Subgames in Massively Multiplayer Online Games

Michael A. Hawker

Master of Science

School of Computer Science

McGill University

Montréal, Québec

June 2008

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science

DEDICATION

This thesis is dedicated to my parents who have encouraged and supported me from the beginning.

ACKNOWLEDGEMENTS

I would like to thank my supervisors Jörg Kienzle and Clark Verbrugge for their support, advice, wisdom, and time. Without them this work would not have been possible. The opportunity to work on the Mammoth project has been an invaluable experience these past two years.

I would also like to thank all the students (Jean-Sébastien Boulanger, Loc Bui, Jeremy Claude, Alexandre Denault, Adrian Ghizaru, Nadeem Khan, Marc Lanctot, Alfred Leung, Pierre Marieu, Nicolas NgManSun, Alexandre Quesnel, Russell Spence) who have worked on the Mammoth framework which made the implementation of the work in this thesis possible.

Although a team effort, a special thanks goes to Marc Lanctot for the initiative and organization of the Orbius gaming event which spawned the notion of subgames in Mammoth.

I would also like to thank Alexandre Denault for his support as a mentor and advisor, his invaluable advice, and his work on replicated objects which made the work in this thesis possible.

Finally, I would like to thank the rest of my family and friends for all the experiences we have shared which brought me to where I am today. And my special sentiments to Jennifer Lowry for her continual understanding and support.

ABSTRACT

With the launch of World of Warcraft in 2004, Massively Multiplayer Online Games (MMOGs) really came into their own as millions of people started playing worldwide. Providing scalability to such a large audience while maintaining a consistent gameplay experience is a difficult task which many companies face in an industry where only few succeed.

This thesis focuses on the issues of how a MMOG can be scaled to support more concurrent players and how consistency can be maintained in a Distributed Multi-Server Environment (DMSE). As a basis for investigation the notion of "Subgames" (i.e. games within games) was introduced. As smaller, more flexible game units, subgames reduce scalability problems but raise consistency concerns by requiring modular game actions in a distributed environment to function. This is addressed through a new transactional protocol and action framework which abstracts and solves consistency issues while creating an infrastructure which allows for scalability. A complete solution is illustrated using these techniques through the design of general game mechanics and subgames.

The approach here further enables scalability of MMOGs in a DMSE and provides a general framework for the further investigation of MMOG consistency and scalability through subgame instances.

ABRÉGÉ

La popularité des jeux massivement multi-joueurs en ligne (MMOGs) a grandement augmenté avec l'arrivée du jeu World of Warcraft, qui est joué par des millions de personnes à travers le monde. Cependant, ce type d'application nécessite des infrastructures extensibles pour accommoder des milliers de joueurs, tout en offrant une expérience de jeu consistante. Ceci représente un grand obstacle que plusieurs compagnies doivent affronter, mais qui est surmonté par peu.

Cette thèse aborde les problèmes reliés à la croissance du nombre de joueurs simultanés, tout en discutant comment maintenir un environnement distribué multi-serveurs (DMSE) consistant. La notion de sous-jeux (un jeu qui se déroule l'intérieur d'un autre jeu) a été utilisée pour mieux étudier le problème. En tant qu'unités de jeu plus petits et flexibles, les sous-jeux facilitent la croissance, mais augmentent les problèmes de concurrence puisque leur bon fonctionnement nécessite des actions modulaires dans un environnement distribué. Ces défis sont adressés par un nouveau protocole transactionnel et un cadre d'applications d'actions qui font abstraction et règlent les problèmes de consistance, tout en offrant une infrastructure qui permet une certaine croissance. Une solution, où les mécanismes de jeux et de sous-jeux sont adaptés en conséquence, illustre les techniques proposées dans cette thèse.

Ces techniques permettent une plus grande croissance pour les jeux MMOGs dans un DMSE, tout en fournissant des outils de sous-jeux qui permettent l'étude des défis de consistance et de croissance.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
Introduction

Massively Multiplayer Online (Role-Playing) Games (MMOGs or MMORPGs) have redefined traditional computer game notions and defined an era. They have almost become common knowledge since the advent of the most popular MMORPG: World of Warcraft [8]. Unlike traditional video games focused on the aspects of a single player or a small group of players, MMOGs throw their players into a virtual world to interact with real people (via a personalised avatar) in a dynamic environment. These additional features add complex networking and social challenges to the development of the game world and the game itself.

MMOGs reach millions of people worldwide [8, 6, 3] generating massive amounts of income for their developer's corporations. It is imperative as a new class of service provider that these games remain operational. However, creating a system that is able to scale to such a large audience with hundreds of thousands of concurrent users, while maintaining a consistent gameplay experience, is a difficult task. The quality of the game experience for a player, however, can make all the difference between the success or failure of the game.

Motivation

There are more strict requirements on the network infrastructure in a typical MMOG. Simple network designs do not "scale" to the hundreds of thousands or millions of players supported by a popular MMOG. In order to allow many players

to play the game "simultaneously," players are usually spread across many different copies of the same game world, on multiple different individual servers [26]. Unfortunately, while this allows more players to play the same game, many players will be isolated from each other. This practice, while practical from a developer perspective, usually upsets players wishing to interact with each other [26].

The motivation behind this thesis is to break down a MMOG into smaller more manageable games called "Subgames" to allow for increased and more flexible interaction between players while still being able to scale and increase the total possible number of players within the same "global world." To this end, the design presented here is based on a Distributed Multi-Server Environment (DMSE), allowing for greater scalability compared to a traditional client/server network architecture [10].

Good game design requires strict consistency of the game state (even in a distributed game environment), and for subgame purposes, the ability to modify and enhance the default game mechanics. These issues are also discussed and addressed with the notion of a light-weight transactional protocol suitable for games within a scalable and modular framework. The complete design is backed by a non-trivial implementation appropriate for scalable subgame architectures which can increase interactions between players in a consistent fashion.

Contributions

The specific contributions presented by the research in this thesis are the following:

1. Creating a modular "subgame" infrastructure to expand upon a global game world in a scalable fashion.
2. Providing a practical design for encapsulating game state changes into entities called *actions*. This helps maintain consistency and provides a way to override default game behaviors.
3. The creation of an efficient transactional protocol to maintain consistency for games in a distributed multi-server environment.
4. A versioning scheme used in conjunction with consistency resolution to prevent malformed state-change requests and ensure distributed operations are properly ordered.
5. A demonstrated implementation of the complete architecture in a non-trivial, distributed game environment.

Roadmap

The next chapter provides a background to the main principles discussed in the thesis research, Chapter 3 provides further background on the Mammoth project (the game research framework used for implementation and experimentation), Chapter 4 discusses the main contributions of this research in detail with respect to accommodating both subgame and consistency requirements, while Chapter 5 focuses more on surrounding challenges encountered in the implementation itself. Finally, Chapter 6 concludes and discusses future work enabled by this thesis. An Appendix is also provided including other miscellaneous implementation work done on the Mammoth project during the duration of this thesis.

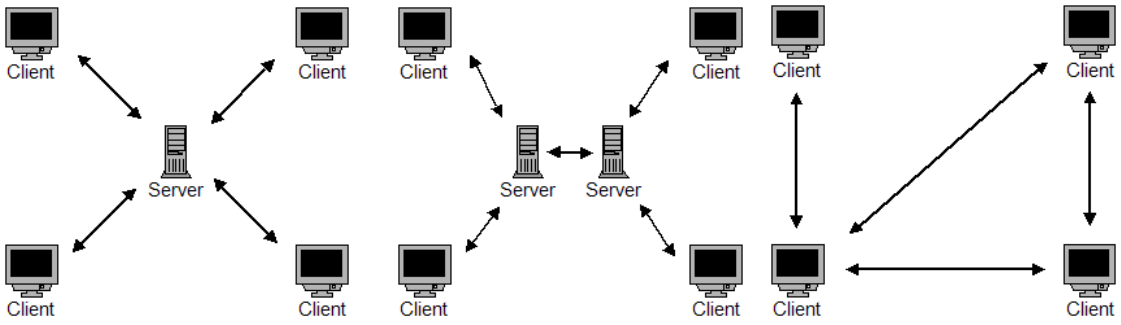
Chapter 2
Related Work: Massively Multiplayer Online Games

While computer games have been around since the 1970's, MMOGs are a relatively new genre of game. Originally, online games called Multi-User Dungeons (or MUDs) existed giving people the text-based equivalent of what a MMOG is considered today. The main proponent of change for the advent of the MMOG was the increased network infrastructure provided by the Internet and the availability of dial-up and high speed Internet access in the late 1990's.

With the launch of modern MMORPGs such as EVE Online, World of Warcraft (WoW), and Guild Wars from 2003-2005, MMOGs really came into their own as millions of people started playing worldwide [6, 8, 3]. To achieve this a number of critical systems are required to support the game infrastructure needed to handle a large numbers of players at once. Two main issues which are specifically relevant to MMOGs are *Scalability* and *Consistency*. Below a discussion of useful network designs for MMOGs is presented, followed by scalability and consistency concerns.

2.1 Network Topography

There are many different approaches to create a network topology. Several basic models are shown in Figure 2–1. These designs range from a traditional client/server to a completely peer-to-peer model [14].



(a) Client/Server **(b) DMSE** **(c) Peer-to-peer**

Figure 2–1: Network Architectures

Client / Server

The Client/Server model shown in Figure 2–1a represents a system with a centralised server maintaining all the information about players and the game world. Clients connect to the server and receive information about the game world around them. All game state queries and modifications go to the server to be executed. This provides the greatest control and easiest approach to consistency for the game world, as there is only a single point of access. Unfortunately, this approach is not scalable because the number of supportable players is dependent on the physical hardware and network limitations of a single machine. For its simplicity and easy management of consistency, however, this system is used in the majority of MMOGs [14]. In order to scale their games, most commercial systems duplicate this architecture and balance the load of clients amongst different servers, but this tends to isolate players wishing to play with one another [26].

Peer-to-Peer

The Peer-to-Peer (P2P) model shown in Figure 2–1c is the opposite extreme of the client/server model. With the P2P model, the system is "infinitely" scalable as the network load is completely distributed amongst all of the computers in the system. However, in this model there are several practical problems when applying it for games, principal among them are availability, performance, consistency and security (to prevent game cheating) [20]. Most companies in the industry tend to avoid this model due to these issues, especially with respect to the security concerns. Recent exceptions exist though: WoW is using the benefits of a P2P system to disseminate its client software updates [2].

Distributed Multi-Server Environment

The Distributed Multi-Server Environment (DMSE) model in Figure 2–1b is used by the Mammoth project. It combines the security and update performance of the client/server model with the potential scalability of a P2P network. Like a P2P network the system can simply scale by adding another machine to the cluster; however, unlike P2P, the onus and cost of adding extra machines is still with the service provider, rather than utilizing the power of client machines. As previously mentioned, most MMOGs in the industry are currently configured in a centralised manner due to the security benefits it provides [14].

Unfortunately, as in the P2P model there are a couple of issues with the DMSE model which need to be addressed: there needs to be an easy way to scale the game world and there needs to be a way to maintain the consistency of the game state. Solving these two problems is the goal of this thesis.

Duplication Spaces

While the DMSE model has similar benefits to the traditional client/server architecture, programming a MMOG in a distributed setting is much more complex. To combat this issue, technology from Quazal was used which abstracts the notions of where objects reside on the network using "Duplication Spaces," based on an object replication scheme with "Master" objects and copied "Replicas" [28]. More about how this technology is used and adapted specifically to Mammoth is discussed in Section 3.4.

2.2 Scalability

The choice of a network topology, as described in the previous section, can effect the ability to scale an MMOG based on physical limitations of hardware. However, the network topology does not address how it is possible to scale the overall game architecture to the same levels.

Scalability quickly becomes an issue in MMOGs if any of the basic game architecture algorithms become a performance bottleneck as more users are added or the size of the world is increased. Some of these algorithms may be polynomial or exponential in nature, or the entire set of world objects can simply be too large. In these cases, a game can break down quickly with even a relatively small increase to the number of users.

WoW provides an interesting comparison. Their user base is approximately 10 million subscribers worldwide, with half of them located in China [8]. With so many subscribers, there is also the possibility for many of them to be playing at the same time, and at its peak WoW has had upwards of *a million* concurrent users in

China [5] with an average of 330,000 in the first-quarter of 2007 [4]. With such a large number of concurrent users it is hard to see how a game with a single-server architecture would be able to handle such requirements. Fortunately, there are a number of techniques available to increase the abilities of the underlying network architecture.

2.2.1 Shards/Instances

To combat the issues with the number of concurrent users WoW takes a traditional stance of effectively duplicating the game and isolating players onto separate servers. In the U.S. alone, WoW uses over 200 servers, each supporting "thousands" of players [1]. This common technique is used often in the industry and is known as creating a "Shard" or "Instance" of the game world [26].

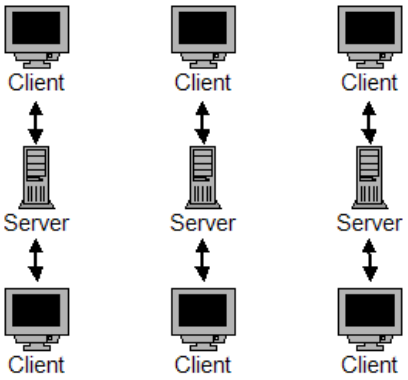


Figure 2–2: Server Shards

Figure 2–2 shows the typical game shard setup. Individual servers become responsible for a set of clients [13]. Each server has a complete copy of the game running on it, and no communication occurs between the servers. Players are isolated from each other if they are connected to different servers.

Shards have the same advantages as the traditional client/server system: the game is controlled from a central location, the game world can be kept consistent, and it can be harder to infiltrate or abuse the system. In addition, the scalability comes from being able to add another server, which contains another instance of the game world, which players can connect to. Unfortunately, since players on different servers are isolated from each other, players wishing to play together must ensure they are connected to the same shard. In addition, each server still has a physical limit to the number of players able to play on it. Therefore it is possible for individual servers to still become overloaded with players.

The other disadvantage to shards is they limit the "massively multiplayer" experience. While playing with thousands of players is still impressive, in the large sparse worlds typical of MMOGs a player may not encounter many other people or will continually see the same people. It is also deceptive to players as they may not realise they are isolated, and not in fact interacting with every possible player actually playing the game.

2.2.2 Interest Management

Interest management is another technique used to scale the ability of a single server. It is used to limit the amount of work a server has to do for each client and therefore allow it to handle more clients at once. There are various interest management schemes, but simplistically, each one can be thought of as a publish and subscribe model [11].

Publishers release new information about changes in the game world and are generally game objects. Subscribers are consumers of this information and are generally connected clients. Interest management refers to whatever scheme is used to determine the relation between these publishers and subscribers [11]. There is also middleware such as Quazal's Duplication Spaces which can do this to some degree automatically [28].

By lessening the number of messages needed to be sent to clients, interest management is beneficial to any architecture. By itself however, it does not solve underlying issues with scalability in a traditional client/server system. A detailed discussion of interest management and different approaches to it is given by Boulanger et al. [11].

2.2.3 Load Sharing

Another technique used to scale MMOGs is *Load Sharing* [16]. It is used more in cases of distributed network topologies, but can also be used with shards. To give an example for shards, players could connect through a central hub (which monitors all other servers) before being routed to the game server with the least number of players connected to it. In this manner there is always an even balance of players spread across all the servers in the network.

Attempts have been made in the past to solve scalability by "mirroring" the game-state across servers similar to something between the shard and DMSE approaches [15]; however, while this can balance the number of connections, it fails to

address the growing number of game objects and absolute consistency. It is important to note, load sharing is just as important for the objects in the game world as it is for handling connections from clients to servers [29].

As mentioned previously, relatively small changes to the number of users or game objects can quickly break a functioning game. By utilizing interest management with game objects and servers, the state of the game can be evenly distributed. This allows the game world to scale across multiple servers in a controlled fashion as well as the network connections from clients.

The Mammoth project uses this implementation of a DMSE topology (Figure 2–1b), interest management, and load sharing (with the use of Duplication Spaces [28]) to provide a scalable game environment. The implementation work of this thesis enables game mechanics to run in this setting as well as keeping the game state consistent across clients and servers.

2.3 Consistency

Being able to scale a MMOG is a necessity, but it is equally important to maintain a consistent view of the game world. Everyone playing the game should have the same information about the global game world and all the items and players they are currently interested in. Ensuring requests and updates are applied at the same time everywhere is a difficult task. Because this coordination for consistency is difficult a traditional single server model is generally preferred for MMOGs [26].

