.mysql.jdbc.Driver
jdbc_user=root
jdbc_pwd=123
tcp_port=5555
turn_length=220
```

Other properties that can be set are the information to access the database, length of each turn etc.

Chapter 5

Experimental setup

In this chapter, design and development of the simulation environment are discussed. Section 5.1 illustrates APIs and software used in the experiment. Section 5.2 presents specification of the hardware used for the experiment. Section 5.3 describes the simulation environment along with its limitations.

5.1 Software Platform

In this section, Actor implementation such as Actor Architecture and resource coordination API such as CyberOrgs API are summarized. Because both CyberOrgs API and Actor Architecture are based on Java's concurrency, it was preferred to use a game developed in Java. *JMaPacman* [1], an open source multiplayer version of the popular 2-D game *Pacman* is used to illustrate the mechanisms supported by CyberOrgs-MMOG API. *JMaPacman* is developed using *Marauroa* [2], an open source game engine for multiplayer games. In this section, both *JMaPacman* and *Marauroa* are summarized.

5.1.1 Actor Architecture

Actor Architecture [7] is Java-based framework that enables a programmer to write actor programs in popular Java syntax. It is actively being developed by the Open Systems Laboratory at the University of Illinois.

Actor model of programming comprises of concurrent entities called actors. Actors can communicate with each other using asynchronous message-passing. An actor does not share its state with other actors and therefore, the model is free from low-

level data races. Actor Architecture provides an execution environment for actors and it supports actor primitives, such as sending and receiving messages, creating new actors, and changing local state. An instance of Actor Architecture run-time system is called an AA platform.

Actor Architecture uses fair scheduling techniques to schedule messages. It provides pattern-matching services to search and identify an actor that can be located in a local platform or a remote platform.

5.1.2 CyberOrgs API

In Actor Architecture, resource allocation relies on the underlying Java Virtual Machine (JVM). To support resource coordination, Actor Architecture is extended to CyberOrgs [23], which makes the resource control visible to the programmers.

CyberOrgs is developed by adding two key components to the Actor Architecture: CyberOrg Manager and Scheduler Manager. Each instance of a CyberOrgs platform comprises of entities called cyberorgs. A cyberorg encapsulates a set of actors, resources (CPU cycles) and some eCash to buy more resources from another cyberorg. CyberOrg Manager is the central component of each CyberOrgs platform. All resource coordination operations are carried out by the CyberOrg Manager. The results of such operations are sent to Scheduler Manager, which schedules all actors in the platform according to these results. The mechanisms provided by CyberOrgs API includes creation of a new cyberorg entity (isolation), adds mobility to a cyberorg (migration) and merging of two cyberorgs (assimilation). CyberOrgs API can be downloaded from.[1]

5.1.3 Marauroa Game Engine

Marauroa is an open source multiplayer game engine based on Arianne,[2] an online gaming framework. It consists of a multiplayer online game engine to develop turn

[1] Xinghui Zhao, http://agents.usask.ca/Agents_Lab/Agents_Lab_-_CyberOrgs.html, access date=7/6/2012

[2] Arianne, http://arianne.sourceforge.net/, access date=7/6/2012

based and real time games, and the various games, which use it. Marauroa provides perception based client-server communication, asynchronous database persistence and handles object management.

Marauroa uses a multithreaded server architecture, TCP socket based communication and a MySQL or H2 based persistence engine. Marauroa, developed in Java provides a flexible game system that allows the developers to extend and modify the game engine. Game scripting can be done using either Java or Python.

Marauroa is based on a philosophy called Action/Perception, on each turn a perception is sent to all clients connected to a server explaining the surroundings (obstacles, entities, power-ups etc) around them. Clients can request the server to do any action in their names. The result of an action may change the state of the game, which is notified to the clients in the next turn. Marauroa is totally game agnostic, i.e., it makes very little assumptions about the type of game under development, allowing a great freedom in creating games of any genre. Further details are available online [2].

5.1.4 Jmapacman

JMaPacman is a multiplayer remake of the popular classical game *Pacman*. Although JMaPacman is a clone of the old arcade game Pacman, it is built on new game design principles and features. JMaPacman supports multiplayer functionality and allows developers to build customized maps. JMaPacman uses Marauroa game engine to add multiplayer functionality and Java2D to render graphics. Figure 5.1 shows a screen shot from the game.

Each JMaPacman instance consists of a game world. This game world comprises of a number of zones. The CyberOrgs-MMOG API assists the zones from the same instance to be distributed over multiple servers by sharing the state of a zone. Dynamic resource coordination is triggered when a particular area of a zone gets over-populated or under-populated. Dynamic resource coordination mechanisms, provided by CyberOrgs-MMOG API allows detection of these load conditions, division of the zones into smaller zones or merging into a larger zone and therefore,

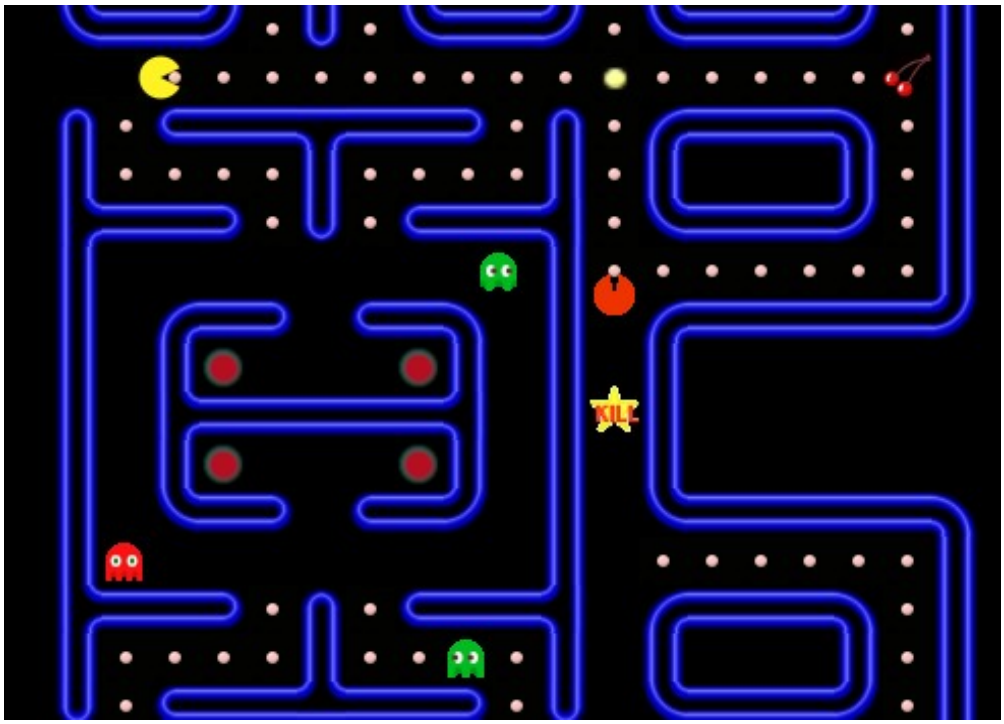


Figure 5.1: Screenshot taken from JMaPacman [1]

distributes the computational load among multiple servers. Each of these zones is encapsulated within a cyberorg entity allowing resource coordination with other cyberorgs.

5.2 System Configuration

Detailed specification of the system used are discussed below:

Servers:

A cluster of 4 Apple XServers operated by Mac OSX servers each with 2 x 2.8 GHz quad-core Intel Xeon processors and 8GB of RAM. All of these machines were used as game servers (resources).

Client generators:

- 1 Mac OSX server with the same configuration as above

- 1 iMac with Mac OS X v10.5 Leopard, 3.06GHz Intel Core 2 Duo processor and 4GB of RAM

- 1 MacAir with Mac OSX v 10.6.8 Snow Leopard, 1.8 GHz Intel Core 2 Duo processor and 2 GB or RAM

All of these machines were used for creating simulated clients.

5.3 Simulation

There are three types of users who can access the CyberOrgs-MMOG platform - resource owners, game owners and game players. For a particular game, there could be multiple resource owners associated. Same is the case with game players. All of these users connect with the CyberOrgs-MMOG platform through their respective clients. However, this requires a large number of users to properly study the performance of the API. Due to time constraints, instead of going through a user study, a simulator is developed to automate the ownership and resource coordination process (mechanisms supported to game owner and resource owner).

The simulator has two components - one for automating the resource coordination (integrated with the server) and one to generate simulated clients (as a separate application). Each client is a Java thread that sends random actions to the server and receives perceptions. Each of these threads encapsulates a JMAPacman client. As these clients do not need to render the game on a display device, they are simplified from the original implementation of JMAPacman client. The clients are configured only to send and receive data. Clients are generated with an interval of 0s to 20s which is uniformly distributed. On the other hand, the server-side component collects statistical data from the connected clients, which is used to evaluate the API.

Actions are to navigate the map, which can be any of *up*, *down*, *left* and *right*. Perceptions are basically list of changes in attributes of entities in the player's surrounding. These entities include *walls*, *power-ups*, *enemies* and other *players*. Based on the changes in these surrounding entities a simulated player can choose one of the

available actions. If there are multiple available actions that can be executed one of them is chosen in a random fashion. This approach is taken to have the simulated clients behave as close as to the real players. The interval between two consecutive actions is also a random number generated using an uniform random distribution with an interval between 0s to 10s.

