Persistent Transaction Models for Massively Multiplayer Online Games

Kaiwen Zhang

Master of Science

School of Computer Science

McGill University

Montréal, Québec

June 2010

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

I dedicate this thesis to my parents who have supported me from the beginning.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Bettina Kemme for her guidance and time. She has helped me for the last four years, including my undergraduate studies.

I would also like to thank the other professors involved in the Mammoth project, most specifically Jörg Kienzle and Clark Verbrugge. Their support during that same time frame of four years has been invaluable.

Finally, I would like to thank the rest of the students who have worked on Mammoth, most specifically Alexandre Denault. This is truly a team effort, and without the work put forth by others, this thesis would not have not been possible.

ABSTRACT

Massively Multiplayer Online Games (MMOGs) can be treated as a database application. Players request actions concurrently to alter the state of objects in the game. Since the world state is the most valuable asset of MMOGs, it is extremely important to ensure its *consistency*. On the other hand, the defining feature of such games is their capacity to support thousands of clients playing simultaneously, thus requiring *scalability*.

This thesis proposes a solution which leverages typical game semantics and architectures to design scalable *transaction models* for action handling while maintaining the required levels of consistency. These models vary in their levels of isolation and atomicity and offer different consistency guarantees that are suitable for actions of varying importance and complexity. Action handling protocols are then designed according to those models and optimized for scalability and efficiency.

We also present a persistence architecture which is integrated with the transaction models mentioned above. We show how the different consistency guarantees of each transaction model can be maintained by the persistence structure.

Concrete actions are then implemented and designed using various transaction models with persistence support. We then evaluate and compare the performance of the various implementations and discuss the trade-off between performance and consistency.

ABRÉGÉ

Les jeux en ligne massivement multijoueur (MMOGs) peuvent être considérés comme des applications base de données. Les joueurs initient des actions de façon concurrentielle pour modifier l'état du jeu. Puisque l'état du monde est le plus grand atout des MMOGs, il est extrêmement important d'assurer sa *consistance*. D'un autre côté, la caractéristique essentielle de ces jeux est leur capacité de supporter plusieurs milliers de clients simultanément, et donc l'habileté de *gérer une charge grandissante*.

Cette thèse propose une solution qui est fondée sur des sémantiques et architectures typiques aux jeux pour concevoir des modèles de transaction extensibles à la charge pour la gestion des actions tout en maintenant les niveaux requis de consistance. Ces modèles varient dans leurs niveaux d'isolation et d'atomicité et offrent donc des garanties de consistance variées qui sont adaptées à des actions d'importance et de complexité différente. Des protocoles de gestion des actions optimisés sont alors conçus selon ces modèles.

Nous présentons aussi une architecture pour la gestion de persistance des données qui est intégrée aux modèles de transaction mentionnés ci-dessus. Nous montrons comment les garanties de consistance de chaque modèle sont maintenues par la structure persistante.

Des actions concrètes sont alors mises en œuvre et conçues selon les divers modèles de transaction avec persistance. Nous évaluons et comparons la performance de

chacune des implémentations et discutons du compromis entre la performance et la consistance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

MMOGs or MMORPGs, short for Massively Multiplayer Online (Role-Playing) Games, is a genre of online games which emerged in 1997 with Ultima Online. Since the start of the 21st century, MMOGs have seen tremendous growth and annual revenues now exceed $1.4 billion [11]. These games typically revolve around creating large virtual worlds shared by the players. Each player controls their own character (also called avatar) and interacts with others and the environment through a graphical interface. One unique aspect of MMOGs is persistence of game state. Persistence refers to the fact that characters' data spans multiple play sessions and can be measured in months or years. In other words, a character's data is being read and updated every time it is being played. Furthermore, the virtual world itself is constantly evolving whether players are currently logged in or not. The main appeal of MMOGs is thus character progression: by accomplishing certain tasks (quests), a player can improve its abilities and acquire more items (to be stored in a personal inventory).

Not only are MMOGs considered products but they are also services. Customers typically have to subscribe in order to play by paying a monthly fee (called "pay-to-play"). The game company is then expected to satisfy the following requirements:

- **Availability** : Players must be able to connect to the game and access their data at any time. Game performance must be acceptable. Any downtime should be scheduled.
- **Maintenance** : The game software must be maintained and new content should be added to the game to keep players interested.
- **Correctness** : The game software should be bug-free and working. The player should be able to play the game as intended.
- **Security** : Clients' personal data should be kept safe and secure.

World of Warcraft, the most popular MMOG currently on the market, has more than 11.5 millions paying subscribers [12], each paying on average $15 a month. MMOGs are therefore unique in gaming in the sense that development is required well after release and has a direct impact on the income of the game.