Using a DMSE model for networking would be beneficial due to its innate scalability; however, consistency with distributed objects is a much more complex problem. In the single server model, server-side consistency can be obtained simply

because there is only a single node which contains game data, and therefore setting up controls to synchronize updates is easier. In a DMSE, there are multiple nodes each with different sets of data; in order to make updates, more network messages are required, and even more work is required to maintain consistency.

There are still other consistency concerns in a client/server setup existing between clients and the server itself (as clients usually have an outdated, inconsistent view of the world); however, many techniques have been developed to resolve these issues. Mammoth, for example, had used a coordinated state change network message (described further in Appendix A.1).

"Dead-reckoning" is a common technique used to allow clients to make use of particular information to estimate game state in order to resolve short-term consistency issues [18]. Dead-reckoning is popular in games because traditional perfect consistency schemes for distributed environments (from the database community) are not "efficient" enough for the real-time nature of modern computer games [22].

With Dead-reckoning a game client utilizes information it currently has about the game state and its internal physics engine to compute where objects are in the future within a given amount time [27]. In this manner the game can seem to be moving in realtime, while in reality network messages are lagging slightly behind. Problems arise however when the predicted state differs from the actual state received. In these cases, visible "jumps" will occur or extra work is required to merge the divergent states [24].

The main issue with dead-reckoning is that most of this research focuses on player movement or other activities that can be predicted based on physical models.

The process breaks down when more complex interactions involving objects which move unpredictably at any point in time are introduced [27]. It is impossible for an algorithm to predict the change in state, if a player randomly decides to pick up a tomato. Since objects and interaction are a major component for MMOGs, many "impossible" predictions can occur at any given time. In addition the state update is necessary to prevent an object from appearing in two places at once. To solve this problem stronger consistency models are needed. Two main approaches often discussed for resolving these consistency problems are *local lag* and *lockstep* synchronization.

"Local lag" is where an intentional delay is used to counteract the delay caused by sending network messages [23]. With local lag the displayed game state runs slightly behind the known game state, hopefully giving enough time for a state change message to reach its destination before it needs to actually be displayed. Unfortunately, this delay is hard to estimate, and if the specified delay is not sufficiently long a rollback is needed to restore and repair the game state [17]. This is an expensive process that may affect other clients as well – rollbacks on one client can invalidate data sent to other clients, causing a cascade effect dependent on the number of participants in the game [25]. The *Timewarp* algorithm is a well-known design that ensures strong consistency through rollbacks [17]. In the case of an MMOG, however, the number of game participants is quite large and real-time progress is critical, making state rollback and repair impractical.

Similar problems with scalability arise when using the *lockstep protocol* to synchronize game state changes. The lockstep protocol revolves around a "stop-and-wait" mechanic where each client announces and receives other updates before proceeding to the next game step [9]. In this manner clients can remain consistent and secure; however, its performance is effected by the number of clients and the slowest clients connected in the network [9].

When a DMSE is introduced to increase the scalability of the game system, local lag and lockstep break down. In a DMSE, it is possible for a message to get rerouted from one server to another in order to find the Master object. In this case an extra time cost is incurred. However, for local lag, this increases the chance of needing a rollback as the message is delayed [13]. The rollback also becomes increasingly more expensive as the system scales upwards [24]. In the case of the lockstep protocol, every client in the system would be slowed down to wait for the extra time in the rerouted request. In both cases, it will become increasingly unmanageable as the number of clients and servers expand to the numbers found in typical MMOGs.

All of the schemes mentioned here, while providing consistency for their intended realms, fail to provide scalable implementations suitable for a DMSE and large-scale MMOGs. The work introduced by this thesis in Chapter 4 shows it is possible to maintain consistency at the point of an initial request across multiple servers. Thus, it becomes possible to scale a system while maintaining consistency in an arbitrary game environment.

Chapter 3
Mammoth Architecture

The following chapter describes the "Mammoth" Massively Multiplayer Game Research Framework and its current implementation. It is provided as an introduction into the Mammoth project and as further background to general gameplay elements and requirements usually found in MMOGs.

After introducing the Mammoth platform, basic concepts provided in its implementation needed for the research performed in this thesis will be described. Afterwards, a discussion of potential conflicts in consistency follows.

3.1 What is Mammoth?

Mammoth is a Massively Multiplayer Game Research Framework [7]. It was created as a collaborative project between a group of McGill University students and professors in the summer of 2005. Its goal was to provide an implementation platform for academic concepts from the fields of databases, distributed systems, fault tolerance, graphics, modeling and simulation, networking, artificial intelligence, and aspect-orientation to have a practical application to test different methodologies in a controllable environment.

It has since grown to a code base of over 80,000 lines of Java in about 800 class files. There is always a steady stream of development from numerous undergraduate and graduate contributors. Jörg Kienzle and Clark Verbrugge are the two main overseers of the project, and I joined the project at the start of 2006. The project

has been demonstrated at various conferences in Canada and has gained support and collaboration from gaming industry companies such as Electronic Arts and Quazal.

Like most massively multiplayer games, Mammoth consists of a virtual world where players take control of a digital avatar, moving around and interacting with objects. However, unlike other games, Mammoth has a fixed number of avatars in the world. These avatars will always exist even when players are not playing the game. A player logging into Mammoth then takes control of one of the avatars in the world during their gaming session.

3.2 Ground Work

Initially, the Mammoth virtual world was fairly bleak providing basic movement for players and very little in the way of interaction with objects contained within the game world. An item could be picked up, which would place it directly in the player's inventory, or it could be dropped directly at the player's feet from their inventory (see Figure 3–1).

While this mechanic was simple to implement and provide implementation for, it is not a realistic behavior nor one typically found in most games, especially in this day of age. Most research at the time for Mammoth revolved around a player's movement over the network, and as such, item interactions left much to be desired.

Before starting with the main implementation work of this thesis, I did some groundwork to increase the interactions available to players with items in the world. This increased interaction has been a major factor in making the research on sub-games, consistency and stable states described in this thesis possible, as well as

Figure 3–1: First Mammoth Client

enabling other research on path-finding [21], artificial intelligence [19], and interest management [12].

3.2.1 Item Interactions in Mammoth 1.0

Initially, items in Mammoth were fairly simplistic. Item properties included a name, weight, and value. As stated previously, items could only be picked up directly to the player's inventory or dropped at their feet (at the exact same location as the player).

When an item is dropped in such a manner, it is obscured by the player avatar. This is annoying to users, especially, if they wish to pick up the item again, as they would be required to move to do so. If multiple items are dropped, anyone wishing to pick one up later would have similar troubles as all the items would be overlapping. A user would be required to search through the pile of items one by one.

Searching through a pile on the ground would not be possible for a user, if their inventory was nearly full. Imagine a player wished to pick up a flower with weight 0.01kg. They have room in their inventory for one more item with a maximum weight of 0.1kg. If the flower is obscured by a Tomato of weight 0.2kg, it is impossible for a player to pick up the flower without first losing the security of an item already in their inventory. They would have to drop another item, pick up the tomato, move (as to not drop the tomato on their original item), drop the tomato, pick up the flower, and then pick up their original item.

This magnitude of manual labor expected of a player is unacceptable as it makes a game tedious instead of fun. Further work was pursued to ensure items would have more realistic interactions. It would also allow for a more modern up-to-date gameplay comparable with other games today. As most modern games allow a player to choose where they would like to place an item on the ground. Usually, this intermediate navigation is a visual distinction, but can also be realised as a separate game state. To update Mammoth, the notion of a "hand" was added.

3.2.2 Improving the User Interface

One of the major introductions to Mammoth was the notion of a player's "hand." A player would be able to pick up an object which would replace his mouse cursor

with an icon of the object he now held in his hand. This allowed for a, much needed, deeper interaction with items.

Not only could the player now choose if he wanted the item in his inventory, but they could also drop the item nearby or in an alternate location. This allows for additional gameplay scenarios; for instance, having an obstructing log which the player needs to move, before proceeding on their way.

Having a hand also prevents the manual labor of sorting items, as stated previously. With a hand any item can be picked up, regardless of if the player has space in their inventory for it or not. Only if a player tries to place the item in their inventory afterwards would a check be made for a full inventory. In this manner if there was a pile of objects on the ground, a player can simply continue to pick items up in their hand and move them aside until they find the one they are looking for.

Players would even be able to drop the item from their hand into a container in the world. Thus, the simple addition of a hand provided a system of state changes referred to as picking up, dropping, getting (an item from a container), and putting (an item in a container).

3.2.3 Containers

Containers were created not only as a better way to account for a player's inventory, but also to make the game world more dynamic. Now, a player had a choice. They could not only pick up an item and store it in their inventory or move it on the ground, but they could hide it in a different place, such as a trashcan (see Figure 3–2).

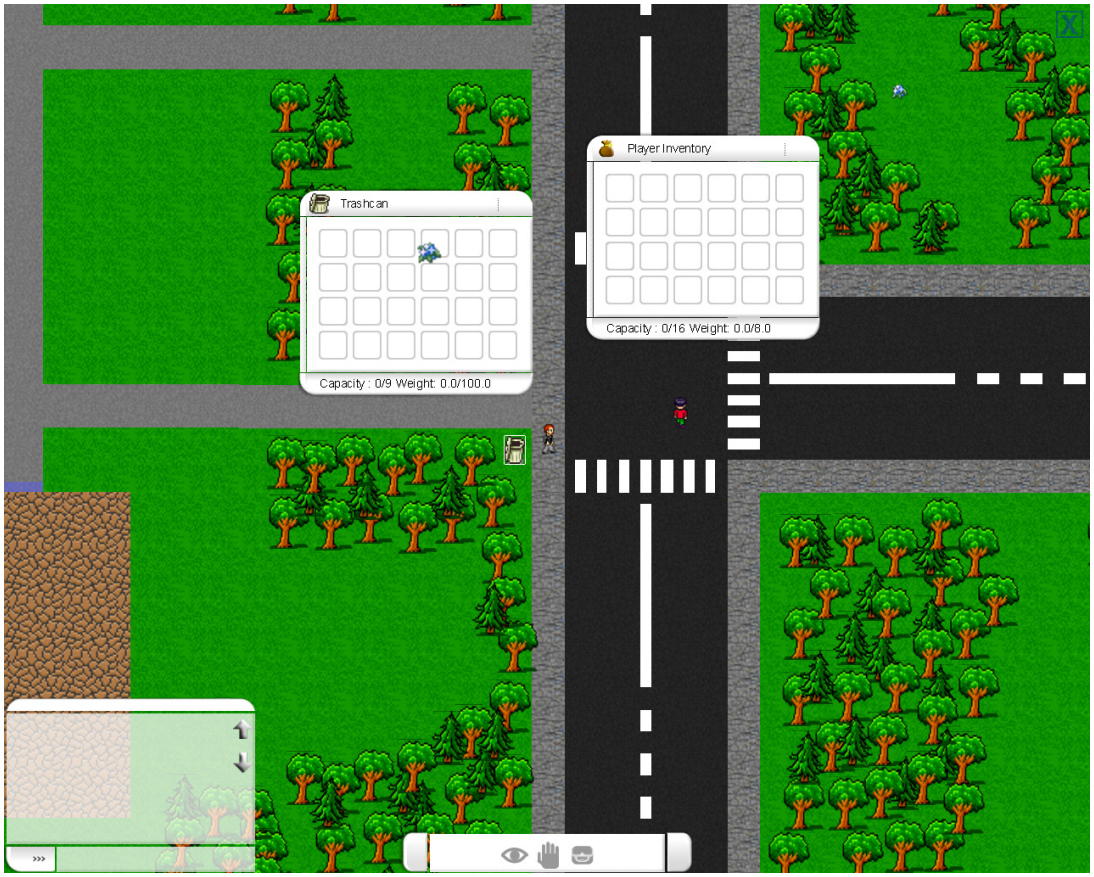


Figure 3–2: Dropping a Flower in a Trashcan

3.3 Potential Conflicts

Besides a lack of gameplay at the time, Mammoth wasn't without technical issues. Especially, with an implementation of items and containers many potential conflicts existed.

It is possible for two players to try and pick up the same item at the same time from the world or another container in it. At this point, there must be a resolution of conflict for consistency to remain. There can also be problems with consistency brought on by network lag. With these problems, both players could end up with

the item in their inventories, which would be incorrect. Similarly, a player who had picked up an item initially could find their item suddenly disappears when a lagged client requests to pick up that same object which they believe is still on the ground.

The added notion of a hand creates a single point to help resolve conflicts, rather than a large inventory object. It also provides an intermediate location where a player can come into possession of the item quickly and then decide where the item should be placed. The two-step process ensures there is less load on the server by separating checks for ownership and container space.

In the case two players initiate a pick up (even if one is from a lagged client), the server can check if a player's hand is empty, and if the item is not owned, assign the item to a player's hand. The second request by the other player would find the item is owned already, and the request would be denied.

Unfortunately, issues arise on clients with partial views of the world, especially clients with network lag. It would be possible for a client to receive updates about an object it does not know about yet or in a different conflicting state. Thus, applying the update fails and results in an inconsistent state on the client.

Of course when an invalid state is present due to lack of consistency, it not only disrupts the gameplay experience but also the basic essence of how the game functions. Further work was required to resolve these consistency issues in Mammoth. Initially, research was done on the traditional client/server system and implemented successfully (see Appendix A.1). However, the solution was not applicable in a peer-to-peer or multi-server environment.

3.4 Object Replication

As part of the most recent work being done on the Mammoth project, an object replication scheme has been introduced, similar to that discussed in Section 2.1. It is the latest functional part in a long standing effort to run Mammoth in a Distributed Multi-Server Environment (DMSE).

In essence, every object has a master copy which is the only copy allowed to change the state of the object. All other copies of the object on other machines (such as clients or other servers) are replicas which are only allowed to read the state of the object or forward state changing requests to the master object. Whenever the master object is updated, the update is disseminated to all the other replicas.

However, while the master/replica system existing in Mammoth is based on the work of Quazal and their Duplication Spaces [28], the rest of the Mammoth system is based upon remote procedure calls completely transparent to the game layer. The game simply invokes the operation on the game object as normal: Mammoth automatically redirects the call to the master object, if necessary. This transparency is not only convenient for developers, but it also makes it easier to migrate master objects between servers for load balancing or fault tolerance reasons.

By using a master/replica system, it is easier for an object to remain consistent as only one copy of the object can be updated. However, the main difference now is all of the game objects no longer need to be centrally located. Unfortunately, more work was required to ensure the system was robust enough to deal with complex actions involving multiple object updates required by MMOGs. This will be discussed further in the next chapter.

Chapter 4
Subgames in MMOGs

As was discussed back in Chapter 2, modern day MMOGs have many issues surrounding their implementation. Two primary concerns are *scalability* to allow more people to play and *consistency* to allow everyone to share the same experience. In most cases, these two qualities are mutually exclusive. Stable platforms such as a traditional client/server model (refer to Section 2.1) provide consistency but do not scale well. Where as a Distributed Multi-Server Environment (DMSE) provides much more scalability but has many consistency issues to overcome.

In this chapter, it will be shown how it is possible to achieve both scalability and consistency in a DMSE by the research in this thesis. First, to solve scalability the definition of *subgames* will be presented, requirements needed for their implementation, as well as issues in consistency brought on by subgames. Afterwards, the solutions to consistency issues in a DMSE will be discussed using a new transaction like protocol and encapsulated actions.

4.1 Motivation for Subgames

In a typical MMOG there is some entity which represents the game world. This "global world" defines how the game is played, how players are allowed to interact with each other and the world around them, as well as miscellaneous rules and regulations for other entities in the game itself. These things can be thought of in more concrete terms as rules, goals, laws, physics, economy, etc...

However, when a world contains many thousands of players, having complex interactions becomes troublesome. It is difficult for a game to regulate many players at once, especially in a simple manner that can be scaled to the numbers needed in an MMOG. It is especially important for all these players to not only interact with each other but with other objects in the game world as well. As the game expands in number of players, features and game items, the number of possible interactions increases, with a naive implementation quickly becoming unscalable.

In order to achieve these goals the notion of a "*Subgame*" was introduced. A *subgame* is, simplistically, a game within a game. In a narrowed definition, a subgame defines an alternate or subset of rules for the game world. E.g. a typical MMOG Side-Quest could be thought of as a subgame.