Chapter 6

Experimental results

This chapter presents the experimental results of the simulation. All of the experiments were performed using the simulator briefly discussed in Section 5.3. The organization of this chapter is as follows: In Section 6.1 unique features provided by the API are summarized. Section 6.2 discusses about the design of the experiment. In Section 6.3, various QoS parameters such as response time and average outgoing bandwidth are evaluated in respect of CyberOrgs-MMOG implementation. Mechanisms provided by CyberOrgs-MMOG API enables multiple peers to coordinate resources. Section 6.4 presents the performance analysis of the API mechanisms.

6.1 CyberOrgs-MMOG API

CyberOrgs-MMOG API comes with a library that allows integrating resource control mechanisms for Massively Multiplayer Online Games. Following are some of the features come with the API.

- *Improved programmability:* The API allows game developers to integrate sophisticated functionalities, which adopts to existing proven models and approaches. By properly configuring the API developers can build scalable MMOGs without compromising the game performance. However, when integrating the API-supports to an existing MMOG, developers have to make sure that the game design allows spatial distribution of the game world and provides enough abstraction so that the API can encapsulate and control the distribution of game computation.

- *Resource coordination mechanisms:* The API supports mechanisms that allows acquisition of resources during a overload condition in a flexible and seamless way. Similarly, mechanisms to release resources are also integrated within the API. The addition and release of resources can dynamically control the game performance even in case of hour-to-hour fluctuations. The results on the evaluation of these API mechanisms are discussed later in this chapter.

6.2 Experiment Design

In order to demonstrate the mechanisms supported by the CyberOrgs-MMOG API, the experiment is designed to show that the quality of service requirements can be fulfilled without significant overhead. Two of these performance parameters are evaluated in this experiment - average response time and average outgoing bandwidth.

Response time represents the delay that a player experiences when sending a request to a server. In short, response time is the time difference between a player's action and a server's response to that action. It is expected that the average response time meets the requirement of the game's rendering rate. The game client should be able to render in such a rate so that a player does not experience any lag. The mechanisms supported by the CyberOrgs-MMOG API facilitates acquisition of resources whenever the response time goes beyond a certain limit and vice versa. Similarly, average outgoing server bandwidth can be controlled. Usually, for a multiplayer game server the outgoing data packets contain more data than incoming requests from a game player. Therefore, outgoing bandwidth plays a vital role in keeping a high-performance gameplay.

In the following section, these two performance parameters are evaluated to show that these parameters can be used to control the game performance. The overhead caused by these mechanisms are analyzed in Section 6.4. The overhead is observed by comparing the average response time and outgoing server bandwidth with those of a server not integrated with the CyberOrgs-MMOG API. Each of these experiments was performed for up to 20 runs and averaged.

6.3 Controlling QoS Parameters

This section presents the evaluation of game performance parameters by coordinating resources dynamically. The experiment is done with two parameters- average response time and average outgoing bandwidth among servers. At first, each of these parameters are evaluated based on the number of players. A new resource is added or released to the game when the number of players reach a certain threshold. Then we evaluate by defining a cut-off value for these parameters. Following subsections present the results.

6.3.1 Average Response Time

Figure 6.1 and 6.2 show the change in response time against the number of players. A new resource is added when the average number of players per server reaches 50, in other words, when the total number of players reaches a multiple of 50. In these figures, black marks represent addition or release of resource. In this experiment, new players are distributed among the resources in an uniform fashion. From the figure, we can see the response time rises up to 70ms and each time a new resource is added an improvement in response time is observed. Moreover, we can observe from the figure that the peak response times are not equal, though the average load among all servers is the same. The reason behind this can be a resource is loaded by other applications. Another reason can be the uneven distribution of players among the servers as the simulated players can move from one zone to another during a migration (resource addition). On the other hand, Figure 6.2 shows the release of resources in response to decrease in number of players. The experiment starts with 200 simulated players uniformly distributed among 4 resources. Similar to the previous experiment, resources are released when the number of players decreased to a multiple of 50. In this case, the minimum average response time observed is approximately 20ms.

Figure 6.1 and 6.2 show addition or release of resources based on the number of players. However, this approach does not keep the increase or decrease in response

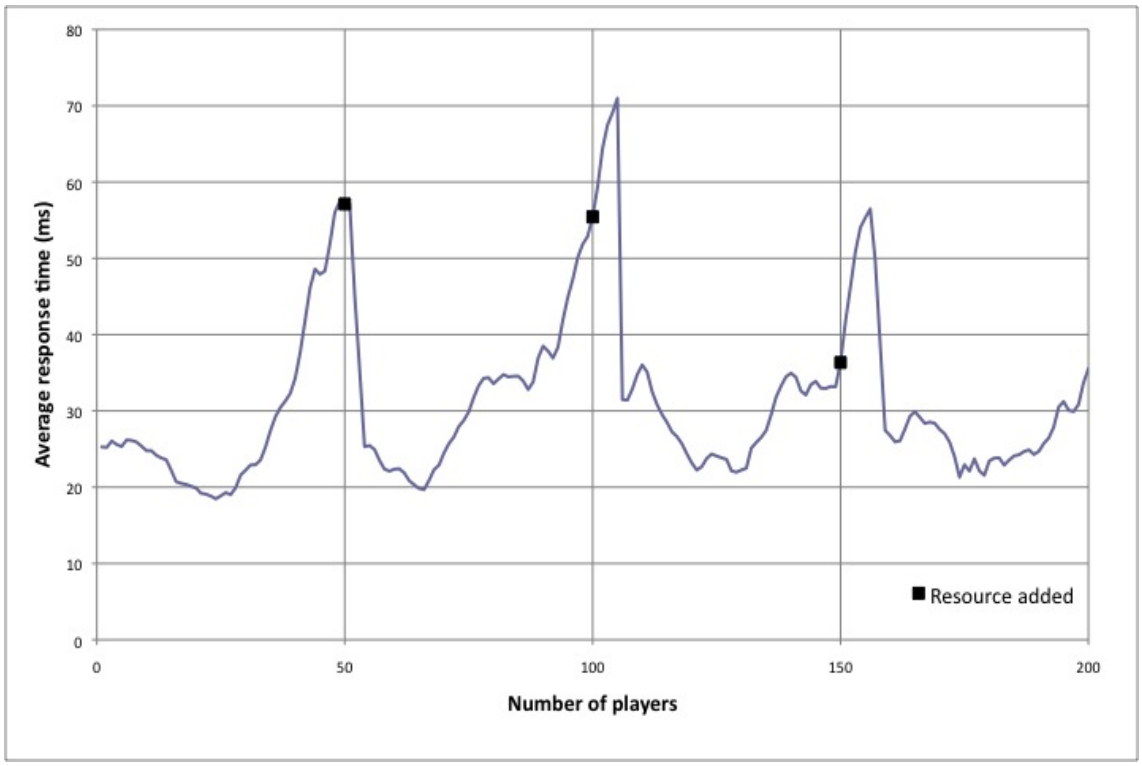


Figure 6.1: Resource added in response to increase in number of players.

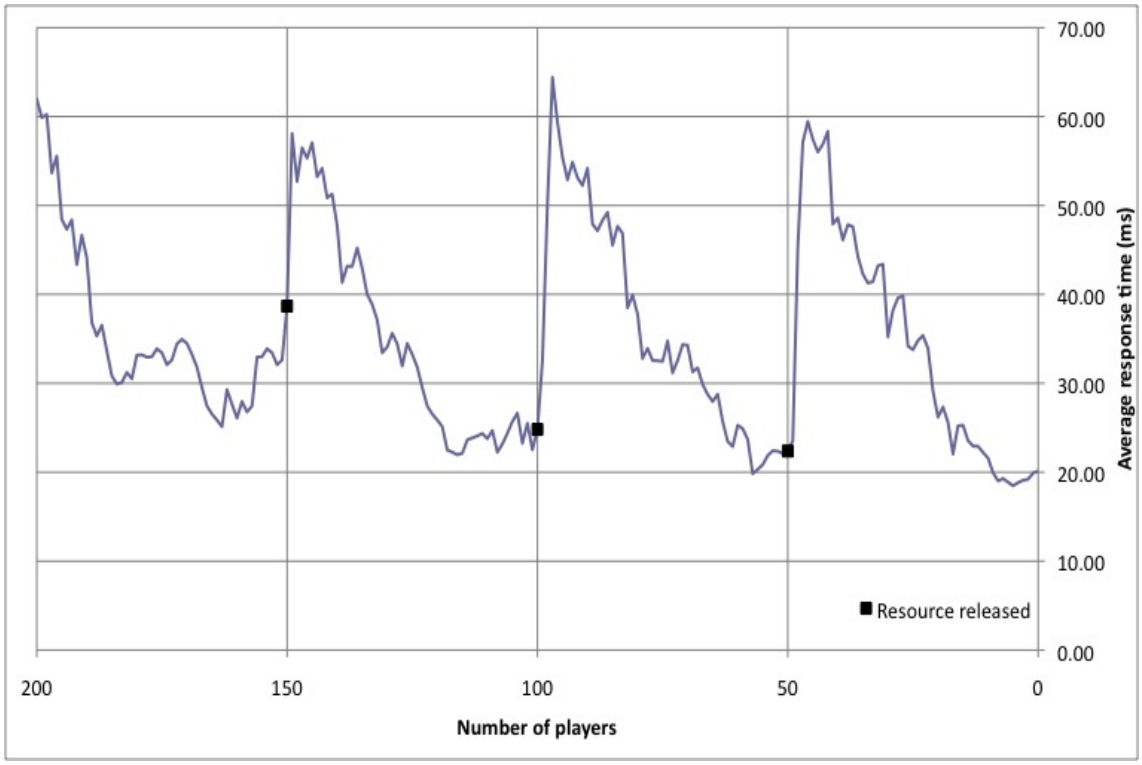


Figure 6.2: Resource released in response to decrease in number of players.