Challenges. In particular, client data is an important concern. Since the primary objective of players is to develop their character, it is critical to maintain consistency and persistence of the game state. Consistency here refers to the game behaving as intended by the game developers. MMOGs contain numerous non-deterministic events where desirable outcomes are unlikely to occur. This design extends the playtime by forcing players to repeat the same tasks several times until success is achieved. Therefore any data inconsistency or loss of data can be difficult for players to compensate and can result in the loss of several hours or days of progress. To further highlight the importance of game data, a recent trend in MMOGs is the purchase of virtual property through micro-transactions. Some game companies even opt for free access to their games ("free-to-play"). However, players

desiring to progress further in the game must purchase premium content from the company. Such micro-transactions have generated $250 millions for the industry in 2009 in the US [29]. It is therefore even more crucial to ensure consistency and persistence of purchased content.

The other unique aspect of MMOGs is their scale. In addition to creating a large world filled with interactive content, the game must also support a large number of players. The scalability of an MMOG is primarily limited along two dimensions. The first is the content pipeline, which must generate enough content to keep an increasing number of players occupied. The second is the scalability of the system itself, which must support an increasing load. Although a game like World of Warcraft has millions of users, its user base is actually divided into different realms (also called shards). Realms are copies of the same world that operate independently and usually only serve players in a certain geographical region. Typically, characters created on a realm cannot transfer to another one. On average, each realm only has 4000-5000 players connected at any time. Dividing the users is one way of coping with scale limitations, but as games evolve, the number of concurrent players in each world should increase.

From a data management point of view, we can therefore establish the following challenges for MMOGs:

- **Consistency** : Any operation performed on the world state must be consistent according to the semantics of the game.
- **Persistence** : Any operation performed on the world state that was committed and visible to players must be durable.

- **Scalability** : The system must support thousand of simultaneous clients.
- **Performance** : The system must be efficient and deliver responses to the clients within an acceptable delay.

Overview. In this thesis, we will present action models which can satisfy the requirements above. In other words, the models dictate how consistency can be achieved for a MMOG while maintaining scalability and performance. By leveraging game semantics, actions can be categorized according to their complexity and importance. Each category uses a different transactional model which provides a suitable level of consistency. We then show how action handling protocols for each model can be optimized for scalability and performance.

The key observation is that there is a trade-off between consistency and scalability, thus lower consistency models can perform more efficiently. Furthermore, there is a larger volume of simpler actions, such as movement [3, 10], relative to more complex ones, such as trading. Relaxing consistency requirements for frequent actions can significantly improve the performance of the overall system. On the other hand, we cannot completely relax consistency requirements for higher complexity actions since they are more critical to the integrity of the game.

We then present how persistence can be achieved in our solution. The state of the game is monitored in real-time and logged on stable storage for recovery purpose. Essentially, persistence functions according to the action models, where higher complexity actions will be monitored more closely. We demonstrate how our persistence strategies satisfy the consistency requirements of the associated action

models. We also show how the persistence data can be used for fault-tolerance and recovery.

MMOGs now commonly use a distributed architecture with multiple servers for scalability reasons. This introduces more issues, most notably keeping data consistent across all servers. Distribution concerns will be addressed.

Finally, our solution is implemented in Mammoth [24], a Massively Multiplayer Online research framework. Experiments using multiple implementations of actions of varying complexity are used to evaluate the performance of action models and their corresponding protocol.

Contributions. The main contributions of this thesis are:

- A thorough analysis of the execution model and consistency requirements of current MMOGs.
- A set of consistency categories that are useful in the context of MMOG semantics. Each category provides different consistency guarantees suitable for MMOG actions.
- Coordination protocols for each category optimized for performance and scalability.
- Persistence architecture and support for each consistency category and their extension for each protocol.
- Integration of our solution into the Mammoth framework.
- Performance comparison of the protocols and an assessment on the impact of consistency on performance and scalability.

The remainder of this thesis is organized as follow: Chapter 2 presents background information on massively multiplayer online games and their system architecture and other proposed solutions. Chapter 3 details the different consistency categories and transaction models we propose for actions in MMOGs. In Chapter 4, the solution is extended to consider durability and fault-tolerance. Chapter 5 shows details on the process of developing actions and our implementation in Mammoth. The results of our experiments are then discussed in Chapter 6. Finally, Chapter 7 presents some conclusion and outlines future work.

CHAPTER 2
Background and Related Work

As explained in the introduction, the typical structure of MMOGs have clients each controlling a single character, or avatar, in the world through a graphical user interface (similar to Figure 2–1). This game world is populated by various types of objects: characters that are either controlled by players or artificial intelligence, non-interactive objects (e.g. trees, buildings) which form the environment, and mutable items which can be used or interacted with by players. Players let their characters take *actions* which are the basic mean of interaction in MMOGs. This includes moving on the game world, picking up items, dropping them from the character's inventory, and trading with other players. Since the game world is common to all players, the actions executed by one player should be observable by the other players.

2.1 MMOGs as a Database Application

Essentially, we can treat MMOGs as a database application [18]. The world objects are the data, where each object type could be represented by a relational table with various kinds of attributes. Actions are a logical sequence of read and write operations on the attributes of one or more objects, generally requiring the transactional ACID properties:

- Actions require **atomicity** , executing either in their entirety or not at all.