What differentiates a subgame from a completely isolated shard is subgames are still connected to the global game world, whereas a shard is a completely isolated system. Normally, in the case of an isolated shard, scalability is achieved due to its encapsulation. A shard can be moved or contained on any individual server without the need to migrate data. Subgames maintain this scalability by minimizing and optimizing the crucial data which needs to be communicated between the subgame instance and the rest of the game world. In this manner, a game can appear larger and be more realistic, while encapsulating specific player groups to allow for a more scalable system.

By having a smaller subset of players to deal with, a subgame can ensure those players receive important information about all the players within the subgame. Extraneous information about the game world is not as crucial and can tolerate

slower or lost updates in the case of overload. It allows for players in a subgame to be aware of what is going on in the outside world without disruption of the subgame itself. In addition, it allows for a subgame to be more scalable, as fewer players are interacting with it, or if need be, another copy of the subgame with a different set of players could be created.

A subgame can be thought of as an encapsulation of specific gameplay mechanics and sets of rules. It is meant to be a modular unit which can be isolated or integrated into larger collaborative systems. When a player voluntarily joins a subgame (be it a side-quest or a game of capture-the-flag), the player becomes part of a smaller community who have similar interests or goals for some given amount of time. It is then possible to increase the interactions between these players in a similar group without having to disturb the thousands of other players in the global world.

There can be many different types of subgames having alternative gameplay mechanics, outside the scope of the regular game world, such as Capture-the-flag, Tag, or Hide-and-go-seek. If subgames launch from a global game world in this manner, they must share the global game infrastructure. As such, these subgames must cooperate and interact appropriately within the context of the global game world.

Figure 4–1 shows a generic infrastructure for subgames. At the top enclosing all subgames, is the global world. The global game provides basic rules and regulations which form the core game mechanics. Within this context a *Subgame Manager* provides a framework to manage subgames. Each subgame can be created as a

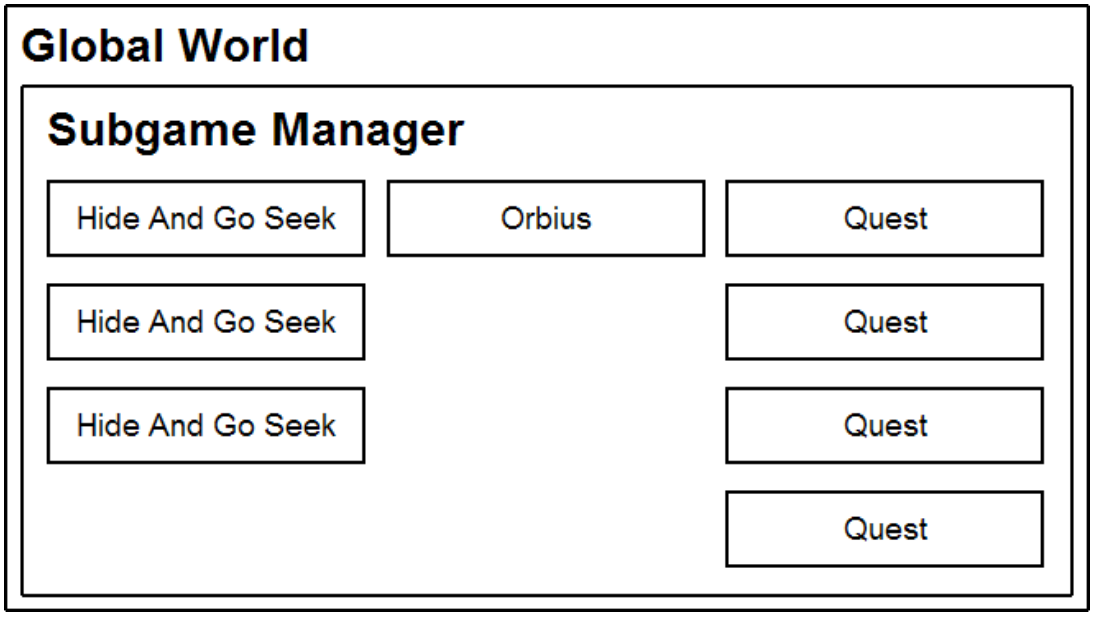


Figure 4–1: Global World and Subgame Infrastructure

unique entity as in the case of a larger game like Orbius (See Section 4.3), or multiple co-existing instances of smaller subgames, like side-quests.

Subgames can be various sizes. In the case of *side-quests*, there may just be a single player joining the subgame to make use of its unique abilities. The quest may just keep track of a particular task the player performs, and upon completion provide some reward. A group game like *hide-and-go-seek* provides a wider reaching interaction affecting dozens of players at a time.

It is also important to note, that while subgame objects are encapsulated themselves, they should not be closed systems (otherwise they would be shards). Imagine the case of a *scavenger hunt* where players are looking for a particular trophy. If multiple instances of the game coexist, it should not matter if the trophy they find belongs to their particular instance of the subgame, as long as they only find one

of them. Since subgames are discrete units, optimizations like this can be made to increase the performance between these similar subgames. In the case of different subgames, actions which can occur between them can be known and thus optimized to allow for greater scalability. By mixing subgames to different degrees in this manner, it not only allows for increased scalability, but also greater flexibility and realism as well.

Subgames solve the issue of scalability in much the same manner as shards. Using subgames allows for a much more dynamic scalable world compared to lumping players en mass. Even though players are somewhat isolated, there is still a greater interaction with the global world. Traditional network techniques such as the lockstep protocol are not sufficient, as they are effected by the slowest client on the network. With such a broadened scope of clients, this can have a great negative impact on the system. How this issue is addressed to enable subgames to work in a DMSE is discussed later in section 4.5.2.

4.2 Subgame Requirements

For a virtual world or global game to be able to support subgames, there are a number of basic requirements that need to be met. These requirements are discussed in the following paragraphs.

Items and Item Types

Obviously, as has been discussed (in section 3.2.1), items play an important role in games, especially in games where the human player plays a single character as in the case of MMOGs. Item objects are the main source of interaction with the game world. Subgames may need the functionality to define their own kind of items, or

re-define the properties of items that exist in the global world. To make this possible, the notion of classes of objects or *Object Types* must exist (refer to Section 5.2). If this is the case, a subgame can define its own object types, or override the properties of existing object types.

Items in the game world are instances of these object types. In order to populate the world with items which are important for a particular subgame, a subgame needs to be allowed to instantiate items and place them into the world. Conversely, a subgame might also require that certain items do not exist in the game world. In this case, a subgame should be able to remove item instances from the game world (or temporarily hide these instances from all the players participating in a subgame). It could also simply be the case that the subgame wishes to remove special items it created specifically for itself.

Graphic User Interface

It is also important to be able to change the image/representation of an item or a player. It may be important for a player in a subgame to know if they are on a specific team or if an item is in a particular state.

Many subgames may also require mechanisms to modify the user interface of the game. For instance side-quests need to be able to display open quests and objectives as well as completed ones for a users benefit. Other games may wish to display the current score or other information vital to gameplay.

Actions

Actions help divide the abilities a player can use into discrete units. Having abilities such as *pick up* and *drop* as concrete units provides a single point of access;

making it easy for subgames to intercept or override the default behavior of the game world. In capture-the-flag for instance, the effects of grabbing an opponent's flag are easily monitored or modified through the object pick up action.

Modularity

As the notion of a *subgame* suggests, it is important to have subgames be a modular entity which can be loaded into the global game world. In this manner, subgames can be loaded individually or in groups and contain all the extra required components such as object types and actions necessary to play the game. It also allows designers to work independently from other games or the global world.

4.3 Subgame Examples

Before proceeding further, a few examples of subgames will be provided. Not only will this provide further context and understanding, but it will allow for visualisation of the concepts being discussed.

Orbius

Orbius was the first subgame created before the notion of the global world and a separate subgame manager existed. It provided the basic understanding for what a subgame was, and what was required for subgames to function. The infrastructure for the work described here was all built up around these notions.

The basic idea behind Orbius was a simple Capture-the-Flag (CTF) type mechanic. Players would be divided into at least two separate teams and assigned a team color. Their objective would be to find five specifically sized orbs (ranging from small to large) scattered across the world and return them to an agreed upon container. After assembling a complete "base" with five of their team-colored orbs,

a “golden” orb of their color would appear somewhere in the game world. Team members would then need to find the golden orb and place it in an enemy base.

In addition there were a few changes to more standard CTF rules. Orbs were quite large and thus (based on size and weight) players could only carry a single large orb or possibly some combination of two of the smaller ones. This created a need to have multiple people find orbs. Also, it was possible to “tickle” another player, a special action which, with a given probability, would have the tickled person drop the item in their hand (or if no item was in their hand, an item from their inventory). This gave an offensive “weapon” to players to disrupt opponents who were farther ahead in their orb collections. Finally, for every player that stood near their established base, any opponent who tried to steal an orb from it would have a greater chance of failing to pick it up.

For the creation of Orbius, many things in the global world needed to be changed. The Orbius subgame needed to be able to know when players interacted with items and containers, and be able to change the default mechanics of taking and dropping (e.g. when a player failed to pick up an orb from a base). It also introduced a new type of item, the Orbs themselves, as well as the ability to dynamically create and destroy them. It also introduced the new action of “tickling” which needed to be incorporated into the game. Finally, the subgame needed to be able to keep track of the players, their associated teams, and the team score, in order to declare victory.

“Find the Trophy”

For validation and testing of subgame designs a simple “Find the Trophy” (FTT) game was implemented. A unique “trophy” is placed somewhere in the game world,

and it is up to the players of the game to find the trophy. Whoever finds it gets a point and becomes "it." They then have a set amount of time in which to hide the trophy again somewhere else in the world. To prevent other players from following them to see where the trophy is hidden, they have an increased speed. Once the trophy is hidden again, the process repeats, with other players seeking out the trophy etc... In cases where the trophy is not found, whoever hid it gains another point and the trophy is randomly moved to a new location for everyone to find again.

FTT offers similar problems and implementation requirements to Orbius. It needs to be able to detect when a player interacts with an object, and it needs to be able to not only spawn an item but move it within the game world too. And while no new actions are introduced, the fact that a player's speed changes was a new requirement, and implies the ability to modify a player's inherent abilities. Scoring is also required, and in addition there are also time factors related to the gameplay which need to be centralised in some fashion.

While FTT has a more simplistic game mechanic than Orbius, it offers a similar range of requirements. By having implemented both games, it is clear subgames require careful planning, and a solid global world infrastructure is needed to have a stable enough environment for practical MMOGs.

4.4 Consistency and Interaction Issues

There are some basic concerns when addressing Subgame principles above and beyond any game-specific requirements. The biggest of these concerns is game state consistency. Without a consistent global game world, it is impossible for subgames to reliably determine what is occurring and when, the game world can easily diverge for

different players, making gameplay difficult and frustrating. Secondly, there needs to be a concrete, modular notion of actions for players to perform in the subgame world. As evident in the example subgames above, actions are often specialized to the game mechanics. Unfortunately, as will be shown below, these two requirements are sometimes opposed.

As mentioned already in Section 3.3, ensuring game state consistency is a challenging problem in the context of MMOGs. For playability and fairness reasons, it is important that all players "see" a consistent state of the game world that surrounds them at all times, regardless of how the state of the surrounding objects is distributed on physical machines. This is especially true for players that participate in the same subgame, since usually the participants of a subgame interact more closely with each other compared to normal players – they not only need to know about each other's positions, but are also more likely to interact with the same game objects. In order for a subgame to be playable, fair, and most importantly fun, the state of the world and the state of all objects relevant to the subgame should be perceived identically by all participants, even if the game state is scattered over several nodes in a distributed system.

The complexity of ensuring the system remains in a consistent state depends on the MMOG architecture and how the game state and game objects are distributed. Clear and precise consistency protocols have to be put in place. Once protocols are in place, actions can be built up upon them to allow players to safely interact with the global game world without worry of corrupting the game state and becoming inconsistent.

4.4.1 The Consistency Problem

As has been repeatedly mentioned, item state consistency in MMOGs is a big concern. If the game is not consistent, then problems can arise and attempts to correct invalid states must be made. It has been discussed how current methods are expensive, dependent on the number of players (which for MMOGs is very large), time consuming, and requires extra systems to be in place. Furthermore, invalid requests can continue to be made during any rollback process taking up more precious computing resources. It is therefore advantageous to spend slightly more time upfront to ensure the game state is always consistent across the network.

In a typical MMORPG, most interactions do not need to occur instantaneously, and thus, taking the extra time to maintain consistency is more beneficial for the game and not as noticeable to the player. Traditionally, inconsistency can be an acceptable tradeoff for performance in some situations (especially when it comes to movement); however, many MMORPG actions such as item interactions require strong consistency regardless for the game is to remain playable.

Maintaining consistency of a game in a client/server setup is not a trivial thing; even in this simple environment, network lag and other timing concerns can cause consistency problems. And of course, the problem only gets more difficult when dealing with a multiple server environment, where information itself is distributed. Below describes consistency problems which arise due to game actions depending on how information is organized.

Need for item information in two spots

For basic game actions, such as picking up or dropping items in containers, information about the location and owner of an item needs to be stored in two locations. The item itself needs to know if it is inside a container, and the container it resides in needs to know what objects it contains.

Note that the object and its container are two logically separate game entities, and updating the game state due to picking up (or dropping) an item requires at least two separate activities – an object must be removed from its current location, and placed in a new location. Without some effort of consistency control the two activities required may execute non-atomically, or in different orders, and having the item appear both on the ground and in the player's inventory at the same time is unacceptable.

Of course, if information about an item's location was stored in one object it would allow for optimal consistency (as there is only a single entity to update). In a single server instance with proper synchronization, for instance, a single point of update would not be a problem; even in the distributed case with object replication (refer to Section 3.4), a single update could be handled.

Unfortunately, it is not as simple as being able to just store location information in a single object. For a practical, functional game, item location information is distributed, with individual object data residing close to the site of most active use. The necessity splits up information. Below several implementation strategies are discussed for dealing with this consistency problem in a distributed environment.

Item knows all. An obvious and single solution is to retain ownership information itself in only one place. If only the item knows its location and owner then it is easy to check if the item can be picked up or dropped by another entity. Unfortunately, if an entity wants to know the objects it contains in its inventory it needs to iterate through all possible items in the world. Of course if the entity is a client or in a distributed environment it may not even have access to all the possible items in the world. Therefore, finding one's enclosing container is a constant O(1) operation, whereas inventory lookup is O(n), where n is the number of items in the world.

Container knows all. Symmetrically, it is possible to store ownership information entirely in the container. If only the container knows the items it contains then it is very easy to display the items in a container, by simply iterating through the list of items inside. To find out where an object is in order to check the validity of a pick up or drop, however, becomes much more complicated as all containers must be searched for the item. In this case, retrieving the inventory list is a constant O(1) operation, whereas the pick up check (to determine if an item is owned by another player) involves querying all "n" containers in the game world, an O(n) operation.

Third party. A third possibility is a third party lookup system which contains both information about containers and item locations and ownership. In this model, the items and players themselves know nothing about where they are located or what they may contain.

This design is attractive for its simplicity and ease of atomic updating. Unfortunately, all checks and updates for both an object and its container, must be processed

by a single location. This makes load-balancing awkward, and as containment forms a connected data structure, this implies a single node will be responsible for all object operations. In this sense it is no different from a traditional client/server framework with increased message overhead.

A Solution. Given the trade-offs, and despite the greater consistency problems, the design used in this thesis stores ownership information in two locations. The object knows in which container it may reside. Containers also know their entire contents. This allows for fast checking of either ownership or inventory (which occurs often), at the cost of having to update distributed data in an atomic fashion.

4.4.2 Actions

Actions were created to aid not only in the consistency of the game world, but provide a better framework for these changes as subgame design relies on a modular, easy way to add, remove, or modify game actions. An action thus encompasses almost any concrete state change that a player or game entity can perform, such as *pick up* or *drop*, and as opposed to general game mechanic implementations, everything relating to the state change is encapsulated within the action itself. Consistency checks, feasibility checks, and the actual performance of the action itself are all contained within a single code module for each individual action which can be performed.

By separating actions in such a manner it becomes much easier to monitor changes in game state, interrupt those behaviours, and to expand the set of actions available to the game. Expandability in this manner is especially important for subgames which may wish to define their own new sets of actions to increase gameplay as well as interrupt existing ones.