time under control. By controlling the response time a game developer can control the quality of the gameplay. Figure 6.3 and 6.4 show the results of such control in response time. In this experiment, a new resource is added or released only when the response time reaches certain values (can be defined in the configuration file described in 4.6), which in this case are 60ms and 20ms, respectively. These threshold values are set based on the minimum time required to render a frame. According to some studies,[1] the required minimum frames per second is highly dependent on the type of animation in concern. However, often 24fps (frames per second) is used as a standard value in many computer games. This gives the Graphics Processing Unit (GPU) around 42ms time to render and display a frame. Based on this observations, the upper limit for response time is set to 60ms to detect an overload condition. On the other hand, 20ms of response time is set as a minimum requirement while releasing resources. We allowed this space in response time to avoid fluctuations which often develops due to network delays. It is also observed that the number of players may not be the same for every addition or release of resource. Another observation is that the peaks do not happen when a resource is added rather after a few more players are added. This is because the simulator application continuously adds new players in a random time interval of 0s to 20s, which often may occur before a migration (resource addition) process is complete.

6.3.2 Average Outgoing Bandwidth

Figure 6.5 and 6.6 show the change in average outgoing server bandwidth per server against the number of players. As in Subsection 6.3.1, similar setup is used in this experiment. A resource is added or released when the total number of players reaches multiple of 50 and the change in average server bandwidth is observed. As we can see from the Figure 6.5, the average bandwidth rises up to 110KB/s and each time a new resource is added an immediate improvement in average server bandwidth is observed. Figure 6.6 shows that the bandwidth goes down to 20KB/s

[1] http://www.grand-illusions.com/articles/persistence_of_vision/

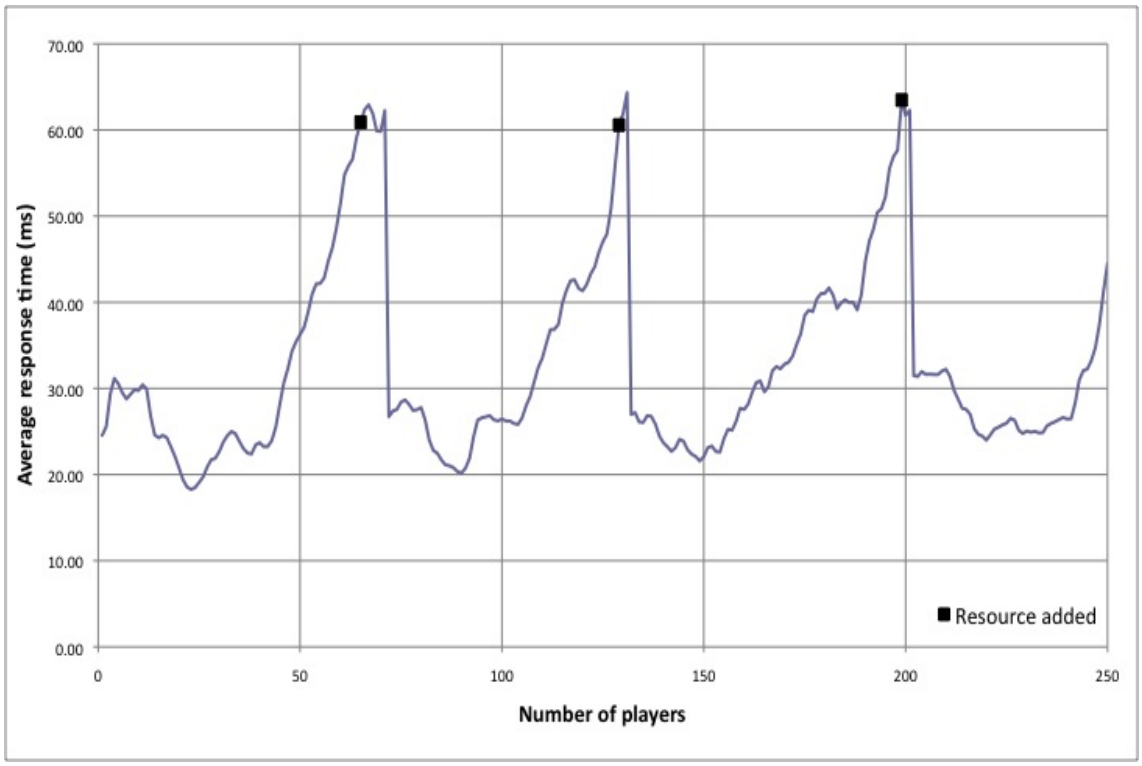


Figure 6.3: Resource added in response to increase in response time.

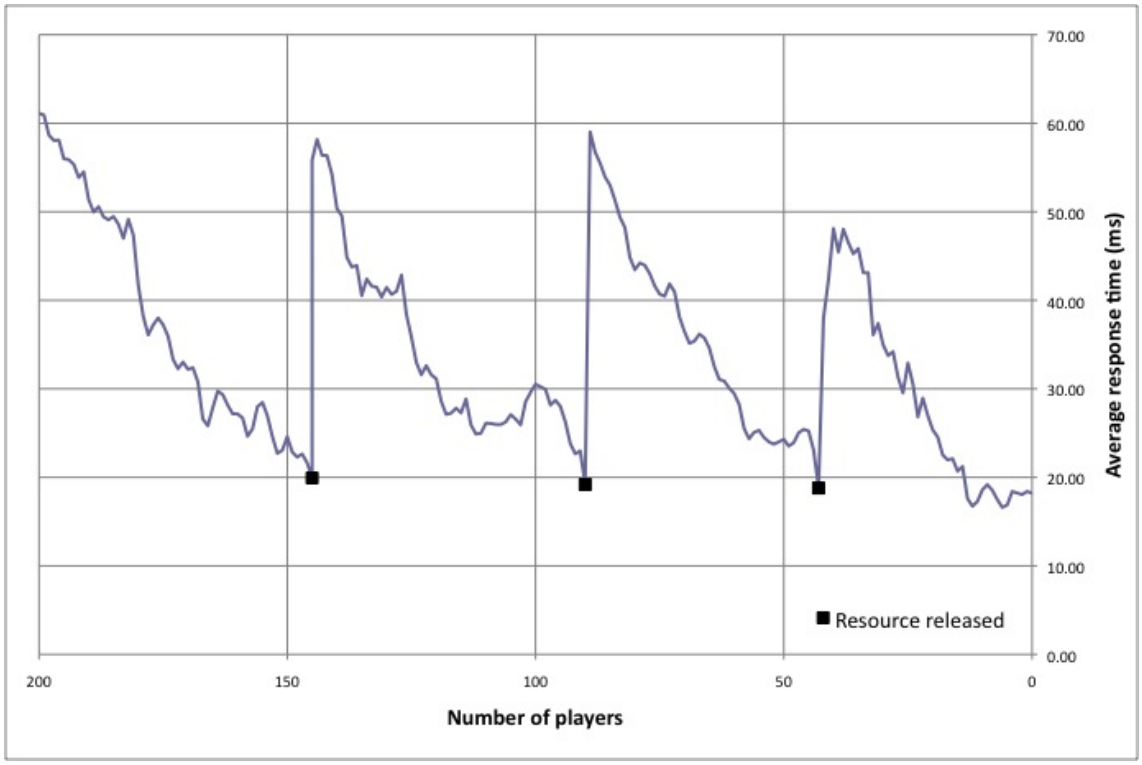


Figure 6.4: Resource released in response to decrease in response time.