Actions in a Distributed Environment

When dealing with a distributed environment, having encapsulated actions also helps to ensure that there is a coordinator between the various objects which need to be changed. The action object itself thus serves as a central point for coordination, and is responsible for ensuring consistency if multiple objects are involved.

Not all subgame activities are neatly captured within action objects; however, further design issues will be discussed in Sections 4.5.3 and 5.4.

Conflicts

When dealing with realtime games, potential exists for two separate actions to be performed at the same "moment" in time on a single object. These conflicts can occur easily when there is a client with a larger network delay between the actual update of an object and the message reporting the update of the object. When an event occurs as such, there needs to be a way to determine which of the two "simultaneous" actions occurred first or will take precedence.

Imagine a scenario in which two players wish to pick up the same game object. In this case, only one player should end up with the item and the other should see the item disappear as normal. From a player perspective, precise correctness here usually makes little difference, as a player who successfully picks up the item on the ground will be pleased with picking it up before the other player. Conversely, the other player will just have assumed the other player reached the item first regardless of the actual time the messages were sent on their respective machines.

It is therefore possible to arbitrarily choose one of the two conflicting messages to take priority. Usually, this is automatically done by whatever networking code is

in place; if a single thread exists to apply these updates, or the object becomes locked upon change, then it is impossible to simultaneously apply these updates together. Therefore, as long as checks are in place, one update will be applied, and the other will fail as it can no longer be performed on the new state of the objects.

Such a simple, centralized solution to atomicity becomes significantly more complex in a DMSE, particularly when actions require a simultaneous update of distributed game data. In the next section several solutions are discussed, culminating in a flexible, generic abstraction.

4.5 Solutions

For a subgame architecture in a DMSE requires solutions to several problems. Key among them are the need to support a modular, flexible action design, and the necessity of ensuring consistency for all players.

There are thus two main parts to these problems. One is the general consistency of the game world itself in a DMSE. The other is how to allow subgames to override the default behaviors and change how the world reacts to certain actions. Below we discuss these concerns and present a specific solution. Actual implementation details are given in Chapter 5.

4.5.1 Consistency Resolution

Maintaining consistency is the largest issue discussed in this thesis. Consistency bears heavily on the core mechanics of an MMOG, and it is important to both the game itself and any extensions to the game as well, such as subgames.

In the case of a single server, consistency is not necessarily a complex problem: a single process or thread synchronization can maintain consistent control of an object,

and a single network message can consolidate the state change information so clients can update both objects at once as well. As has been mentioned previously, this scenario fails to be applicable to a DMSE. The following section will explain how to combat these consistency issues in the context of three core, generic cases. First, the simple single object update case is addressed, followed by the two object update, and finally the case where three objects must be updated simultaneously.

Single Object Update

The case of a single object being modified in a DMSE has little difference from a traditional client/server setup – mainly it is an issue of a different server handling a request, based on the distribution of game objects.

This is true even in the context of many clients connected to the system, through many different points, and other servers making requests as well. Even with the potential to create many simultaneous requests to change an object's state, by ensuring only one update/request can occur at a time through a simple access lock consistency can be maintained for single object updates.

As was discussed briefly in Section 3.4, replicated objects play a large role in creating a DMSE. They also help ensure consistency during single object updates.

Replicated objects help accomplish this task by not only ensuring there is only one master copy of an object that may be updated, but also by disseminating the changes to all the replicas of the object. With such a system in place, updates such as player movement or scoring can be easily maintained.

Versioning

Sequentializing single object updates eliminates consistency problems, but does not ensure updates are correctly applied in order. There can be cases where two requests were made on an object, but only one should be applied. Once the first request is applied, the second request made on the original state of the object (and not its current new state) should be automatically denied. The second client must then recheck the conditions as they are now that the earlier update has been applied and perform another request.

To monitor outdated requests, a "versioning" scheme is applied. The master object starts at version zero. The version number is replicated along with the normal object data to the replica proxies. Every time a "versioned" request is made from a replica, it sends its current version number. The master (after acquiring its lock) can then compare the version number in the call request to the version number of itself. If it does not match, it rejects the request; otherwise, it completes the request as normal and increments the version number.

Normally, with a versioning scheme, starvation is a concern. In the case of an MMORPG, requests are usually initiated based on the proximity to the desired object. In these cases, there is a limit to the number of players which can surround or interact with a given object if physical space is mimicked within the game world. In addition, in the event a version number is outdated on the request, it was probably the result of another party picking up the desired item. In this case, the request is denied which would have been the outcome regardless.

With versioning in place it is not only possible to order simultaneous requests, but also to ensure even lagging clients do not make use of old data. If a client has yet to apply an update (and receive the latest version number) and then makes a request, it will be rejected. However, concerns with continually outdated clients, which lacked the necessary state information to ever make a valid request, were not the focus of this thesis.

4.5.2 TTLS: Timed Test Lock and Set

Having lockable and versioned objects works quite well for single object update scenarios. However, in cases where updates to more than one object need to be applied consistently is a more difficult problem. If a player and an item both need to be updated simultaneously (as previously discussed in Section 4.4.1), a transactional system is necessary to update information in two places.

Imagine a scenario where a player wishes to pick up an item off the ground. The player needs to be told they have picked up an item, and the item needs to be told it is now in the possession of the player. These two updates need to happen at the same time to ensure the item knows about the player and the player knows about the item without interference from another party. On the other side, the client needs to update both of these objects at the same time in order to not see the player holding the item and the item lying on the ground simultaneously.

Normally, in distributed systems such as databases, these sorts of simultaneous updates are applied in a transactional manner. With transactions, locks are first acquired on all the objects, updates are applied, and then after everything has been approved, a commit message is sent to all objects to apply the changes. Then each

object can unlock and allow further changes. This process may not seem too long or be an inconvenience in its normal setting; when applied to games however, waiting and delays are unacceptable.

Typical MMOGs have many requests occurring simultaneously, and there is also an expectation of immediate feedback. A player taking an item expects to see the item in their hand as soon as they have requested to pick it up. If this process takes too long, players become frustrated and lose interest in the game. A full-blown transactional system with distributed locking is far too cumbersome for what needs to be achieved.

Fortunately, in the case of basic game conflicts, such as two players attempting to pick up the same object at once (see Section 4.4.2), it is easier to resolve than in a normal transactional system. A game can freely read the state of any object without locking, as it is assumed this data is relatively accurate and up-to-date. It can be used to make decisions and display information to the player. Any inconsistencies with the "real-time" data would become apparent if the player requests to change any data. For any such changes, a write lock is required to secure the data and ensure no other party (be it another player or the game) changes it simultaneously.

Normally, when a lock cannot be acquired the system waits for the lock to become available. Waiting on a lock is where the majority of transaction systems can build up time and be unsuitable for use in games. Instead of waiting, we can "abort" the request and return. Since reading the state of the object does not lock it, the assumption is that if the object is already locked, it is currently in another "transaction" or state update. More than likely, the state of the object will have

changed when this object becomes unlocked (e.g. another player picked up the item just before the player). In such a case, the original request which was blocked by the locked object would be invalidated by the change in state and version of the master object. Therefore, rather than waste time waiting to receive a denied request, it is faster and better for the player experience to return immediately. This scheme also avoids any deadlocks, which can occur in traditional transactional systems, as no waiting ever occurs on a locked object (another important factor in the case of games). And as long as a consistent locking order is maintained i.e. the player is locked and then the item, there will never be a case where two requests for the same item both fail because of acquired locks, unless there is a third party involved.

The design presented here is called "Timed Test Lock and Set," or TTLS for short. Its goal is to provide a partial locking system in a distributed environment, allow for a transactional type update system for multi-object requests, and to do so without the overhead of traditional distributed locking or atomic commit systems.

In the case of games, most action requests can be broken down into updates of two or three objects. The two object case has already been described; for a third object scenario, imagine a player wishing to place an item in a container. Not only must the player and item be updated, but the receiving container as well. These two main cases will be discussed in detail within the workings of TTLS and a game environment. Afterwards, a summary of the associated message costs for varying strategies will be provided.

Two Object. The process of TTLS starts with an *action* which encapsulates the needed game mechanics and acts as a coordinator for the series of updates.

Ideally, it is executed remotely on the same machine as the master player object, as players are in most cases the instigators of actions, but this exact procedure is discussed in more detail in Section 5.4.2. However, it is important to note, the order actions are performed in is important to prevent multiple concurrent actions from all failing, as mentioned previously. Therefore, we will assume that the player object is always locked first.

Figure 4–2 represents a UML Sequence diagram outlining the TTLS procedure. The first step in TTLS, is to lock the first object (the player in this case) as normal; however, we must flag the start of the process as a TTLS procedure, so the player will not release its lock right away as if it was a normal call. The player object will then confirm that the action is possible. Normally, without TTLS, the player would unlock and propagate the change as an update; however, with TTLS those steps are delayed until later. It is important to note, if this action was indeed executed on the same machine as the player, the initial TTLS call and locking of the object is localised.

The second step of TTLS is to execute a call on the second object. This could be a local call or a remote call depending on where the object is located. This call proceeds just as normal, as it is the last object in the sequence. The object will check if the requested update is possible and if it is, change its state and publish the change as normal. The return call is then sent to the requester (just as normal), and the player object can then be told to finish the process by sending an update message and releasing its lock.

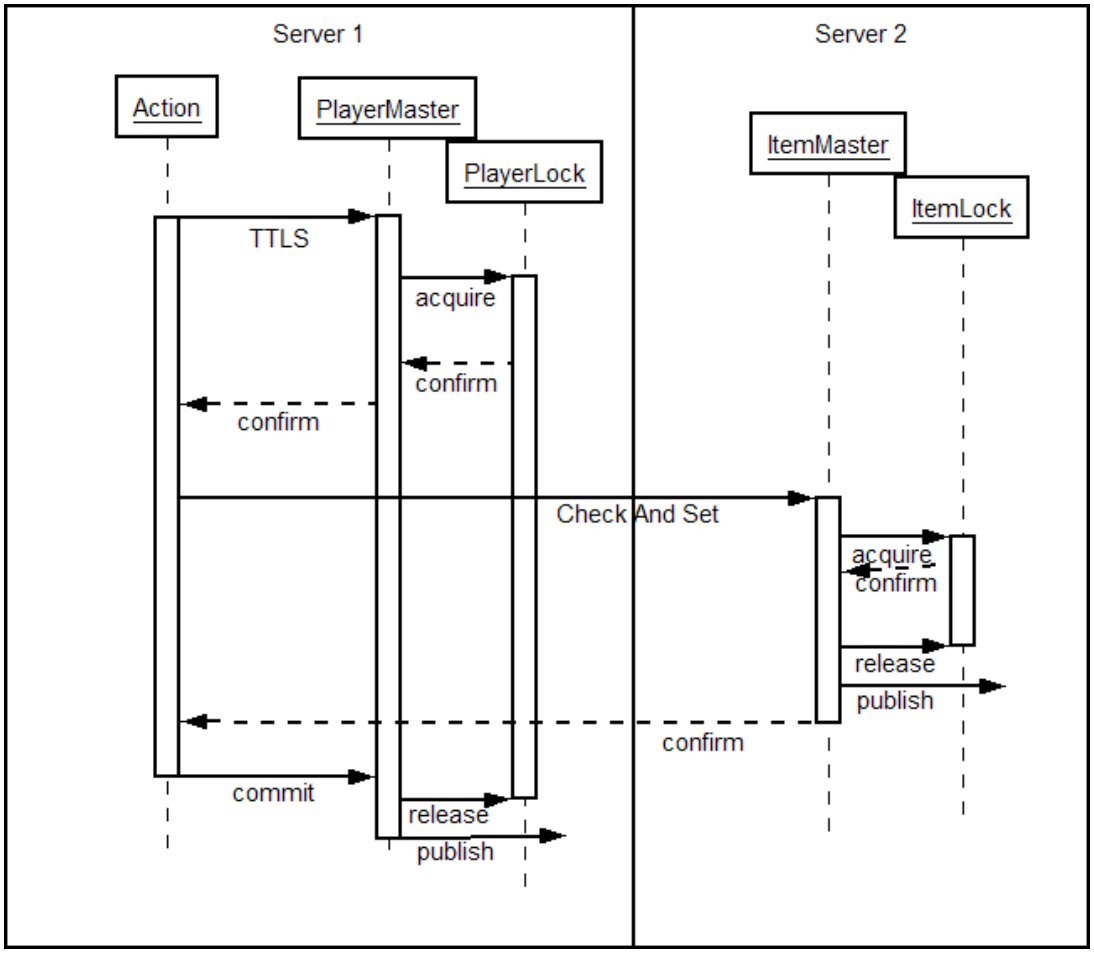


Figure 4–2: TTLS with Two Objects: Pick Up

Unfortunately, while Figure 4–2 shows a successful "simultaneous" update, it does not mean anything if the client receiving the updates does not apply them at the same time as well. To combat this problem, a *nounce* was introduced. A nounce is simply a way of identifying the transaction and each update within it. In this manner, a client can wait for each update in the transaction before applying

them. Figure 4–3 shows the updated TTLS protocol with the nounce creation and propagation. Nounces will be discussed in greater detail below.

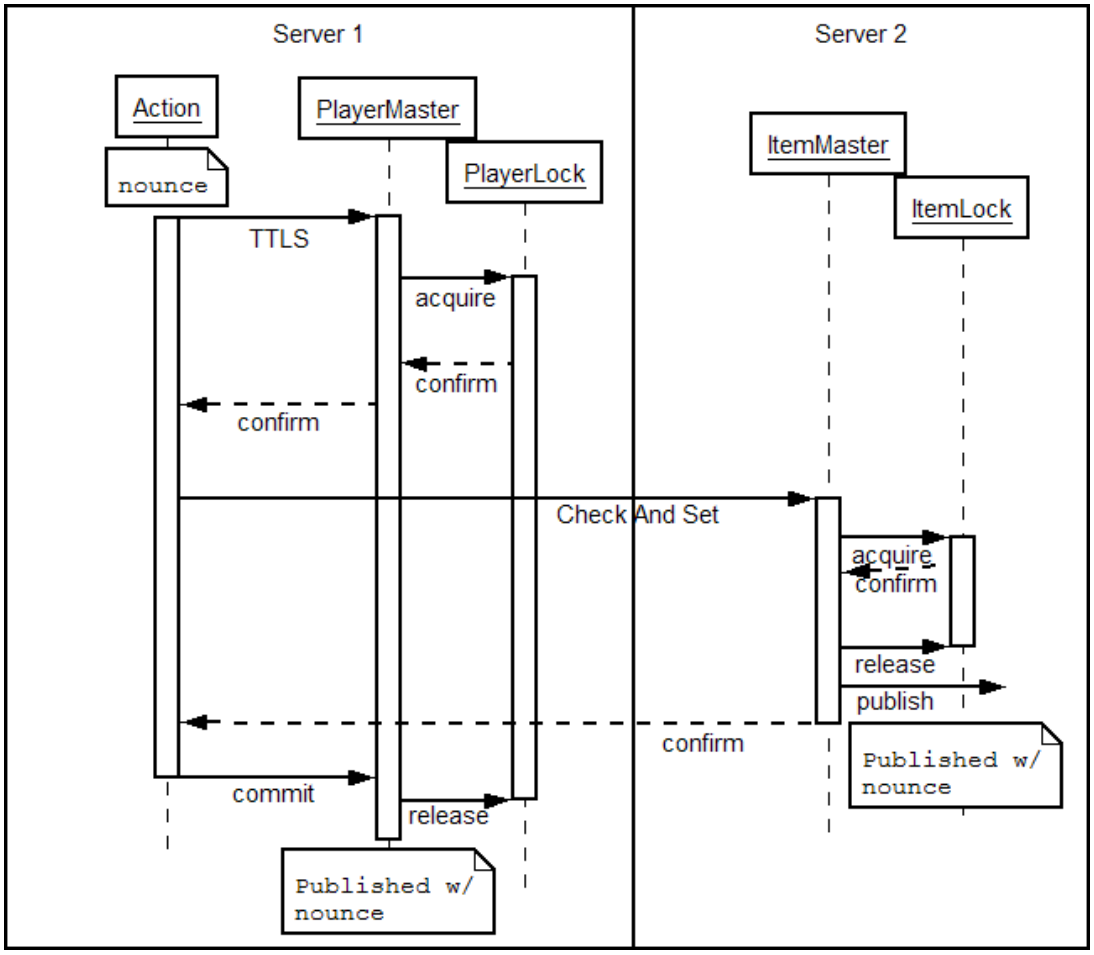


Figure 4–3: TTLS with Two Objects: Pick Up w/ Nounce

Fast response is essential in games, and so TTLS includes a basic timeout mechanism. This occurs when the first object has waited too long for a response from the second object. Failure is assumed, and the first object aborts its update and the action is cancelled. Other methods of failure are possible; however, the focus of this

thesis was more on ensuring consistency during normal operation, and not recovering from node failures, which would involve more complex mechanisms to repair the state of the system.

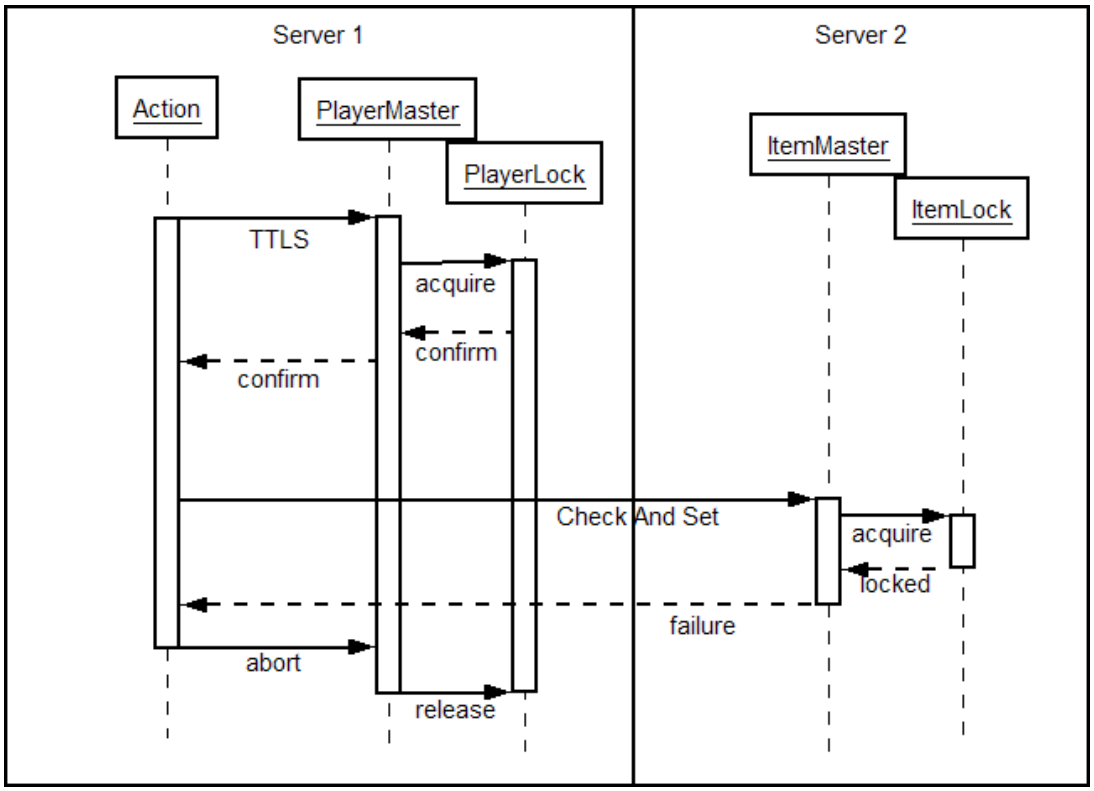


Figure 4–4: TTLS with Two Objects: Failure on Remote Object

With ensured network protocols, focus was given to failure scenarios revolving around object state and the sharing of objects. Figure 4–4 shows the case where the second object in the transaction is locked (or otherwise fails) on the request. When the second object returns with a failure (or theoretically it could not respond and let the first object time-out to save a network message at the expensive of lock time), the first object must undo the changes it made, restoring the state of the object

to before it was locked and unlock itself. This "rollback" could even be saved by not initially applying the request (though checking for its validity) until the commit message is received, as the object will have remained locked.

Of course failure does not only have to be the case where the object is locked. It could also be that the version id of the object has been changed since the action was initiated, meaning some update has invalidated the state the action presumed. Another reason for failure could simply be the case where the item is not able to be picked up. Maybe the item was picked up by another player before we received the update and we initiated the update. In this case, while the object is no longer locked, its version id would be different.

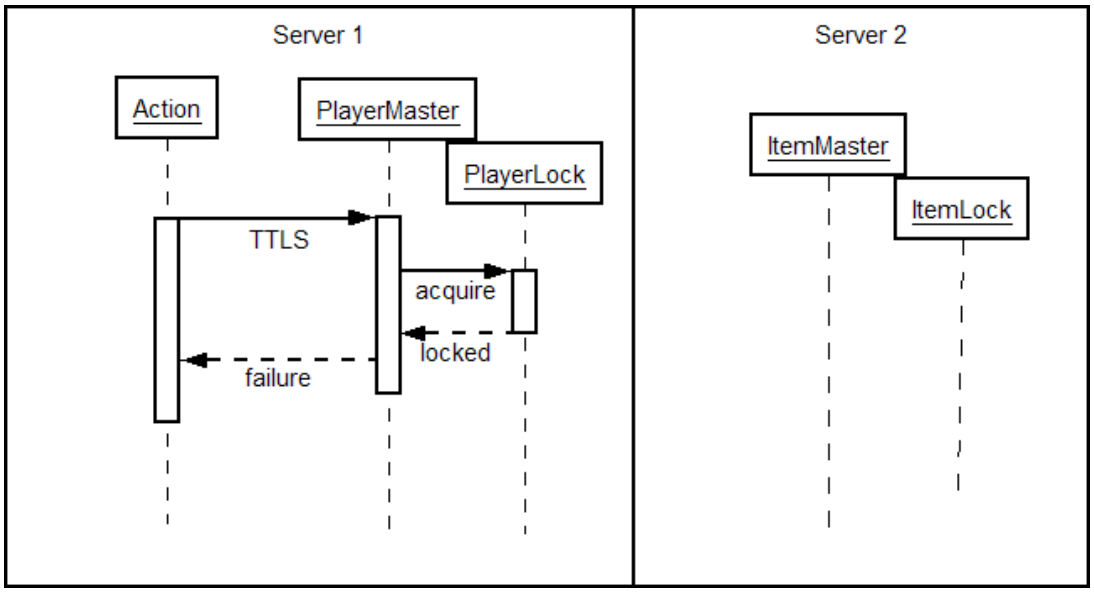


Figure 4–5: TTLS with Two Objects: Failure on Initial Object

Finally, the most simple case is where the initial object may already be locked or have changed since the request. Figure 4–5 shows how an action wishing to perform

on an outdated or locked object will simply result in failure before any interactions are requested over the network.

Three Object. There are often situations in a game where three objects need to be updated simultaneously. Imagine a player wishing to put an item in a container within the world. Here three objects are involved: the player, the item, and the container receiving the item as well.

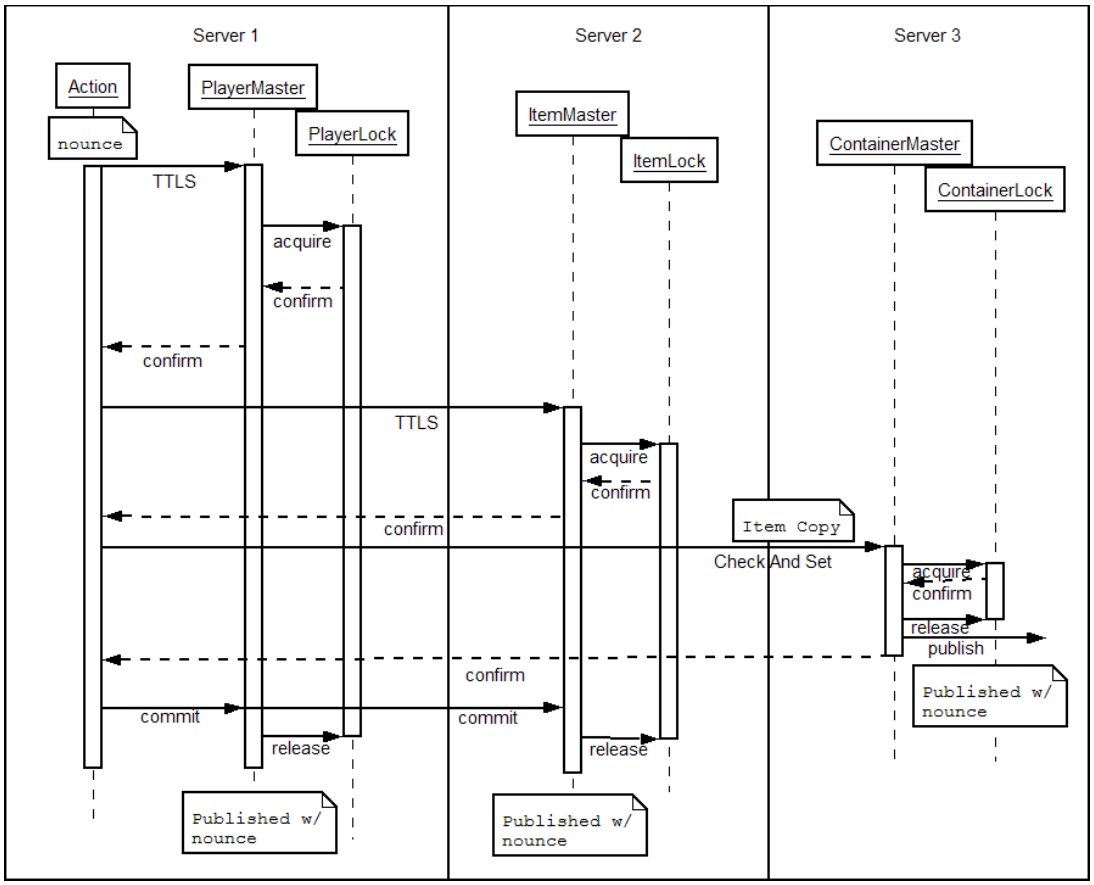


Figure 4–6: TTLS with Three Objects: Pick Up from Container

Fortunately, the initial TTLS system presented here also is scalable and extends well to three objects. Figure 4–6 shows the full successful protocol UML Sequence diagram of the three object case. The order for these operations is to start with the player as mentioned previously, followed by the item and then the container. It is also important to keep the container locked for as little time as possible, as there may be other players wishing to add or remove other items from the container, and their actions should not fail because of the actions of another person.

As shown in Figure 4–6, TTLS is extended by performing another TTLS request on the second object as well, keeping its updated state locked without a published update message until final confirmation has been received. When the third object returns as normal, confirmations are sent to the first two objects to proceed as normal.

Of course, theoretically these procedures could be further extended to more than three objects, but then message overhead and lock time start becoming problems. However, in most normal cases, game actions can be broken down to involve at most three objects. More complicated things such as trading items between players typically involve more complicated systems anyway, as an actual transaction would need to occur, and since it is involving items players' already own, they can be easily locked and controlled. In addition, complex multi-object interactions such as trading are not usually as time-sensitive. Exact specifications and costs on messages will be discussed later in this section. Importantly, however, it has been shown that a system can exist which implements a transactional-like update system in a context which is suitable for fast-paced gameplay mechanics.

Observers and Nounces

While TTLS ensures that requests are applied to master objects "simultaneously," the corresponding updates also need to be applied to clients at the same time; otherwise, inconsistencies can arise, such as a player seeing the object they picked up in their hand and still on the ground at the same time.

To solve this problem a *nounce* is used. A nounce is a form of identification associated with the update to alert the client that it needs to wait for multiple pieces of information before proceeding. In the case of TTLS, actions, and replication, the nounce contains a list of all objects involved in the "transaction." Thus, each nounce has an id number corresponding to the action transaction, and a list of all objects involved. Furthermore, each nounce sent along with a RPC request, is specifically tagged to the corresponding object the update will apply to; i.e. the player take action request's nounce is tagged with the player object's id.

When a client receives an update message it should first check for a nounce. For single object requests, no nounce is needed, so if no nounce is found the update is applied as normal. Otherwise, the nounce should be added to the pool of nounces corresponding to the nounce id number. Later, the client can check if it has obtained an update for each object in the nounce's object list. For example, in a take action the nounce states the update applies to two objects, the item being picked up and the player picking it up. Therefore nounces are required from updates for the player and the item. If a nounce and an update have been obtained for each object in the nounce list, then the client can apply all the updates at once.

In the case where not all the nounces have been received, a check is required to ensure that the client is actually interested in all the objects in the nounce. If, for instance, the receiving client is interested only in the item, but not the player picking it up (just beyond their "sight range"), then the client can apply the update stating the item was picked up and ignore the fact that the player has picked it up, as the player is not of interest. Now that the item was picked up, the item is no longer of interest either (as it belongs to a player out of sight range). Therefore, if after searching through the nounce object list (while ignoring nounces which have already been received) there remain only ids referring to objects which are not in the interest range of the client, the updates can be applied without worry of inconsistency. Otherwise, the client will continue to wait for another update message containing the nounce for the missing object. In this manner, it is simple to apply the updates simultaneously on the client as well at the negligible cost of a small increase to message size.

Message Overhead

While TTLS provides consistency in a DMSE, the efficiency of TTLS has yet to be addressed. TTLS has less overhead than an atomic-commit transactional system, but does not have the speed of a simple distributed design where network messages are sent simultaneously. Overhead, for TTLS, varies depending on the number of objects involved. Below the single object, two object, and three object cases are discussed in greater detail.

For TTLS, the single object update case is very efficient, as there is no additional overhead in terms of network messaging. The only difference between a normal

distributed system with no consistency checks and the TTLS version is the object versioning and simple-access locking systems (as described previously above). Since the request fails on an invalid version or if the object is locked, there is a negligible amount of overhead to the system in the case of direct contention.

In the two object case, TTLS needs to send an additional message to the second machine if the master objects are on different servers, where as in a simple distributed setting these two messages would be sent simultaneously, saving time. Notice, however, the second message in TTLS is between two servers which in many cases may have better network connectivity than clients. Also, in the case where the two objects are on the same machine, the extra message for TTLS is not required at all.

The three object case is, as expected, a little more complex, as it borders on a more transactional system. Nevertheless, the final update is published immediately after the third object has verified the request. This is therefore still faster than waiting for a final handshake commit in a traditional transactional protocol. Again, if multiple masters, involved in the action, exist on the same machine fewer messages need to be sent and a faster response can be obtained, and as in the two object cases these extra messages are all executed as inter-server traffic, possibly exploiting a faster server network.

Finally, the case of three object TTLS can be specialized, if a player is placing an object in their own inventory. In this case, the problem can be reduced to the two object TTLS as the player is both the actor and the container. Since the object

master of the player is obviously on the same machine of the container (which is also the player), less overhead is required.

So, while TTLS does incur some additional overhead to maintain consistency, this overhead is potentially lessened by extra messages occurring on a faster inter-server network, and in many cases can be reduced down to an optimal state which matches the effort required in a non-consistent distributed system.

4.5.3 Action Resolution

A usable subgame architecture requires a modular design for actions, as well as appropriate consistency resolution. The design used here consists of an *Action Controller* regulating different *Action Behaviors* associated with classes of actions, or *Action Types*. Figure 4–7 shows the general layout of this system starting with a requested *Action Instance* and finishing with the *action behavior* to be executed.

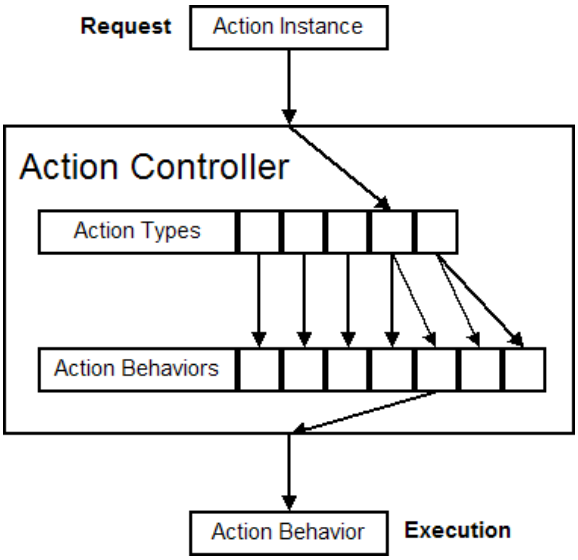


Figure 4–7: Action Controller Regulation Design

Action Types and Action Instances

As part of action resolution, a system must be in place to distinguish between the different types of actions available. An *Action Type* is basically the shell of an action. It is a container which bundles and describes what objects will be involved in the action. When a player wishes to perform an action, they create an instance of an Action Type or an *Action Instance* which specifies the components involved in the action, such as the player and the item they being picked up.