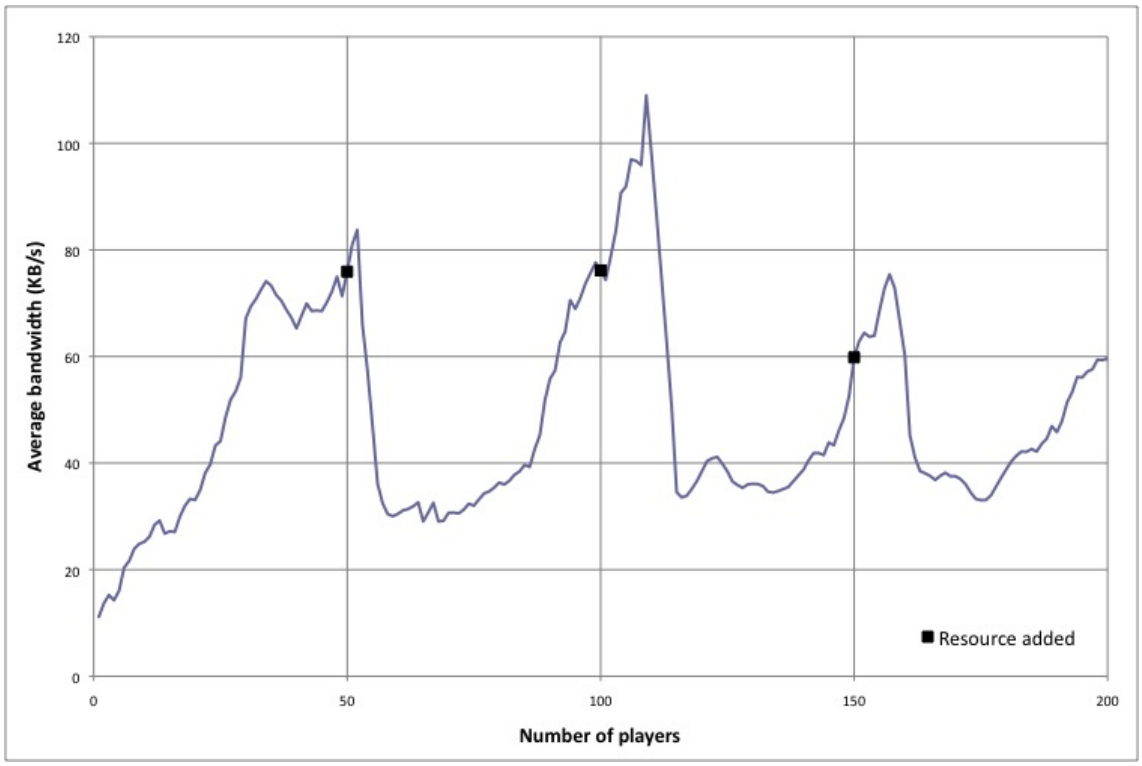


Figure 6.5: Resource added in response to increase in number of players.

while releasing the resources, We also observe that the peak bandwidths are not equal, though the average load among all servers is the same. Similar reasoning as discussed in Subsection 6.3.1 can also be applied to explain this observation.

To keep the bandwidth under control, cutoff values are set to 80KB/s and 30KB/s and the results are shown in Figure 6.7 and 6.8, respectively. As the available outgoing bandwidth is a network resource which can vary from one server to another, these values are set based on the previous approximation of bandwidth requirement when number of players were considered as a control parameter (Figure 6.5 and 6.6). This approach allows the resource coordination decisions to be made only when the average bandwidth exceeds this threshold. By controlling the bandwidth, a game owner can control the network delay experienced by a game player. Same as the experiment with the response time, the threshold bandwidth can be set in the configuration file discussed in Section 4.6.

It is possible to enforce cutoff values for both response time and bandwidth at the same time. A game owner can choose to control one of these parameters or both.

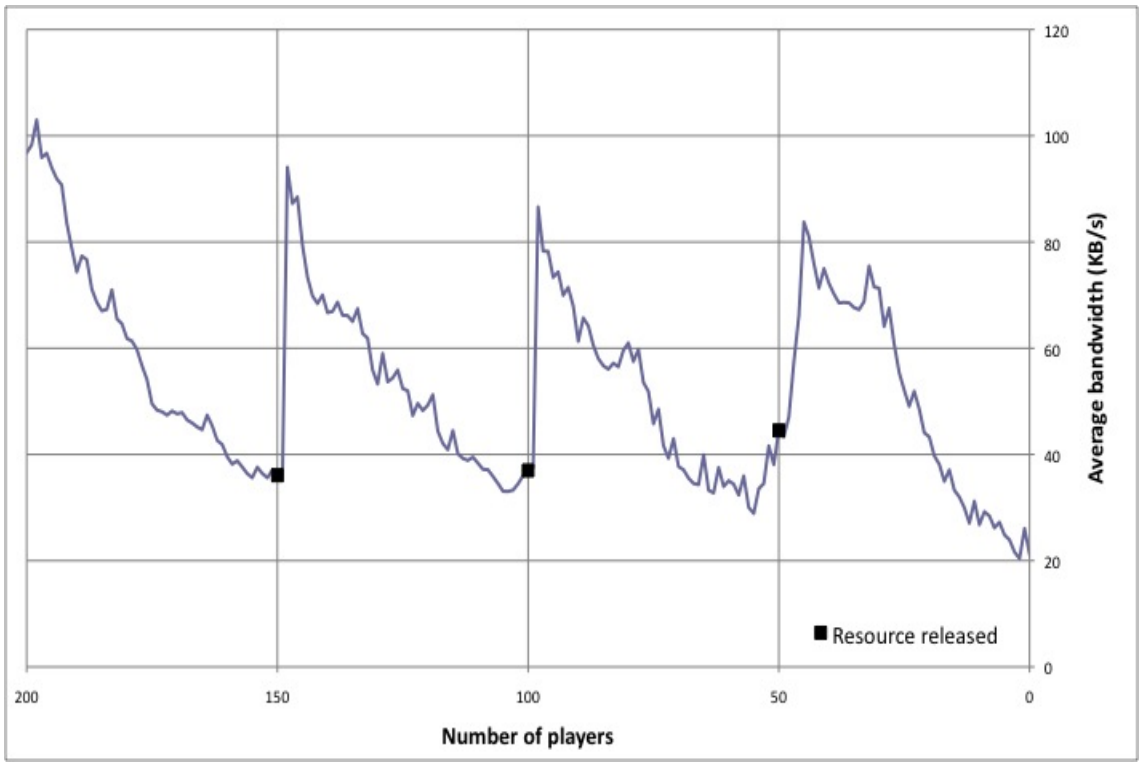


Figure 6.6: Resource released in response to decrease in number of players.

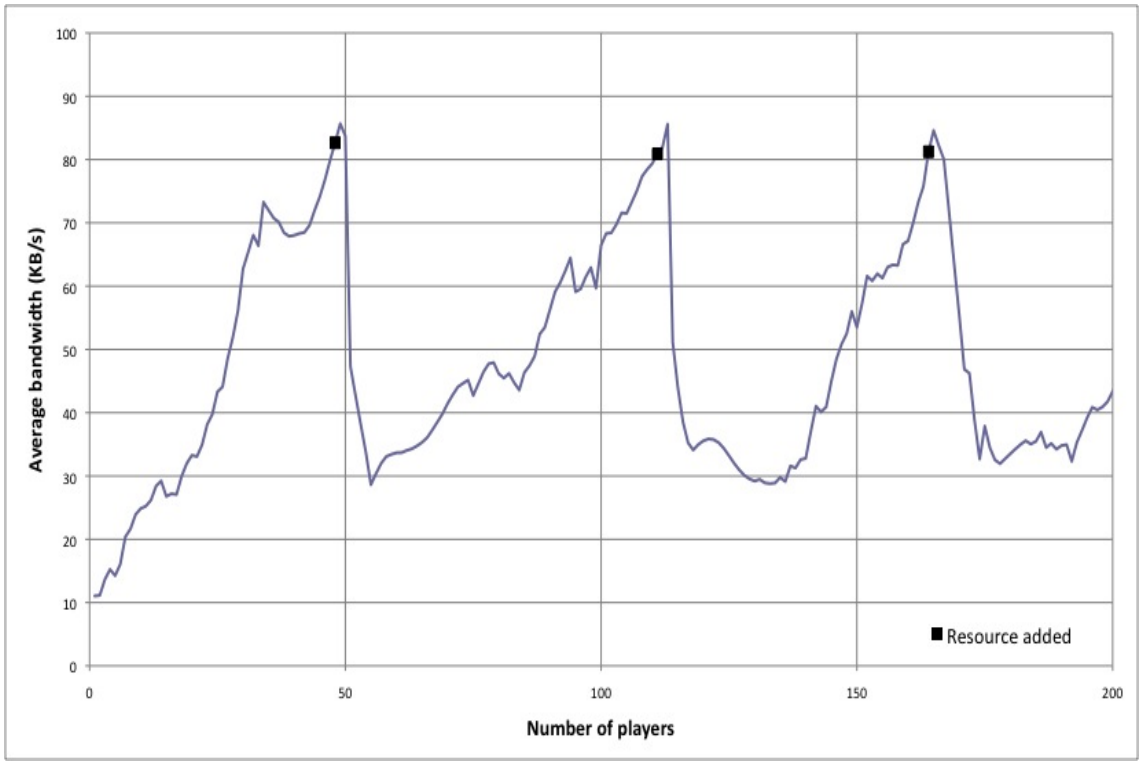


Figure 6.7: Resource added in response to increase in outgoing bandwidth.