When a subgame wishes to create a new type of action, it can simply create a new action type which classifies itself by what type of objects it acts on or utilize an existing type. The "tickle" example from Orbius (refer back to Section 4.3) is a good example of a new action which is outside the normal class of actions.

While action types classify actions, *Action Instances* specify what objects are involved with the action itself. Action instances, however, perform no operations on the objects they contain; for that *Action Behaviors* are needed.

Action Behaviors

An *Action Behavior* is associated to a type of action and is the actual "workhorse" for the action system. An action behavior is uniquely instanced and has two functions, the first to validate an action instance's objects so the requested action can be performed, and secondly to perform the action on the requested objects in the action instance.

While a default action behavior is created for each action type in the system to create the normalm, "global world" game mechanics, it is also possible for subgames to subclass or create their own classes of action behaviors for existing action types.

A subgame action behavior can then specify that it would like to be used instead of the default action and take place of the behavior, for example when a specific type of item is picked up (like an Orb). In this manner subgames can co-exist as long as they work and intercept different classes of items. Note that behaviors themselves are independent and do not have knowledge of other behaviors; for this purpose there is an *Action Controller*.

Action Controller

The Action Controller is an instanced class existing on every machine in the system. Its purpose is to load all the behaviors for all possible action types and allow the system to perform those actions.

When an entity (such as a player or other game system) wishes to perform an action, it creates an instance of the `Action Type` corresponding to the mechanic they would like to utilise (such as pick up or drop). This action instance holds the objects involved in the action (such as the player and an item). It then gets passed to the action controller.

The action controller will then proceed through the list of action behaviors associated with that particular action instance. Each action behavior will be asked if it would like to perform its action on the set of objects. If they respond yes, their action gets performed. Otherwise, another action behavior is queried. This process continues until the default action behavior gets checked at the bottom of the list. If that fails, the requested action cannot be performed.

In the case of a successful action behavior being found, its execution is handled either locally or remotely depending on the type of objects involved in the action. This is discussed later in more detail in Section 5.4.

Chapter 5
Mammoth Implementation

While initially the concepts discussed in the previous chapter can be easy to grasp, implementing them can cause other problems. To help complete the discussion, an overview of the research and implementation work in the context of subgames for Mammoth follows.

The first subsection introduces the *Subgame Manager*, the central component that coordinates all subgame related activities. The reasons for its introduction are explained together with a brief history on the previous architecture of Mammoth. Following the discussion on the subgame manager, a detailed look at XML Object Types and and how the data structures for inventories had to be changed is presented. Finally, the action controller framework is explained. Additional work related to Mammoth implementation difficulties, but not directly related to the research in this thesis can be found in the Appendix.

5.1 Subgame Manager

As explained in the previous chapter, it should be easy for a researcher to design their own subgame, and plug it into the Mammoth framework. At a given point in time, there could be multiple subgames available to players. Players should have the option to be able to join an existing subgame instance or create a new one if they so choose.

To this aim, a *Subgame Manager* component was added to the Mammoth framework. Its main purposes is to take care of several things: the loading of different types of subgames, the instantiation of specific subgame instances, the starting of an instance, the maintenance of the subgame hierarchy, the joining and leaving of players, and the coordination of running subgame instances, if needed. Every machine in the Mammoth network has its own subgame manager which maintains the same set of subgames available to all players.

In the first prototype implementation during the first months of this thesis (where Mammoth was still running in classic client / server mode), the subgame manager existed on the client and the server, where it loaded the subgame modules specified in a Mammoth property file during startup. Once loaded, a subgame could be instantiated and take control of the virtual world. The subgame manager would then directly intercept a game message (such as pick up and drop) coming from the network and prevent or modify its execution, if a subgame so desired. However, it was realized, especially in a distributed environment and with multiple subgames, that such a scheme was not flexible or scalable, as multiple subgames might have conflicting views on a message, and distinguishing these conflicts was not possible within the architecture (at that time).

Two changes to the Mammoth framework made the implementation of the subgame manager more elegant and modular: the introduction of the action controller (see Section 5.4), and the object replication network interface (see Section 4.5.1).

The introduction of the action controller framework made it possible for a subgame to simply define a new set of actions relevant to itself. At startup, the subgame

manager loads each subgame's set of actions into the action controller. Whenever an action is executed by a player, the action controller dispatches the command through the various subgame actions in sequence, regardless of if the player is playing a subgame or not; for instance, a subgame might allow other players outside the subgame to see an object (belonging to the subgame) but not allow any interaction with it. If a subgame does not explicitly define an action (whether permissive or dismissive) then the action controller looks for a corresponding action in another subgame; or, if there is none, in the default set of actions for the global world. A point of future research is to evaluate how to detect and handle conflicting action behaviors with regards to subgame instances (see Section 6.1.1).

In addition to defining new actions, subgames can also define timers and specialised scoring. In a distributed setting, it is important to ensure that this information is available to all players and updated in a consistent way. In the original client/server Mammoth implementation, this information was kept on the server. This can not be done in a Distributed Multi-Server Environment (DMSE). Luckily, with object replication, the solution is simple: subgame instances themselves can be implemented as replicated objects. The replicated subgame contains all important information related to the subgame, for instance the current score or a list of players playing the subgame. All subgame participants have replicas of the corresponding subgame master object and can therefore access the scores and other important information directly. Whenever subgame information needs to be updated, the player simply calls the corresponding method on the subgame object. And thanks to the object replication technique, this call is transparently forwarded to the machine that

currently holds the master subgame object. The state change is executed on the master machine, and the changes are broadcast to all participants.

5.2 XML Object Types

The world of Mammoth (or world map) is loaded from an XML file. In the previous version of Mammoth, the notion of item types (see Section 4.2) did not exist. Hence, the XML file would define every item instance individually. As a result, even if there would be, for example, several identical tomato instances in the world, the XML file would contain the definition of a tomato multiple times. First of all, this is pretty inefficient, as it creates a larger XML file to parse. More importantly, though, if someone wants to add or change the property of a set of identical items, like for example change the weight of all tomatoes, every tomato instance in the XML file would have to be modified.

To remedy this situation, and to allow subgames to define their own item types, the notion of *item types* were introduced and then later more generic *Object Types*. It is a common structure which defines the properties of an object (item or scenery) in the world and that can than be "instantiated" in the XML. As a result, the information about all tomatoes (for instance) is centralised, and hence it is possible to change the representation of a tomato, its description or its properties by modifying the XML definition of the tomato type. Then, all the actual tomato instances in the world will change accordingly.

In addition, it was made possible to "override" the default properties defined in a type from within the instance. In the instantiating object XML tag, the normal properties for objects can still be specified. In this case, at instantiation time, the

properties defined by the instance are used, if specified, and only the properties that are not specified in the instance are taken from the corresponding object type.

With the new system, the XML map is a more compact representation of the world, and it is easier to maintain and update. It is now also possible to instantiate a particular type of object at runtime as long as a base type exists within the game to create new objects. Object types also form a hierarchy (just like classes) which can be helpful for classification purposes, such as having a rose count as a flower to a subgame about collecting flowers.

Apart from actions (see Section 5.4.1), object types are one of the main features for subgames to implement play mechanics. For instance, a subgame can create a new "flag item type" from which it can instantiate "flags" in order to implement a capture-the-flag game. Likewise, it is possible to create some sort of reward trophy for winning the game. Using the classification mechanism of object types it is now also possible to define rules such as: "a player is only allowed to pick up fruit," a fruit being any item type that is a subtype of the fruit type, for instance an apple, orange, or banana.

5.3 Inventory Refactoring

Originally, the Mammoth engine had inventory objects existing both in players and containers. However, both implemented a common interface to an inventory object, and most instances of inventory manipulation were about retrieving the inventory object and manipulating it directly. Figure 5–1 represents the old architecture of Mammoth. Unfortunately, with the move to a distributed environment and Remote Procedure Calls (RPC), the direct approach does not work as well.

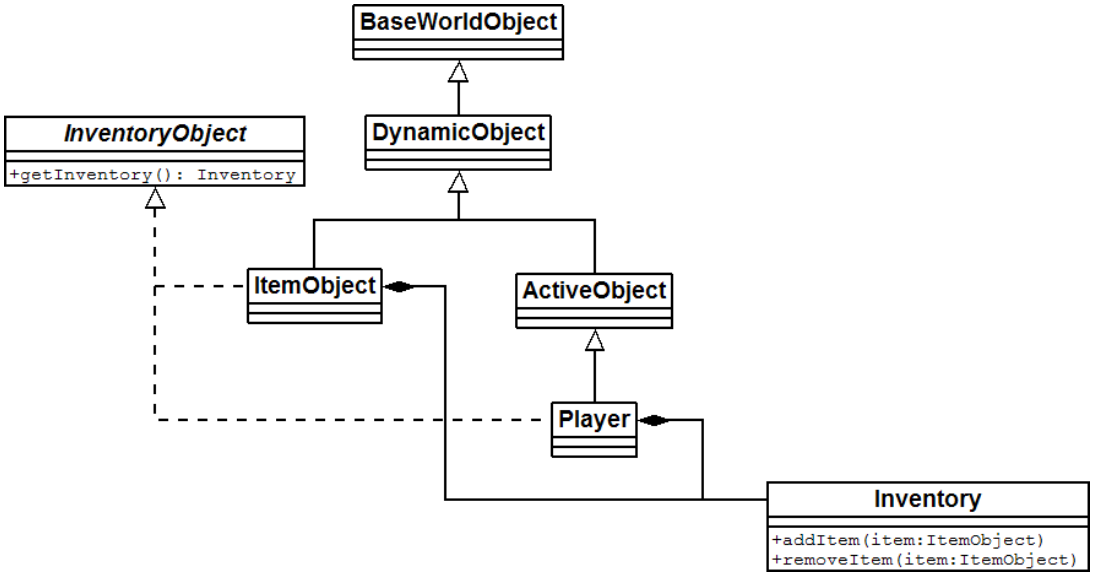


Figure 5–1: Mammoth Architecture for DynamicObjects Before

When directly modifying an inventory object, the parent object has no mechanism to monitor or be notified that the state of the inventory it contains has changed (outside of registering a listener, which seems unconventional). This causes complicated situations to arise when a client retrieves a remote copy of an inventory and changes it locally (by retrieving the inventory out of its parent object) without notifying the original server-side inventory object.

To combat this problem, the common shared interface between players and container items was extracted to their shared parent class (see Figure 5–2). In addition the ability to retrieve the inventory object directly was removed and instead replaced with such functions as `getItemsInInventory()`, `getInventorySize()`, `addItem()` and `removeItem()`.

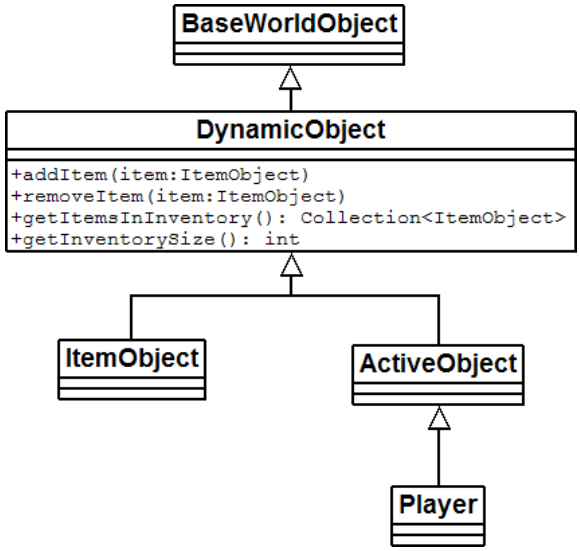


Figure 5–2: Mammoth Architecture for DynamicObjects After

In this manner, whenever such calls were made to modify the inventory, they could be executed using the Mammoth RPC system, and the owner of the inventory would be directly involved in its manipulation.

The main reason behind this change (in addition to the consistency issues) was to allow subgames to monitor events such as adding and removing items from inventories.

5.4 Action Controller

The *Action Controller* provides a centralised location for the control and execution of game mechanic actions (as was previously discussed in Section 4.5.3). It also provides a platform for subgames to "hook in" and intercept or change the default game behaviors of the global world. It also allows subgames to create new types of

actions for their unique purposes. It was also a necessity for Mammoth when it migrated from the traditional client/server message system to a distributed multi-server environment with replicated objects and remote procedure calls.

The action controller itself is an object which resides on every machine in the network. It is created upon launching the application, and instilled with basic knowledge to allow the game to function (see default behaviors below). It's main purpose is as to act as a switchboard by taking in a type of action (such as pick up or drop) and find the matching behavior to execute locally or remotely. The action controller class diagram can be seen in Figure 5–3.

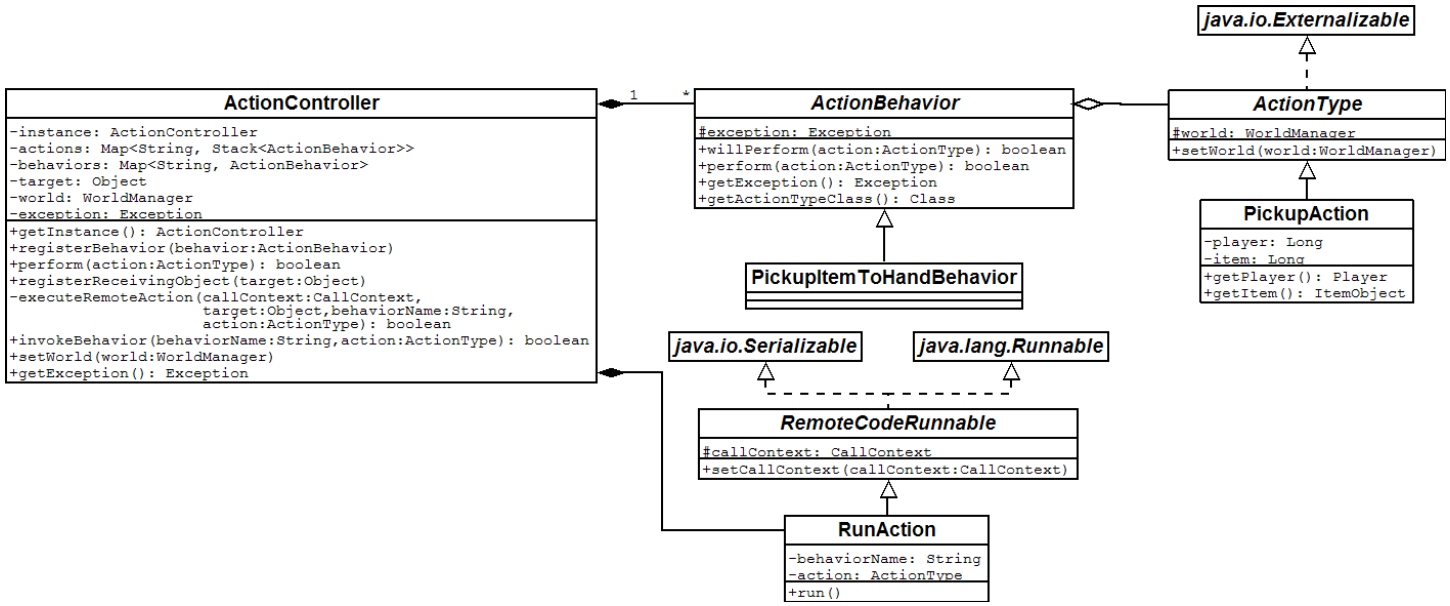


Figure 5–3: Action Controller Class Diagram

5.4.1 Actions

With the new system, any change to the game state has to be done from within an *action*. Most actions are instigated by a player. Typical actions in the world are picking up an object in the world, placing an object in a player's inventory, removing the object from a player's inventory, and placing the object back in the world.

The notion of an "Action," actually corresponds to two separate entities, each with a respective segregation. The first part of the action is what defines its basic game mechanic and is known as the *Action Type*, it corresponds to move, pick up, drop, get or put in most cases in Mammoth. It is instanced to form an *action instance* which holds information about the objects involved in the overall action.

The second part of an action is the *Action Behavior*. Its job is to actually execute the required game mechanics behind the action. It is broken down into two parts: a check for the preconditions of the action (e.g. does the player have enough room in their inventory?) and the actual execution of the action. This distinction between check and execution is important later for remotely executed actions (see Section 5.4.2).

Action Instances

As described above, an *action type* is a category of action (e.g. move, pick up, drop, get, or put). When a player wishes to perform one of those actions, they create an instance of the action type to create an *action instance*. The action instance then holds the type of action to be performed and the objects involved in the action (e.g. the player and item in the case of pick up).

This action instance is then passed to the action controller, which finds a suitable action behavior to actually execute the request.

Action Behaviors

An *Action Behavior* is what actually defines how an action is performed. There is at least one action behavior for each action type. Default behaviors are loaded at the start of any client or server to provide the default actions in the world such as move, pick up, etc...

An action behavior has two main methods `checkPrecondition()` and `perform()` each of which take a corresponding action instance as an argument. When an action instance is passed into the action controller, each action behavior corresponding to it is polled with its `checkPrecondition()` method. The action behavior then can check if it would like to perform its action, i.e. in the case of the default pick up behavior, can the player actually pick up the item. If he can, it returns `true`, otherwise it sets its exception attribute and returns `false`.

The action controller processes behaviors in a "lowest subgame first" order, i.e. the default behavior is processed last. Therefore, if no behavior returns true on its `checkPrecondition()` the action cannot be performed, and the exception from the behavior will be available to be polled by the client to see why the action failed. This is typically displayed in the chat box to the user. e.g. "Item out of reach." or "Item is not takable."

As soon as a `checkPrecondition()` method returns true, the action controller takes control again and performs a *Remote Action* call (see the following subsection). In the rare case where the Mammoth client is running independent of a network

(stand-alone), the behavior's `perform()` method is called locally, which changes the state of the player and the item in the pick up case. More on the stand-alone client is described in Appendix A.3.

5.4.2 Remote Actions

As discussed in Section 4.5.1, actions involving more than one object need to use a locking scheme such as TTLS (see Section 4.5.2) in order to provide consistent updates to the distributed game state. Remote locking, i.e. asking for a lock on a remote node, should however be avoided whenever possible for obvious reasons: remote calls take time, and therefore the objects involved in the action would have to be locked for a long time. Also, in case of node failures, locks might be held even longer.

In order to minimize locking, it would be ideal to initiate an *action* on a node that holds the master object of one of the objects involved in the action (often the node holding the player master object). Then, for a two-object action, the only wait would be for the second item. In the case of node failures, only the player node will have waited, but since he is initiating the action, chances are he is not wanting to do anything else.

Therefore, it would be beneficial to have the functionality to execute the selected behavior above on the action controller of whatever machine the player master is located on. To do this, object proxies received a special method `execute()` which accepts a special "Java Runnable" to allow for the remote execution of these action code fragments. Via this method, the action controller can send the name of the behavior which it wishes to be executed and the action instance containing the

object ids of the objects involved in the action i.e. the player and item for pick up, to the master node of the player.

It is also the reason why action behaviors are broken into two components, the check and the execution. By checking preconditions of an action with the local replicas, if they fail (e.g. the player is out of reach of the item they wish to pick up) then an expensive remote network call can be avoided and network bandwidth has been saved. Otherwise, if everything checks out, the network call can be made and the action can be performed remotely using the second method on the action behavior. Of course, before executing the method on the remote node, the check can be performed again, since the master object state might have changed in the meantime; however, this usually results in a versioning error (see Section 4.5.1).

Thus, via the built-in RPC system, a remote call is made to the player master object. It then executes the special wrapper created by the action controller on the client. This wrapper, will execute a special `invokeBehavior()` method on the remote machine's action controller. The `invokeBehavior()` method will take the behavior name, find it in its local behavior list and pass in the reconstructed action instance to its `perform()` method. Then the action will execute as normal; however, on the remote machine of the player master instead of the originating client (see Figure 5–4).

This process saves network bandwidth and time compared to traditional locking techniques as discussed in more detail back in Section 4.5.2.

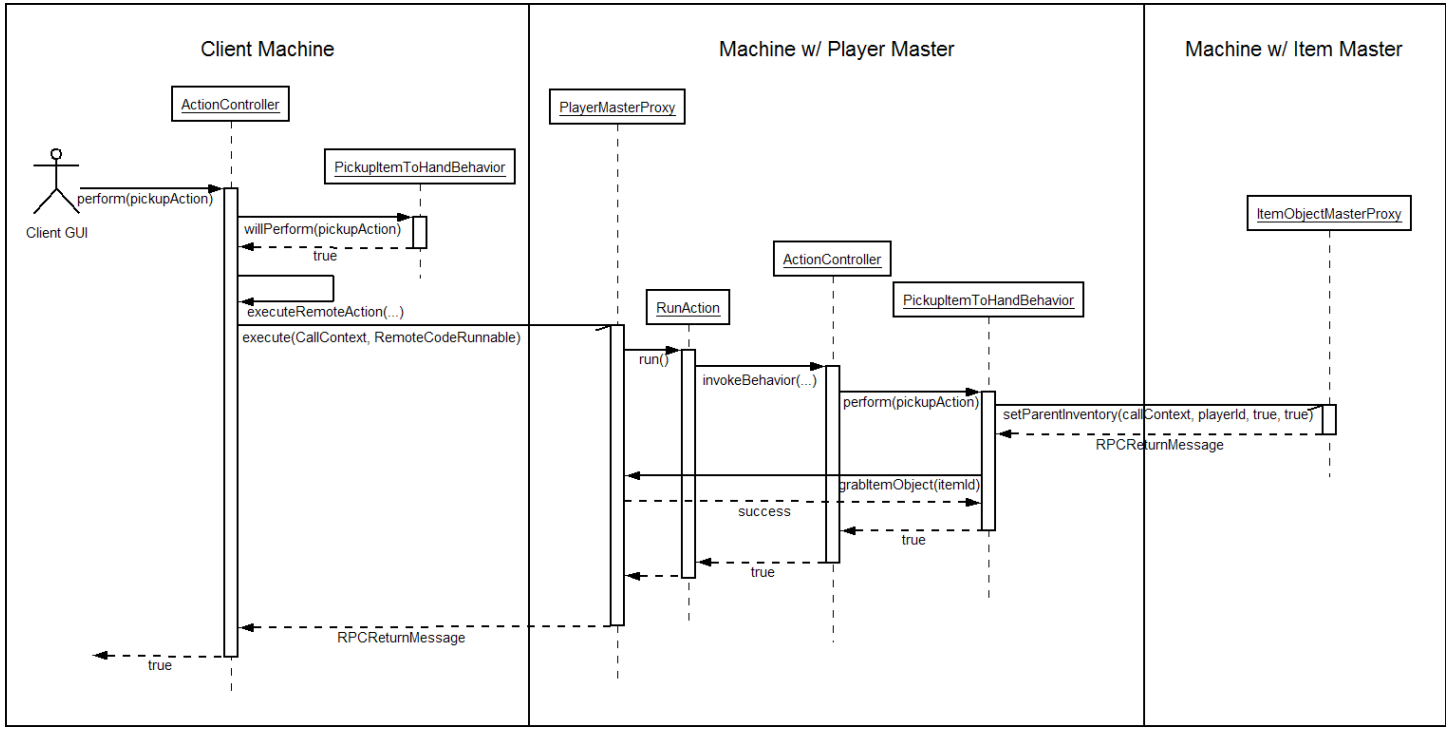


Figure 5–4: Action Controller Sequence Diagram

5.4.3 Actions and Subgames

The client automatically creates an action controller and automatically loads in the default game actions when it is started. Afterwards, the subgame manager loads every subgame as specified in the properties file and they in turn load actions they wish to modify the behavior of. These are loaded on the action controller's stack for each action type.

When a player initiates an action, the subgame action behaviors are queried first before relying back on the default world behavior. In some instances the subgame may subclass the default behavior to provide extended functionality, or merely add hooks to the subgame to be notified about special events, for instance the player placing an object in a container.

It is also important as there could be cases where even on a node where the player is *not* playing a subgame actions still need to be processed using the subgame behaviors. For instance, maybe people not playing capture-the-flag can see the flag, but are not allowed to pick it up. Or a trophy to be found in "Find the Trophy," can be picked up by anyone, but for a limited time before they are forced to drop it automatically.

5.5 Subgames and Visibility

There could be instances where subgames would want to hide important game items (such as flags in capture-the-flag) from players who are not playing that particular subgame. Providing this functionality is possible on the server-side via *interest management* or on the client-side via *visibility management*.

Since the Mammoth project implements both of these services, it could be done in either location. It was decided at the present time, the *Visibility Manager* on the client would be sufficient as it alleviates work from the server.

Like most other Mammoth components the visibility manager is modular and can be switched, for instance for debugging purposes, with various implementations. One example for an implementation displays everything in the game world regardless of its innate visibility. It is also possible to extend the visibility manager with specific rules for active subgames in this manner.

More information about the implementation of the visibility manager created is available in Appendix A.2.

5.5.1 Preventing Multiple Concurrent Player Actions

One thing which was observed back in the original client/server version of Mammoth was that users would not sit patiently waiting for things to occur. They would create many simultaneous requests at once. Therefore, a system needed to be implemented in order to "lock" the client and prevent the user to create any further actions while their initial request was still being performed. This cycle involved locking the Graphical User Interface (GUI), and then waiting for the action's return value.

However, with the new replicated objects system and the action controller, this notion of a client-side lock needed to be implemented differently. This implementation, however, was in the end a lot more elegant and localized in the action controller code, and did not pollute the graphical user interface with hooks in many places as done in the previous implementation.

Instead of checking for locks in all locations interested in performing certain actions, a single lock was placed on the request to perform an action. If a client is already executing an action, the lock is not granted. The request to perform an action is ignored, and a friendly reminder is sent to the user, letting him know about the action failure. In addition, the lock on the perform action changes the mouse cursor to signal to the user that they must wait for an action to complete. Usually, if the network connection is fast enough, this "wait" cursor is not seen. Once the action has been performed and the updates received (determined by checking for the corresponding nounces (see Section 4.5.2)), the action completes, and the client unlocks.

Chapter 6
Conclusion

Massively Multiplayer Online Games (MMOGs) are built from much more complex and dynamic systems than those found in other computer games. They require a complicated harmony of different systems working together in order to function. With more people playing these social games, year after year, the economic impact will continue to have great potential in the future.

The motivation behind this thesis was to break down a MMOG into smaller more manageable subgames to allow for increased and more flexible interaction between players while being able to increase the scalability of the system. Subgames also provide the ability to modify and enhance the default game mechanics. In addition, game architecture and design can be greatly stabilised through maintaining strict consistency of the game state, especially in a distributed game environment.

Subgames provide a stable, scalable framework on which to build upon a virtual world in the complex environment of MMOGs. As it was described in Section 4.2 there are a variety of requirements needed for subgames to function, such as the ability to create new objects and object types, modify the user interface, create new and modify existing game actions, and exist in a modular framework.

With such a platform now in place, subgames provide a vast number of options to expand upon the global game world and create interesting dynamics for players

to experience. It was also shown how a consistent game state in a Distributed Multi-Server Environment (DMSE) is also achievable with minimal overhead through the use of a light-weight transactional protocol named TTLS. In addition, the mechanisms used were also suitable for subgames. By encapsulating basic game rules and game state changes in *actions*, a more robust system was created.

In addition to presenting the theoretical ideas behind subgames, this thesis also provides an initial implementation of the subgame concept within Mammoth, the massively multiplayer game research framework developed at McGill University. The core of the implementation consists of a subgame manager component that runs on every machine. It loads the available subgames at start-up, keeps track of the individual subgame instances that are currently active, stores subgame-related state, and manages the participating players. Subgames can also define new game objects by defining object types using XML. Finally, action types and action behaviors allow a subgame to create new game actions, or alter the existing behavior for default actions in the game world in a consistent and scalable fashion.

6.1 Future Work

The work in this thesis represents a first step. The research accomplished has defined the concept of subgames and their inherent differences to shards in a manner of theory and practical implementation. Further work can be achieved as a second step to further conceptualize subgames into a discrete form. The work which has been provided can also be used as a basis to proceed to more in-depth studies of how subgames are supposed to interact on a larger scale.

In this thesis, we showed ideas and techniques that make it possible for *one subgame instance* to define new objects, to define new behaviors, or to override the objects and behaviors of the global world it inhabits. The ideas and techniques, however, did not address the issues which arise from concurrently running subgame instances which specify conflicting alterations to the global world. As mentioned in Section 4.1, subgames can form hierarchies. Also, players can be allowed the freedom to participate in several subgame instances concurrently. In this case, it is necessary to look at the possible conflicts between subgame instances which a player participates in. Finally, the graphical user interface of the Mammoth client must be customizable in order for subgame players to initiate subgame-specific actions. These issues are described briefly in the following sections.

6.1.1 Conflicts Between Different Subgames

Using the techniques proposed in this thesis, subgames can alter the state of the global world and redefine how players interact with objects. For instance, a game such as Orbius (see Section 4.3) has to limit the amount of weight a player can carry in order to make it more difficult for a team to collect the orbs of different weights in order to win. It is also possible for a subgame to remove items from the virtual world. For instance, "Find the Trophy" (see Section 4.3) might want to make sure that only one single trophy exists in the world.

What if these alterations to the properties of the virtual world conflict? What if, for instance, one subgame forbids the picking up of objects above a certain weight, while some other subgame requires heavy objects to be picked up in order to win?

Can these two subgames co-exist in the same virtual world? What if a player wants to participate in both subgames at once? Should this be allowed?

There are many issues related to these problems. First of all, conflicts between subgames have to be detected. This is not trivial, since subgames can extend the virtual world in many ways. Probably, some extensible framework or language will have to be defined which allows subgame designers to express the subgame properties and expectations in a formal way. A pairwise analysis of formal subgame definitions could then reveal potential conflicts.

Once a conflict is detected, it has to be resolved. It would be possible to simply forbid the conflicting subgames to be instantiated at the same time. It can also be imagined that properties of the virtual world are only changed for players joining a subgame. As a result, non-subgame players and subgame players would see and experience the same virtual world in different ways. At first glance, this seems to solve conflicts, provided that a player is not allowed to participate in conflicting subgame instances simultaneously. However, further studies have to reveal if this idea would not create awkward situations and unfair gameplay situations between players.

6.1.2 Interactions Between Players Participating in Subgames

It will be important to study how different instances of the same subgame interact. Many questions arise in this context. Ideally, players participating in one instance should be able to see players participating in the other instance.

As an example, imagine two games of soccer taking place concurrently. A soccer player from the first game should be able to see the players of the other game, if

the games take place in the same virtual world. However, would it be allowed for a player from the first game to take the ball away from the second game? Probably not. If the goal was to make the virtual world more realistic, it could be allowed; unfortunately, a game that is realistic, but not fun, does not serve its purpose.

Therefore, if non-subgame players were allowed to interact with vital objects important to subgame players then this interaction would have to be monitored and artificially stopped, if the gameplay of the subgame is threatened. For instance, in "Find the Trophy," a non-subgame player could take the trophy, but would be forced to drop it somewhere in the near future, otherwise the subgame players could potentially never be able to find the trophy again, especially if the non-subgame player leaves the game.

6.1.3 User Interface Modification

Another challenge which arises in a customisable game framework is modifications to the user interface. If a game creates a new type of interaction, such as the tickling action of Orbius, players need a way of initiating the new action. It is helpful to players to have a visual representation of the new action as well.

There are further challenges in coordinating multiple combinations of actions, as well as providing the needed hooks into the user interface framework. These hooks not only have to provide the modification of the existing user interface, but allow for multiple subgames to add their own modifications in harmony.

As mentioned previously, complex modifications to the user interface are a necessity for complex subgame mechanics. Especially for a variety of subgame types. More sports like games such as capture-the-flag, may not only need special actions

like “designate object as base,” but also windows to display the score. A subgame used to manage side-quests would also need an additional but interactive window interface.

While basic additions are currently possible within the GUI of the Mammoth client, a system would need to be designed to allow these additions to coexist simultaneously.