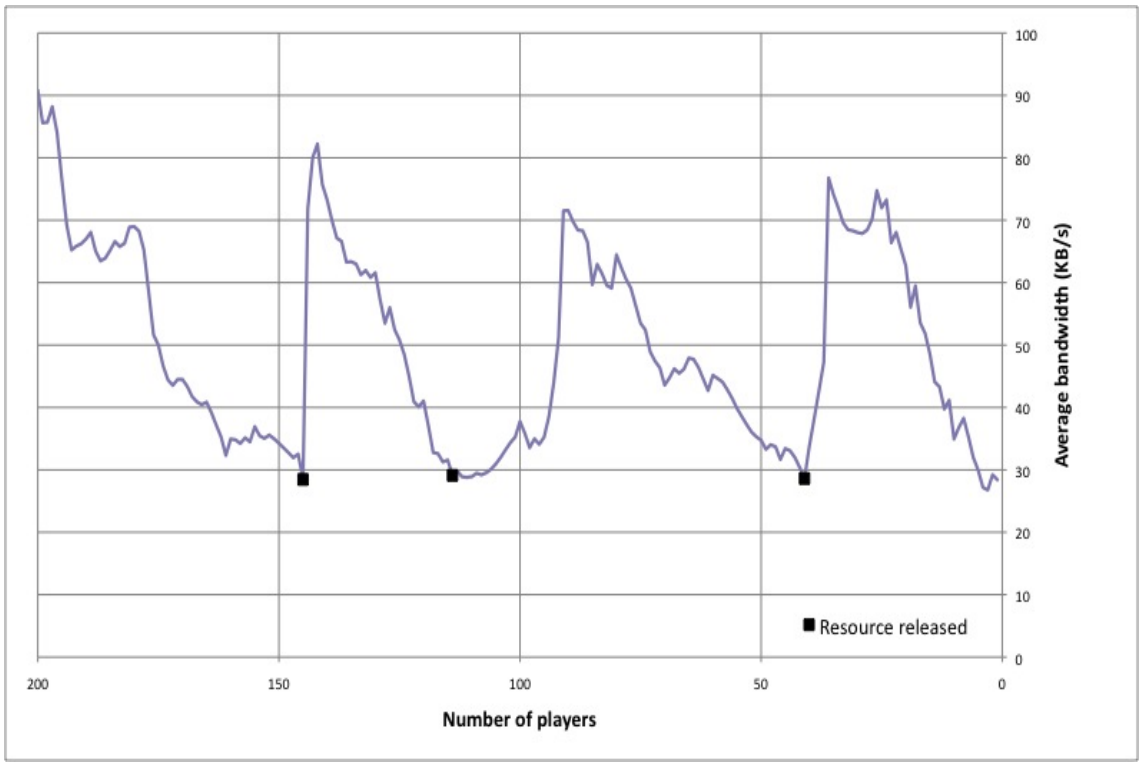


Figure 6.8: Resource released in response to decrease in outgoing bandwidth.

Moreover, other server controlled quality of service parameters can be defined with some modification in the CyberOrgs-MMOG API.

6.4 Performance Analysis

Experiments have been carried out to look at the overhead of using the CyberOrgs-MMOG API. Instead of adding new resources, we observe the change in average response time with increase of players. The data observed are compared with a similar setup without using the CyberOrgs-MMOG API. This section also presents experimental results on the time the API takes to coordinate resources.

6.4.1 Overhead Analysis

Figure 6.9 shows the comparison of response times with or without the use of CyberOrgs-MMOG resource coordination mechanisms. Two separate experiments are carried out. Each of them are executed for 20 runs and averaged for more accurate

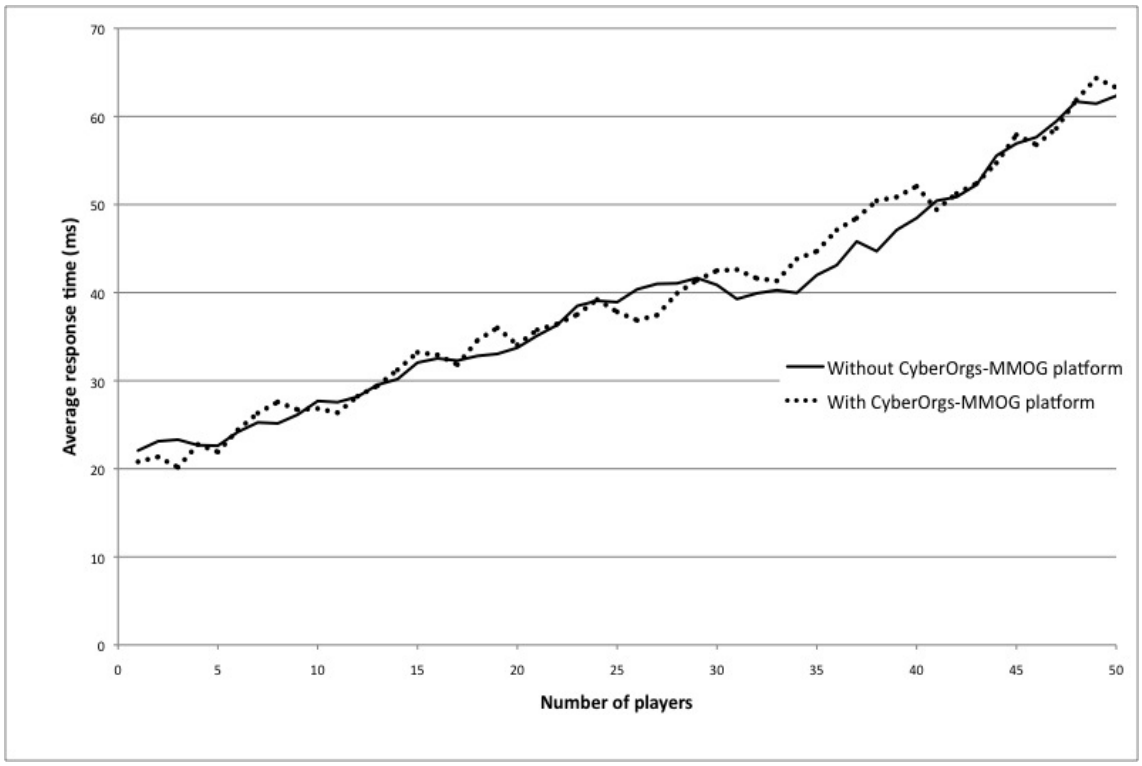


Figure 6.9: Effects on average response time of using CyberOrgs-MMOG API.

results. For the experiment using CyberOrgs-MMOG API, all resource coordination and acquisition mechanisms are enabled except that players were not allowed to migrate. And then the result is compared with the response time provided by the game itself. It is observed that the response times provided by both experiments are very similar. The game with CyberOrgs-MMOG API is configured to make resource decisions when the number of players is more than 30. This triggers the load manager and the zone manager to look for additional resources.

Figure 6.10 presents the result of similar experiment on the outgoing bandwidth. It is observed that the bandwidth requirement is little higher than the game without using the API, specially when the number of players is more than 30. Same as the previous experiment, the game with CyberOrgs-MMOG is configured to make resource decisions and coordinations when the number of players reaches 30. The higher bandwidth requirement is the result of coordination with other resources, migration of computations and synchronization between the zone boundaries.

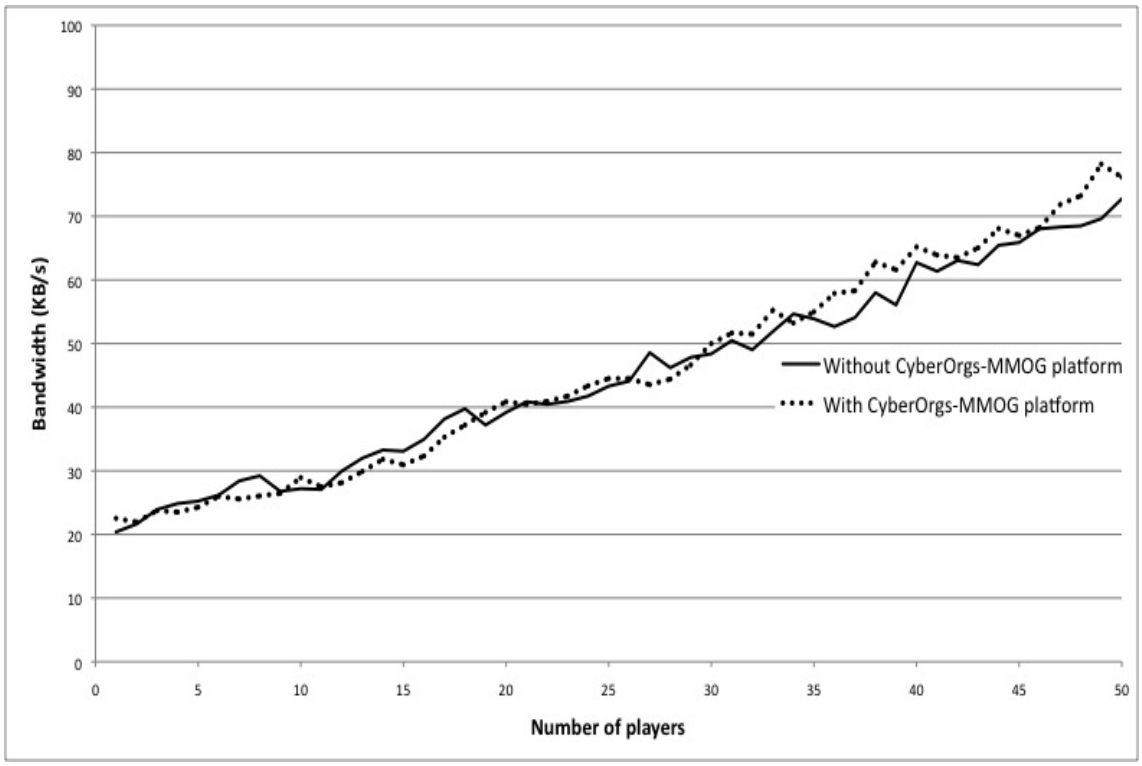


Figure 6.10: Effects on outgoing bandwidth of using CyberOrgs-MMOG API.