Appendix A
Appendix

The following sections found in this appendix are addendum to the work implemented in the Mammoth project during the duration of this thesis. The first section revolves around work done on consistency in a single-server environment before Mammoth had distributed servers. The middle two sections relate to implementation issues discovered while working with Remote Procedure Calls (RPC) in a Distributed Multi-Server Environment (DMSE). The last section provides information about work done on moving Mammoth from a 2D to a 3D world.

A.1 Single Server Consistency

Initially, the Mammoth project was built upon a traditional client/server network topology. Combating consistency on the centralised server was a first step towards understanding consistency in general. Unfortunately, the solution presented here, while sufficient at the time, was a short-term solution which did not scale to a DMSE.

When a request is made on a single server to update the location of an item, the server already has access to both objects. With a single thread, proper synchronization, or locking, the server can "simultaneously" update both objects at once. This avoids any problems with consistency arising from the need to update two objects at once as was discussed in Section 4.4.1, on the server side.

In a single server instance, invalid requests can easily be denied; however, it can be difficult to update the state of a client. If an update is missed or the client is lagging it can try and apply a regular update message inappropriately.

By consolidating the request for the item location change into a single notion of "the object is now here" conflicts can be avoided. The client can then move the object from where ever it thought it was to where it should be (by updating both objects at once) restoring consistency to the client.

If a client does not have a copy of the item, then they can ignore the message, as when they receive the item its state will be correct as it will be coming from the single instance on the server.

Unfortunately, when there are multiple objects on multiple servers involved, the single point of access and single message approaches no longer work. As was discussed in this thesis in Section 4.5.1, it is not a trivial matter.

A.2 Visibility Manager

Firstly, the *Visibility Manager* (VM) is independent of *interest management*. Its only function is to take all the objects that are of interest to the player on the client and decide if they should be visible or not.

Initially, the visibility manager in Mammoth was simply for ensuring roofs disappeared when a player moved under them. However, there were many problems with it overriding the inherit visibility of a game object instead of just the visual graphical view. These problems continued to crop up as other parts of Mammoth

changed leading to confusing erroneous displays, such as no other players being display as they were turned invisible by the visibility manager or objects, that should be invisible, reappearing when other clients logged in.

With the introduction of object replication and the change in behavior to how Mammoth works especially in regards to picking up and dropping items. It was finally time to rewrite the original visibility manager. The goal of the rewrite was to cleanup listener registration and the initialization of the graphical views of objects. As visibility was basically being managed in two places, the old visibility manager and the graphical view of the object, which causes plenty of room for confusion and conflicting results.

The other and probably main reason for the refactor was the `setVisible()` method was now a Remote Procedure Call (RPC). And if an object was trying to change its visibility it would make a call to the master of the object instead of the local copy of the object. This caused many confusing problems (initially their was a problem with changing the position of an object like this too), such as roofs disappearing on other clients that were not supposed to disappear. Basically, the visibility of the graphical view had to be notified when the visibility property of the object was changed on the importing of the remote state. Therefore, the new visibility manager framework was created to combat all these issues.

The visibility manager is defined as an interface, but a base class is provided which defines the general functions of registration of objects with the manager. The base class should probably be used for any further implementations, but many example visibility managers have been created for reference.

Whenever an object of interest comes into or is removed from the (local) world state, the visibility manager is notified. It is also notified when the visibility of an object changes (an item being picked up) or the player moves positions. On these two main events, code can be created to control what the player should see from their current position or override the inherit visibility of an object. i.e. one could create a visibility manager that shows every object regardless of whether it should be visible or not.

Item visibility was also changed to be based solely upon information on the location of the item (world, hand, or inventory); however, this initially did not flag the visibility as being changed when the item changed locations. However, simply calling the visibility changed notification function on the object on a location change solved the problem, as then the visibility manager would recheck the visibility property of the item and correctly update itself.

The visibility manager is created for the *World Window* which is the main view on the game world in the Mammoth Graphical User Interface (GUI). The VM gets notified by the world window whenever an object is added or removed from the game. It is also alerted by the world window when the main player (the character which is being controlled) on the client is registered.

When the VM is notified of these additions it adds the object to its internal list of objects and registers a listener with it so it will be notified when the object's visibility changes or in the case of a player, when they move. When an object's visibility changes the visibility manager's `updateVisibility()` method is called to coordinate the graphical change.

There are many different example visibility managers at the present time. A list of them follows along with a short description.

A.2.1 Roof VM

This is the default visibility manager. It behaves as the game should. When a player is under a "roof" it disappears. Otherwise, all visibilities on the client should correspond to their internal representation.

A.2.2 NoRoofs VM

This is similar to the default VM above; however, roofs are never displayed.

A.2.3 All VM

This VM always displays every object in the *interest area* of the client's player regardless of its inherit visibility property. i.e. if an item is picked up or placed in an inventory, it will still be visible on the ground at whatever position it was initially at. Normally, the item disappears when it is grabbed thus preventing further interaction with it.

While this Visibility Manager may initially seem useless, it is in fact quite useful for debugging purposes. Since, normally it is impossible to simulate simultaneous interactions with items, testing network code is difficult. However, if the item does not disappear from the game world when picked up with this VM, it is still possible to interact with it. Therefore a simultaneous request can be mimicked easily without the need for frantic clicking and luck.

A.2.4 Narrow VM

The Narrow VM was a test just to experiment with a limited range of view and ensure the visibility manager really worked properly. It only shows the player items

that are within their reach which is much smaller than the number of objects they are interested about or that the physical client is responsible for knowing about.

A.3 Stand-Alone Client

One thing that has existed in Mammoth since the beginning was the notion of a "Stand-Alone" client. Basically, it was the graphical interface to the game world without any network engine or server behind it. It allowed for a simple way to test and debug code without the need to setup a server and connect to it, especially when only game interface or other miscellaneous code was changed. It has been an invaluable debugging tool.

Unfortunately, with the move to object replication with RPC, the once simple stand-alone client was broken. Originally, there was a game object which handled changes to the game state. The game object was wrapped by the network layer using simple network messages for every change. Later this lead to the problems and the creation of the single update message discussed in Appendix A.1. To create the stand-alone client all which needed to be done was to create a dummy wrapper which would interface with the client-side GUI instead and direct all calls to a local copy of the game object rather than create a network message, send it, and wait for a response.

The best way to understand is to follow the chain of an action in two cases: the regular network client and the stand-alone client. On a networked client, the player would click to pick up an object. That click would be then turned into a network message requesting the pick up action to be performed and sent over the network to the server where all the objects reside. The server would perform the action and

change the state of its master copy of the object and then send an update message back to all clients interested in the objects being changed. The client would receive this update message and make a call to import the state to update the local copy of the object. The resulting call would also trigger notifications to listeners monitoring the object, such as the graphical representation and the player would see themselves picking up the object.

On the stand-alone client, this procedure was relatively the same; however, no network message is created and the action is performed directly on the local copy without delay. Thus, the player clicks on the object, the game grants permission for the action to be performed, and the objects are updated immediately calling all the listeners in the process. And thus, the player sees themselves picking up the object.

When RPC is introduced with object replication this basic network process changes. Instead of a network message being created for the specified action, a network message is created for the RPC invocation transparently through a replica proxy. And instead of an update message specific to the action, a replication update method containing the changed state is returned. Thus, when the state is changed via RPC, no listeners are notified as the call would never be performed on the local client. Instead they must be moved to where the state update occurs locally when the new replication update message is received.

However, if the same methodology of the original stand-alone client is applied, the process breaks down. As now the stand-alone client has all the original objects without proxies. When the player goes to pick up the object, the calls are made directly and the objects are changed; however, now no listeners are called as the

import state method (where the listeners are notified) is never called. Thus, on the stand-alone client, the graphics layer is never notified that the object has moved to the player's hand and the player does not see the change even though it has occurred internally.

Fortunately, the solution to this problem also brought to light another issue. Imagine, if while using the object replication scheme, a master object resided on another client or a server with a monitoring system interested in certain objects. Normally, when the master object's state is changed via a RPC invocation, the state is changed, but similar to the stand-alone client issue, no import state method is called and no listeners are notified of the change. Thus, anything monitoring the master object is never alerted to the change. Therefore, even on the master it was important that listeners be triggered via the import state method. With this addition, if all the stand-alone objects were changed to be master proxies (instead of the direct access to the objects), whenever calls were made, not only would the state change, but the listeners would be called as well. Therefore, two problems were solved with one solution.

The only issue left to solve for the stand-alone client was that by using a master proxy, a replication update message would be sent whenever an object's "remote" methods were called. Yet, the stand-alone client had no notion of messages or networking as it is an independent *stand-alone* client. Therefore, the original "Replication Engine" handling the network messaging needed to be abstracted. In this manner, two replication engines could be created. One would be the original networked replication engine, where the other would be a "dummy" one not performing

any actions. Thus, the dummy replication engine could be passed to the master proxies on the stand-alone client, and while a network replication update request would be made, nothing would occur.

Thus, with the increased flexibility and ease-of-use of the new RPC and object replication systems, the stand-alone client system that has been of use since the beginning of the project was able to stay intact.

A.4 2.5D Demo

Initially, the world of Mammoth was in two dimensions. A 3D demo was created, but separate from the research work done in 2D. In order to maintain the research done in 2D while moving to a 3D world, the notion of the 2.5D demo arose. As Mammoth was running with a 3D graphics engine behind it (JMonkey), it was possible to project the 2D textures of Mammoth into 3D objects. This allowed the client to seem 3D while the game and servers behind it still worked within 2D space (see Figure A–1). This allowed Mammoth to move from its initial 2D roots to a more updated modern look in 3D.

There were some interesting challenges related to the move to 3D. One main challenge was simple "picking" (or determining which object was clicked on). Normally, this is quite easy in a 2D environment, but not quite as simple in 3D. Thankfully, the 3D engine used could return sets of graphical objects that a ray intersected with. This ray could be cast out from the camera to where a player clicked and once each graphic object was tagged with its equivalent world object id, this problem was solved.

Figure A–1: 2.5D Client

The camera itself provided a lot of new challenges as well. It had to be ensured that the camera would always be "right-side" up, so the player did not see the world of Mammoth upside-down. In addition, as the camera moved (through player control), the player's character's animation needed to be adjusted i.e. if the player was moving towards the camera the character's front image needed to be shown. This involved some complex approximations of direction in regards to wherever the camera was currently located.

Besides the initial challenges of moving from a 2D to 3D world, there were spacial challenges as well. Normally, in the game, a player would move into a building and the building's roof would disappear (as per the *visibility manager*). However, in a 3D world, it would be possible to have the building or a wall blocking the view of the player without the player being inside the building. For this, a ray was used again from the camera to the player's character. With this set of objects, it was possible to determine if anything was before the player's character blocking its view. If such an object existed, it was made translucent so the player's character could be seen through it.

In addition to simply projecting 2D textures into 3D boxes for walls and roofs, the ability to load actual 3D models in place of their 2D equivalents was needed. By adding some additional XML parameters, it was possible to load a 3D model in place of the 2D texture and scale it to fit within the 2D dimensions specified for the object being replaced.

Finally, as an experiment, a "first-person" viewpoint was added. Giving a player the ability to see through the eyes of their character instead of seeing the world from above in the "third-person" (as in Figure A–1). In this perspective the default radii for player interest and reach were not sufficiently distant enough compared to the two-dimensional version of the game. This was because, in three-dimensions a player's field of view is much greater than it is in a two-dimensional game. Also, it was found the focal-point of the camera was too far-away for a player to see items at their feet, therefore it needed to be possible for a player to reach and pick up an

item that was much farther away than before as otherwise it would disappear from their view.

There were many technical challenges to move Mammoth into a 3D world, but it was still possible to maintain the 2D research that had been done up onto this point. This has provided many benefits to allow the project to move forward and start exploring new 3D challenges, while still maintaining the large amount of work that has already been done.

References

[1] Realms f.a.q. http://www.worldofwarcraft.com/info/faq/realms.html, 2004.

[2] Frequently asked questions: Bittorrent distribution. http://www.blizzard.co.uk/wow/faq/bittorrent.shtml, 2005.

[3] Ncsoft's guild wars breaks four million. http://www.arena.net/press/pr-070821.php, August 2007.

[4] The9 limited reports first quarter 2007 unaudited financial results. http://www.the9.com/en/news/2007/news_070523.htm, May 2007.

[5] Blizzard entertainment's world of warcraft: The burning crusade surpasses one million peak concurrent player milestone in mainland china. http://www.corp.the9.com/news/2008/news_080411.htm, April 2008.

[6] Eve online puts the mmo in steam. http://www.eve-online.com/pressreleases/default.asp?pressReleaseID=43, January 2008.

[7] Mammoth - introduction. http://mammoth.cs.mcgill.ca, 2008.

[8] World of warcraft reaches new milestone: 10 million subscribers. http://www.blizzard.com/us/press/080122.html, January 2008.

[9] Nathaniel E. Baughman and Brian Neil Levine. Cheat-proof playout for centralized and distributed online games. In *INFOCOM*, pages 104–113, 2001.

[10] Ashwin R. Bharambe, Jeff Pang, and Srinivasan Seshan. A distributed architecture for interactive multiplayer games. Technical report, School of Computer Science, Carnegie Mellon University, January 2005.

[11] Jean-Sébastien Boulanger. Interest management for massively multiplayer games. Master's thesis, McGill University, Montréal, Canada, aug 2006.

[12] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *NetGames 2006: 5th Workshop on Network & System Support for Games*, pages 1–12, Singapore, oct 2006.

[13] Fabio Reis Cecin, Claudio Fernando Resin Geyer, Solon Rabello, and Jorge Luis Victoria Barbosa. A peer-to-peer simulation technique for instanced massively multiplayer games. *Distributed Simulation and Real-Time Applications, 2006. DS-RT'06. Tenth IEEE International Symposium on*, pages 43–50, Oct. 2006.

[14] E. Cronin, B. Filstrup, and A. Kurc. A distributed multiplayer game server system. `http://citeseer.ist.psu.edu/cronin01distributed.html`, May 2001.

[15] E. Cronin, B. Filstrup, A. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *Proc. of NetGames 2002*.

[16] Ta Nguyen Binh Duong and Suiping Zhou. A dynamic load sharing algorithm for massively multiplayer online games. *Networks, 2003. ICON2003. The 11th IEEE International Conference on*, pages 131–136, Sept.-1 Oct. 2003.

[17] Stefano Ferretti and Marco Roccetti. Fast delivery of game events with an optimistic synchronization mechanism in massive multiplayer online games. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 405–412, New York, NY, USA, 2005. ACM.

[18] Robert D. S. Fletcher. Consistency maintenance for multiplayer video games. Master's thesis, Queen's University, January 2008.

[19] Adrian Ghizaru. Training non-player characters, January 2008. Masters Project, McGill University, Montreal, Canada.

[20] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. `http://citeseer.ist.psu.edu/knutsson04peertopeer.html`, March 2004.

[21] Marc Lanctot, Nicolas Ng Man Sun, and Clark Verbrugge. Path-finding for large scale multiplayer computer games. Technical Report GR@M 2006-2, GR@M, McGill University, July 2006.

[22] Frederick W.B. Li, Lewis W.F. Li, and Rynson W.H. Lau. Supporting continuous consistency in multiplayer online games. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 388–391, New York, NY, USA, 2004. ACM.

[23] Dingliang Liang and Paul Boustead. Using local lag and timewarp to improve performance for real life multi-player online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 37, New York, NY, USA, 2006. ACM.

[24] Martin Mauve. How to keep a dead man from shooting. In *IDMS '00: Proceedings of the 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 199–204, London, UK, 2000. Springer-Verlag.

[25] Martin Mauve, Jürgen Vogel, Volker Hilt, and Wolfgang Effelsberg. Local-lag and timewarp: Providing consistency for replicated continuous applications. `http://ieeexplore.ieee.org/iel5/6046/28207/01261886.pdf`.

[26] Roger Delano Paul McFarlane. Network software architecture for real-time massively-multiplayer online games. Master's thesis, McGill University, 2005.

[27] Lothar Pantel and Lars C. Wolf. On the suitability of dead reckoning schemes for games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 79–84, New York, NY, USA, 2002. ACM.

[28] Quazal. *Duplication Spaces^{TM} Quazal Multiplayer Connectivity White Paper*, January 2002.

[29] Georgios Theodoropoulos School. A unified framework for interest management and dynamic load balancing in distributed simulation. `http://citeseer.ist.psu.edu/388638.html`.