6.4.2 Analysis of resource coordination mechanisms

In this section, time taken to execute resource coordination mechanisms is analyzed. These mechanisms include detection of an overload/under load condition, searching for a suitable resource and migration of the computation (a part of the zone). The experiment is done by configuring the game to enable migration for varying zone sizes. Each of these migrations is performed 20 times and averaged.

Figure 6.11 shows the results of this experiment. Dynamic resource acquisition process consists of three mechanisms provided by CyberOrgs-MMOG API. Mean time and standard deviation for each of these mechanisms - Detection of a load condition, Search for new resource and Migration to another resource are shown. The number of players in the zone that is being migrated is also shown.

Table 6.1 presents the time taken for the resource coordination mechanisms with respective standard deviations. It is observed that the mean time taken for detecting a load condition (over/under load) does not vary significantly with the size of the

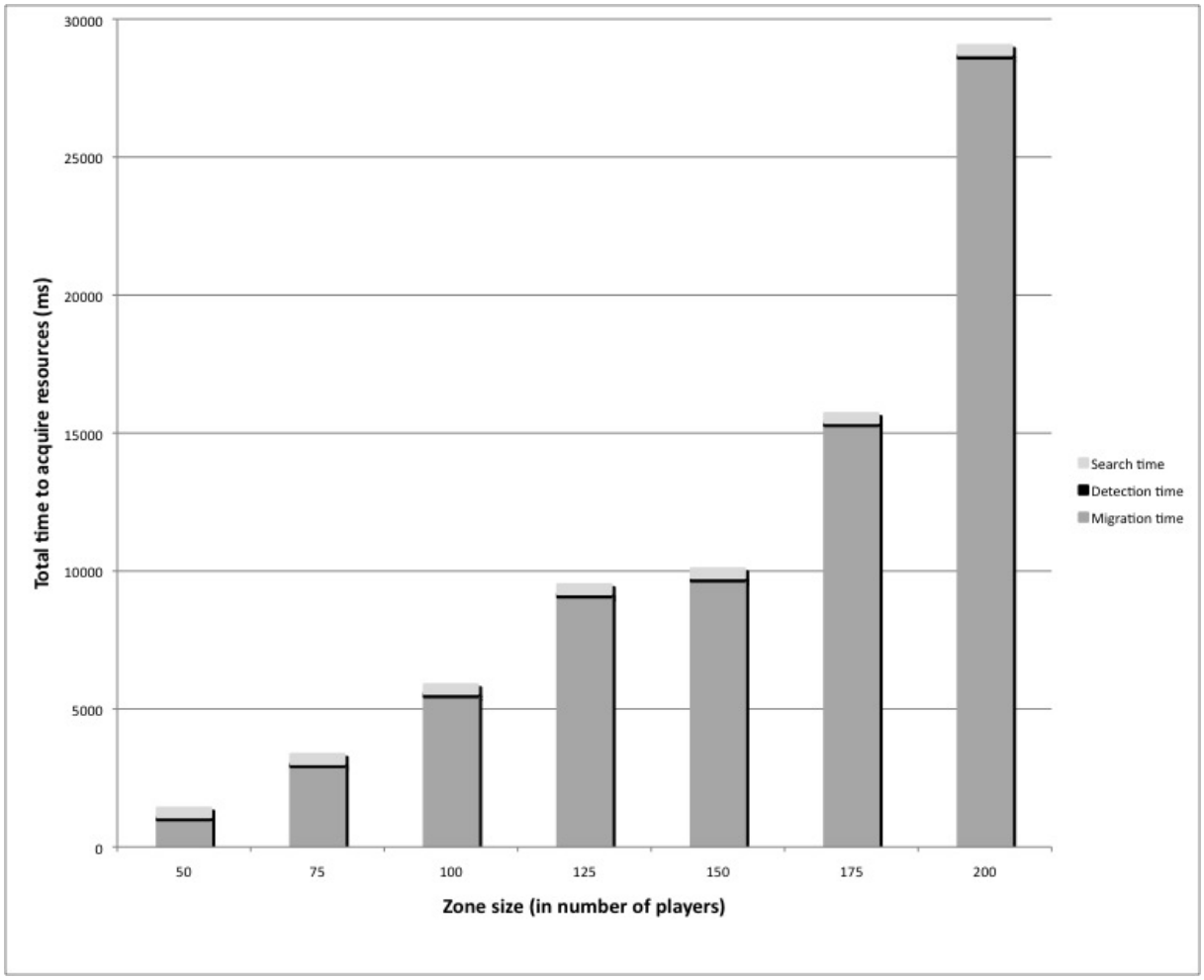


Figure 6.11: Time taken to execute resource coordination mechanisms.

zone. Therefore, it may be safe to assume that the time taken for a detection is not dependent on the number of players in the zone. We observe similar characteristics for search operation. However, in this experiment, the directory manager is located in the same network as the broker and game server. In a real world application, where there can be thousands of available resources to search from and with much greater geographical distances between brokers, resources and directory managers, these values can rise quite significantly. On the other hand, time taken to migrate a number of players along with other entities from one server to another increases non-linearly with the increase of players. When a zone migrates, the CyberOrgs-MMOG API allows migration of entities one by one. Therefore, it is quite evident that the more the number of players and entities the more time it takes. This has been done intentionally to reduce the time lag that occurs during migration. Besides, the bigger the size of the game zone the more data needs to be synchronized between the servers. While the other two operations do not show any correlation with the number of players/entities, for migration, the time increases non-linearly with the increase of players. One observation from the Table 6.1 is zones with sizes 125 and 150 have a smaller difference in migration times than other zone size differences. There could be several reasons behind this. One reason is that when moving zones from one host to another the number of surrounding entities could differ significantly as entities such as the enemies can move in or out of the zone during a migration. However, the actual migration time does not have any effect on the response time and thus, does not affect the game performance. Though this may affect the bandwidth requirement as during these migration periods servers have to be kept synchronized. The analysis provided in this subsection can be used to decide on the minimum size of the zones based on the availability of bandwidth.

6.5 Chapter Summary

In this chapter, experimental results in terms of quality of services are presented. The effects on the game performance by the mechanisms provided by CyberOrgs-

# of players	Detection		Search		Migration		Total
	mean	stddev	mean	stddev	mean	stddev	
50	110	11.2	320	23.2	981	103.4	1411
75	105	13.7	344	14.8	2911	167.7	3360
100	123	6.6	325	22.5	5438	234.5	5886
125	118	9.2	336	19.6	9055	255.6	9509
150	125	12.7	328	15.5	9633	271.9	10086
175	112	9.7	334	16.7	15264	563.4	15710
200	125	5.5	338	20.2	28586	989.3	29049

Table 6.1: Time taken to execute resource coordination mechanisms.

MMOG API are also evaluated. We notice improvement in response time per player as well as average outgoing bandwidth per server when resource is dynamically added or released based on a cutoff value. This approach provides much more flexibility and control over the game performance. It is also observed that supporting these mechanisms do not cause a significant overhead compared to the performance of the gameplay without the API. We also notice that the number of players do not affect load detection and resource search operations though it has a non-linear correlation with the migration operation.

Chapter 7

Conclusion

Online gaming has a great potential for both education and entertainment. The number of players as well as multiplayer games are increasing. In some games, the growth of population is observed to be exponential [15]. Supporting such a large number of simultaneous players require a highly scalable system. Moreover, as the number of players is limited for each server, appropriate resource coordination is required for easy access to the resources. This thesis investigates how multiplayer games can support a large number of players by acquiring resources on the fly. Moreover, we believe ownership of hardware may not be always a desirable option as ownership of hardware requires maintaining and upgrading frequently. Besides, a massively multiplayer online game may require servers all around the world. Appropriate resource coordination allows easy access to such resources as processor cycles and network resources without owning the hardware. This thesis also addresses the issue of random hotspot creation and how dynamic resource acquisition can resolve this issue. It is observed that most multiplayer games that are massively played are specific to some genres. We show that coordinated resource use can meet the performance goals required by any multiplayer online game of any genre to be a massively played one.

This thesis presents the CyberOrgs-MMOG API, which supports encapsulation of computations as well as mechanisms to coordinate resources among these computations dynamically based on predefined criteria. The API incorporates three mechanisms to achieve this - detection of a load condition, search for appropriate resources if necessary and migrate a part of the computation to the available resource. Methodology proposed by previous studies (discussed in Section 2.4) are adopted

to allow game players having seamless experience of gaming during migration.

This thesis also evaluates these mechanisms in Section 6.3. The effects on response time and bandwidth caused by different API calls are investigated in Subsection 6.4.1. The time required by each of the mechanisms are evaluated in Subsection 6.4.2. The results show that the resource acquisition and release in real time can be achieved without significant overhead or degradation in performance of the game. Section 7.1 presents the limitations of the CyberOrgs-MMOG API along with the limitations of the analysis made on the API. In Section 7.2, some future directions are provided.

7.1 Limitations

The limitations of the CyberOrgs-MMOG API and the experiment are listed below:

- The API is demonstrated using a game with a random interactivity level of 0 to 10ms, whereas massively multiplayer online games have varying interaction levels depending on the genre. It would be interesting to observe the game performance provided by different genres with different levels of interactivity.

- The experimental results as discussed in Chapter 6 are based on a simulation. The simulator generates the players for controlled experimentation and simulates the actions performed by a game player. Instead of simulation, which relies on uniform random distribution, a study involving real player actions that may not be uniform in nature would strengthen the results.

7.2 Future Directions

The CyberOrgs-MMOG API can control the quality of the services for all players in a particular server. However, this can be extended to a more fine-grained approach that allows a player to control its game experience by controlling the consumption of resource. For example, a game player may choose to play a variation of the same

game where the experience is enhanced by better graphics or better response time. These services may consume more resources and the player may agree to pay more for these services. On the other hand, another game player may want to stick with the lower quality version of the game.

Another possible future work can be incorporating network resource control along with the processor cycles. I believe controlling network resource is also important as the performance of a multiplayer game, specially a massively played one, is dependent on the performance and efficiency of network resources. To support such control an extension of the CyberOrgs-MMOG API can be developed.

References

[1] JMaPacman. `http://arianne.sourceforge.net/game/jmapacman.html` [Last accessed: 2011-04-09].

[2] Marauroa Game Engine. `http://arianne.sourceforge.net/engine/marauroa.html` [Last accessed: 2011-04-09].

[3] MMOData Charts. `http://mmodata.net/` [Last accessed: 2011-12-12].

[4] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos. Behavior and Performance of Interactive Multi-Player Game Servers. *Cluster Computing*, 6:355–366, October 2003.

[5] G. Agha, S. Frolund, W.Y. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *Parallel Distributed Technology: Systems and Applications, IEEE*, 1:3–14, May 1993.

[6] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Massachusetts, USA, 1986.

[7] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7:1–72, January 1997.

[8] Marios Assiotis and Velin Tzanov. A Distributed Architecture for MMORPG. In *Proceedings of the 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, New York, New York, USA, 2006. ACM.

[9] Mark Astley. The Actor Foundry. *Manual of the Actor Foundry, version 0.2.0, 1999.*

[10] Wentong Cai, Percival Xavier, Stephen J. Turner, and Bu-Sung Lee. A Scalable Architecture for Supporting Interactive Games on the Internet. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, PADS '02, pages 60–67, Washington, D.C., USA, 2002. IEEE Computer Society.

[11] Edward Castronova. Virtual Worlds: A First-Hand Account of Market and Society on the Cyberian Frontier. *The Gruter Institute Working Papers on Law, Economics, and Evolutionary Biology*, 2(1), 2001.

[12] Jin Chen, Baohua Wu, Margaret Delap, Björn Knutsson, Honghui Lu, and Cristiana Amza. Locality Aware Dynamic Load Management for Massively Multiplayer Games. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 289–300, New York, New York, USA, 2005. ACM.

[13] William D Clinger. Foundations of Actor Semantics. Technical report, Cambridge, Massachusetts, USA, 1981.

[14] Daniel Cordeiro, Alfredo Goldman, and Dilma da Silva. Load balancing on an interactive multiplayer game server. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 184–194. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74466-5_21.

[15] Deloitte. The Collaboration Curve: Exponential Performance Improvement in World of Warcraft. `http://www.deloitte.com/assets/Dcom-UnitedStates/Local%20Assets/Documents/us_tmt_WoW_082009.pdf` [Last accessed: 2011-12-12].

[16] Frank Glinka, Alexander Ploß, Jens Müller-lden, and Sergei Gorlatch. RTF: a Real-Time Framework for Developing Scalable Multiplayer Online Games. In *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '07, pages 81–86, New York, New York, USA, 2007. ACM.

[17] I. Grief and Irene Greif. Semantics of Communicating Parallel Processes. Technical report, Cambridge, Massachusetts, USA, 1975.

[18] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[19] Carl Hewitt. PLANNER: A Language for Proving Theorems in Robots. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 295–301, San Francisco, California, USA, 1969. Morgan Kaufmann Publishers Inc.

[20] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, San Francisco, California, USA, 1973. Morgan Kaufmann Publishers Inc.

[21] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. Zoned Federation of Game Servers: a Peer-to-Peer Approach to Scalable Multi-Player Online Games. In *Proceedings of the 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '04, pages 116–120, New York, New York, USA, 2004. ACM.

[22] Nadeem Jamali. *CyberOrgs: A Model for Resource Bounded Complex Agents*. PhD thesis, Champaign, Illinois, USA, 2004. AAI3153330.

[23] Nadeem Jamali and Xinghui Zhao. Hierarchical Resource Usage Coordination for Large-Scale Multi-agent Systems. In Toru Ishida, Les Gasser, and Hideyuki Nakashima, editors, *Massively Multi-Agent Systems I*, volume 3446 of *Lecture Notes in Computer Science*, pages 40–54. Springer Berlin / Heidelberg, 2005. 10.1007/11512073_4.

[24] Ihab Kazem, Dewan Tanvir Ahmed, and Shervin Shirmohammadi. A Visibility-Driven Approach to Managing Interest in Distributed Simulations with Dynamic Load Balancing. In *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '07, pages 31–38, Washington, D.C., USA, 2007. IEEE Computer Society.

[25] B. Knutsson, Honghui Lu, Wei Xu, and B. Hopkins. Peer-to-Peer Support for Massively Multiplayer Games. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 4 vol. (xxxv+2866), March 2004.

[26] Open System Laboratory. The Actor Foundry. `http://osl.cs.uiuc.edu/af/` [Last accessed: 2010-06-15].

[27] Fengyun Lu, Simon Parkin, and Graham Morgan. Load Balancing for Massively Multiplayer Online Games. In *Proceedings of the 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, New York, New York, USA, 2006. ACM.

[28] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Paul T. Barham, and Steven Zeswitz. NPSNET: A Network Software Architecture for Large Scale Virtual Environments. *Presence*, 3:265–287, 1994.

[29] Martin Mauve, Stefan Fischer, and Jörg Widmer. A Generic Proxy System for Networked Computer Games. In *Proceedings of the 1st Workshop on Network and System Support for Games*, NetGames '02, pages 25–28, New York, New York, USA, 2002. ACM.

[30] Jens Müller and Sergei Gorlatch. Rokkatan: Scaling an RTS Game Design to the Massively Multiplayer Realm. *Computers in Entertainment*, 4(3), July 2006.

[31] Myeong-Wuk Jang and Amr Ahmed and Gul Agha. Efficient Communication in Multi-Agent Systems. In *Software Engineering for Scale Multi-Agent Systems III, Lecture Notes in Computer Science 3390, Springer-Verlag, pp. 236-253*, 2005.

[32] V. Nae, A. Iosup, and R. Prodan. Dynamic Resource Provisioning in Massively Multiplayer Online Games. *IEEE Transactions on Parallel and Distributed Systems*, 22(3):380 –395, March 2011.

[33] Bonnie Nardi and Justin Harris. Strangers and Friends: Collaborative Play in World of Warcraft. In *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 149–158, New York, New York, USA, 2006. ACM.

[34] Open System Laboratory. The Actor Architecture. `http://osl.cs.uiuc.edu/research.php?action=topic&topic=Actor+Systems` [Last accessed: 2010-06-15].

[35] Radu Prodan and Vlad Nae. Prediction-based Real-Time Resource Provisioning for Massively Multiplayer Online Games. *Future Generation Computer Systems*, 25(7):785 – 793, 2009.

[36] Leon Ryan. Beyond the Looking Glass of MMOG's. *GameAxis Unwired*, pages 22–31, 2007.

[37] Sandeep K. Singhal. Scalable Networked Virtual Environments Using Unstructured Overlays. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems - Volume 02*, ICPADS '07, pages 1–8, Washington, D.C., USA, 2007. IEEE Computer Society.

[38] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Notices*, 36(12):20–34, December 2001.

[39] N. Yee (2007). Motivations of Play in Online Games. *Journal of CyberPsychology and Behavior*, 9:772–775, 2007.

[40] Yue Zhang and Nadeem Jamali. Negotiating Multimedia Advertising with Attention Owners. In *Proceedings of the International Conference on Multimedia*, MM '10, pages 755–758, New York, New York, USA, 2010. ACM.

[41] Xinghui Zhao. CyberOrgs 1.0. `http://agents.usask.ca/Agents_Lab/Agents_Lab_-_CyberOrgs.html` [Last accessed: 2011-03-29].

Appendix A

Sample Dataset

In this appendix, dataset used to draw Figure 6.1, 6.8, and 6.10 is presented. Table A.1-A.4 show the data used in Figure 6.1, Table A.5-A.8 show the data used in Figure 6.8, and Table A.9 is used in Figure 6.10.

Table A.1: Resource added in response to increase in number of players

# Resources	# Players	Response time	# Resources	# Players	Response time
1	1	25.27	1	26	19.28
1	2	25.22	1	27	19.02
1	3	26.05	1	28	19.93
1	4	25.60	1	29	21.58
1	5	25.33	1	30	22.22
1	6	26.19	1	31	22.91
1	7	26.13	1	32	22.94
1	8	25.92	1	33	23.55
1	9	25.37	1	34	25.23
1	10	24.80	1	35	27.39
1	11	24.78	1	36	29.21
1	12	24.18	1	37	30.43
1	13	23.83	1	38	31.28
1	14	23.60	1	39	32.28
1	15	22.21	1	40	34.26
1	16	20.72	1	41	37.60
1	17	20.50	1	42	41.84
1	18	20.36	1	43	46.12
1	19	20.13	1	44	48.60
1	20	19.90	1	45	47.92
1	21	19.21	1	46	48.33
1	22	19.08	1	47	51.88
1	23	18.83	1	48	56.02
1	24	18.47	1	49	57.46
1	25	18.87	1	50	57.45

Table A.2: Resource added in response to increase in number of players

# Resources	# Players	Response time	# Resources	# Players	Response time
2	51	57.12	2	76	31.73
2	52	45.25	2	77	33.30
2	53	35.46	2	78	34.27
2	54	25.31	2	79	34.36
2	55	25.46	2	80	33.56
2	56	24.95	2	81	34.17
2	57	23.52	2	82	34.76
2	58	22.38	2	83	34.47
2	59	22.11	2	84	34.54
2	60	22.35	2	85	34.56
2	61	22.42	2	86	33.92
2	62	21.87	2	87	32.80
2	63	20.85	2	88	33.80
2	64	20.31	2	89	36.92
2	65	19.84	2	90	38.49
2	66	19.68	2	91	37.86
2	67	20.89	2	92	36.93
2	68	22.28	2	93	38.33
2	69	22.89	2	94	41.92
2	70	24.46	2	95	44.91
2	71	25.71	2	96	47.25
2	72	26.57	2	97	50.13
2	73	27.95	2	98	51.86
2	74	28.75	2	99	52.90
2	75	29.86	2	100	55.44

Table A.3: Resource added in response to increase in number of players

# Resources	# Players	Response time	# Resources	# Players	Response time
3	101	59.38	3	126	23.86
3	102	64.42	3	127	23.65
3	103	67.48	3	128	22.14
3	104	69.08	3	129	21.96
3	105	70.98	3	130	22.25
3	106	31.48	3	131	22.49
3	107	31.41	3	132	25.12
3	108	32.86	3	133	25.86
3	109	34.77	3	134	26.54
3	110	36.02	3	135	27.44
3	111	35.09	3	136	29.50
3	112	32.52	3	137	31.81
3	113	30.81	3	138	33.25
3	114	29.53	3	139	34.48
3	115	28.50	3	140	34.95
3	116	27.25	3	141	34.42
3	117	26.64	3	142	32.63
3	118	25.70	3	143	32.09
3	119	24.37	3	144	33.43
3	120	23.17	3	145	33.88
3	121	22.25	3	146	32.99
3	122	22.68	3	147	32.94
3	123	23.77	3	148	33.20
3	124	24.35	3	149	33.17
3	125	24.08	3	150	36.34

Table A.4: Resource added in response to increase in number of players

# Resources	# Players	Response time	# Resources	# Players	Response time
4	151	41.73	4	176	22.12
4	152	46.27	4	177	23.71
4	153	50.87	4	178	22.14
4	154	54.04	4	179	21.56
4	155	55.32	4	180	23.44
4	156	56.48	4	181	23.81
4	157	49.70	4	182	23.84
4	158	38.67	4	183	22.89
4	159	27.45	4	184	23.55
4	160	26.77	4	185	24.10
4	161	25.94	4	186	24.25
4	162	26.08	4	187	24.67
4	163	27.64	4	188	24.89
4	164	29.29	4	189	24.30
4	165	29.91	4	190	24.70
4	166	29.16	4	191	25.74
4	167	28.36	4	192	26.45
4	168	28.53	4	193	27.88
4	169	28.38	4	194	30.52
4	170	27.58	4	195	31.22
4	171	27.00	4	196	30.09
4	172	25.91	4	197	29.92
4	173	24.00	4	198	30.83
4	174	21.31	4	199	33.66
4	175	22.94	4	200	35.53

Table A.5: Resource released in response to decrease in average outgoing bandwidth

# Resources	# Players	Response time	# Resources	# Players	Response time
4	200	90.78	4	175	49.99
4	199	85.60	4	174	46.64
4	198	85.70	4	173	44.41
4	197	88.19	4	172	43.54
4	196	84.03	4	171	44.49
4	195	76.41	4	170	44.49
4	194	69.26	4	169	43.37
4	193	65.23	4	168	41.75
4	192	65.85	4	167	40.90
4	191	66.22	4	166	40.45
4	190	66.99	4	165	40.85
4	189	68.06	4	164	39.17
4	188	65.05	4	163	37.19
4	187	63.49	4	162	35.30
4	186	63.92	4	161	32.32
4	185	65.18	4	160	34.94
4	184	66.59	4	159	34.81
4	183	65.78	4	158	34.23
4	182	66.30	4	157	35.13
4	181	68.88	4	156	34.50
4	180	68.99	4	155	36.93
4	179	68.22	4	154	35.47
4	178	65.24	4	153	35.05
4	177	58.49	4	152	35.57
4	176	51.66	4	151	34.98

Table A.6: Resource released in response to decrease in average outgoing bandwidth

# Resources	# Players	Response time	# Resources	# Players	Response time
4	150	34.27	3	125	50.83
4	149	33.51	3	124	48.47
4	148	32.74	3	123	44.96
4	147	31.95	3	122	40.90
4	146	32.53	3	121	40.13
3	145	28.46	3	120	41.03
3	144	71.88	3	119	37.05
3	143	80.04	3	118	32.78
3	142	82.21	3	117	32.59
3	141	75.75	3	116	31.27
3	140	73.26	3	115	31.57
3	139	69.99	2	114	29.12
3	138	67.15	2	113	29.78
3	137	66.61	2	112	28.89
3	136	63.29	2	111	28.80
3	135	63.35	2	110	28.91
3	134	63.00	2	109	29.46
3	133	61.27	2	108	29.20
3	132	61.95	2	107	29.54
3	131	60.84	2	106	30.10
3	130	61.58	2	105	30.96
3	129	57.21	2	104	32.01
3	128	53.50	2	103	33.21
3	127	55.98	2	102	34.36
3	126	52.48	2	101	35.25

Table A.7: Resource released in response to decrease in average outgoing bandwidth

# Resources	# Players	Response time	# Resources	# Players	Response time
2	100	37.75	2	75	53.49
2	99	35.86	2	74	52.42
2	98	33.56	2	73	48.91
2	97	34.97	2	72	47.43
2	96	34.10	2	71	46.35
2	95	35.25	2	70	43.58
2	94	38.57	2	69	44.75
2	93	43.98	2	68	46.20
2	92	51.15	2	67	45.47
2	91	71.50	2	66	46.14
2	90	71.56	2	65	47.91
2	89	69.77	2	64	47.72
2	88	68.43	2	63	46.43
2	87	68.32	2	62	44.56
2	86	66.47	2	61	42.71
2	85	59.69	2	60	45.15
2	84	62.92	2	59	44.64
2	83	61.46	2	58	44.07
2	82	59.55	2	57	42.83
2	81	59.12	2	56	41.46
2	80	64.45	2	55	39.84
2	79	62.41	2	54	38.54
2	78	60.55	2	53	37.26
2	77	59.10	2	52	36.07
2	76	56.29	2	51	35.31

Table A.8: Resource released in response to decrease in average outgoing bandwidth

# Resources	# Players	Response time	# Resources	# Players	Response time
2	50	34.74	1	25	71.99
2	49	33.29	1	24	73.27
2	48	34.04	1	23	66.39
2	47	33.70	1	22	68.05
2	46	31.67	1	21	65.35
2	45	33.42	1	20	62.72
2	44	33.07	1	19	56.06
2	43	31.99	1	18	59.48
2	42	30.38	1	17	53.55
1	41	28.62	1	16	51.89
1	40	33.80	1	15	48.57
1	39	38.19	1	14	44.10
1	38	42.68	1	13	43.30
1	37	47.27	1	12	39.75
1	36	76.77	1	11	41.15
1	35	74.12	1	10	34.94
1	34	71.99	1	9	36.78
1	33	69.62	1	8	38.27
1	32	68.49	1	7	35.28
1	31	68.31	1	6	31.99
1	30	68.02	1	5	29.96
1	29	67.88	1	4	27.19
1	28	68.44	1	3	26.78
1	27	70.11	1	2	29.21
1	26	74.75	1	1	28.41

Table A.9: Comparison of average outgoing bandwidth with or without the API

# player	w/o API	with API	# Players	w/o API	with API
1	20.39	22.53	26	44.10	44.49
2	21.64	21.95	27	48.57	43.54
3	23.92	23.74	28	46.20	44.41
4	24.86	23.51	29	47.83	46.64
5	25.23	24.27	30	48.38	49.99
6	26.18	25.98	31	50.47	51.66
7	28.41	25.57	32	49.03	51.49
8	29.21	26.05	33	51.89	55.24
9	26.78	26.47	34	54.65	53.22
10	27.19	28.93	35	53.88	54.99
11	27.08	27.50	36	52.66	57.88
12	29.96	28.13	37	54.08	58.30
13	31.99	29.89	38	57.99	62.78
14	33.27	31.81	39	56.06	61.59
15	33.08	30.94	40	62.72	65.18
16	34.94	32.32	41	61.35	63.92
17	38.15	35.30	42	63.05	63.49
18	39.75	37.19	43	62.39	65.05
19	37.19	39.17	44	65.44	68.06
20	39.17	40.85	45	65.88	66.99
21	40.85	40.45	46	68.02	68.22
22	40.45	40.90	47	68.31	71.85
23	40.90	41.75	48	68.49	73.23
24	41.75	43.37	49	69.62	78.19
25	43.30	44.49	50	72.75	76.15