Figure 2–1: Client interface for the Mammoth MMOG

- Different players can request actions on the same objects concurrently, requiring some form of concurrency control to provide **isolation.** In other words, changes made by concurrent actions should not be visible to one another.
- **Durability** is essential as these game worlds run for long periods of time and need to survive various system failures. Upon restart, a game must be restored to the exact state it was immediately before shutdown.
- **Consistency** is defined in terms of game semantics. Assuming that atomicity, isolation and durability are provided, actions that are applied to a consistent

initial state will result in a consistent final state. Game developers must ensure consistency when designing their actions.

However, there are several reasons why MMOGs cannot simply be implemented as a set of transactions using a traditional database system. First of all, most actions are update transactions and the amount of transactions that need to be processed per time unit can often not be handled by a single database server using traditional database techniques [10]. Secondly, the system is, by nature, highly replicated. Each player sees at least part of the game world and the characters residing in this part. This is typically achieved by having at each client copies of the relevant game objects that are updated whenever changes occur [24]. Given that such games can have thousands of players playing at the same time, the update rate and the degree of replication are huge. Propagating all changes to all copies in an eager fashion, that is, within transaction boundaries, in order to provide a consistent view to all players is simply not possible [6]. In many cases, existing game engines restrict the number of players that can populate a game region or instantiate the game to reduce the number of updates that are possible [15]. Furthermore, they often simply allow many inconsistencies to occur, and the players have to accept them [21]. Durability is often handled in a very optimistic fashion and the game play that happened shortly before an outage can be lost [18].

2.1.1 Action Complexity

Game actions vary widely in complexity and importance. In general we compare them along the following factors:

- Repeatability: an action that is easily repeatable or frequently executed does not require a high level of consistency because transient faults can be tolerated. Actions that are irreversible or non-deterministic must be dealt with greater care.
- Number of objects: Atomicity becomes a greater issue when an action involves more than one object.
- Number of reads: Some actions have conditional effects based on read attributes. Isolation is more difficult to maintain as the number of reads increases.
- Importance to the integrity and enjoyment of the game: Actions that have an effect on a player's virtual property highly influence the player's satisfaction with the game. Those actions either take a long time to accomplish or require real currency to perform, and thus, require a high level of consistency.

For instance, moving a character is considered of low complexity because it only affects a single character and is easily repeatable. The player simply selects a destination and the character will gradually update its position attribute towards the target. Selecting a new destination will stop current movement and initiate a new action towards the latest target. Trading items between two players is highly complex because both parties must agree to exchange precisely the same set of items. This typically consists of a series of interaction steps between the two players. A player must send a trade request to another player, who must accept. They must then both offer items from their personal inventory to trade. Once both players have

accepted the conditions of the trade, they must then transfer ownership of the items they wanted.

In the examples above, consistency is much more important for trading than moving. This is largely in part due to the repeatability of each action. Should an moving action fail (e.g., the character is not at the intended location), this failure is easily tolerated by initiating a new movement action to compensate. For trading, a possible inconsistency can result in one party retaining ownership of all traded items. In this situation, it is quite possible that the advantaged player will refuse to rectify the trade. The impact of inconsistencies in trading actions is thus more important than moving.

2.2 Game Architecture

In this section, we present a typical game architecture and execution model of MMOGs which takes into account the requirements of actions described earlier. We discuss how current systems commonly work and what kind of consistency guarantees they provide. We first focus on a single server system. Section 2.2.3 discusses multi-servers.

2.2.1 Replication Architecture

Each player hosts the client software of the game. It is able to render the game world, and receive input from the player. Players control their avatar and can submit actions. The client software sends them to the server who serializes them, adjusts the game state and notifies all players that are affected by the game change.

This can be supported using a primary copy replication mechanism with master/replica proxy objects [24, 6]. The server holds a master proxy for all objects in

the world. This master contains the latest and correct state of the object. Clients hold read-only replicas of objects the client is interested in. Interest here refers to any object the client may possibly need to read data from and/or interact with from its character's current location. The process of dynamically adjusting which replicas a client holds in its replication space is called *interest management*. There exists various ways of defining what a player might be interested in [7]. For example, one could define a circle around the player's character; all objects within this range are declared interesting for the player and thus, the player would have replicas of these objects. Figure 2–2 shows a possible interaction range of the highlighted character. The client controlling that character would have a field of vision limited to this circle and would only hold replicas of objects inside its radius. Figure 2–3 shows the replication architecture. The master objects are represented by dark shapes contained by the server. Whenever a client wants to perform a write operation on a replica (represented by light shapes), the call is forwarded to the master (1). The master then validates the call and propagates the update to all its replicas (2).

Action execution. Actions consist of a sequence of read and write operations (see Section 2.1). Ideally, an action is executed as a transaction providing all ACID properties. Basically all actions include at least one write operation and thus have to be sent to the server for execution at the master copies. In many systems, the server executes actions serially. However, in order to properly exploit parallel and multi-core architectures, concurrent execution should be possible. In this case, a concurrency control mechanism needs to be implemented, e.g., a strict two-phase locking protocol [27]. For instance, picking an item comprises of reads on the item

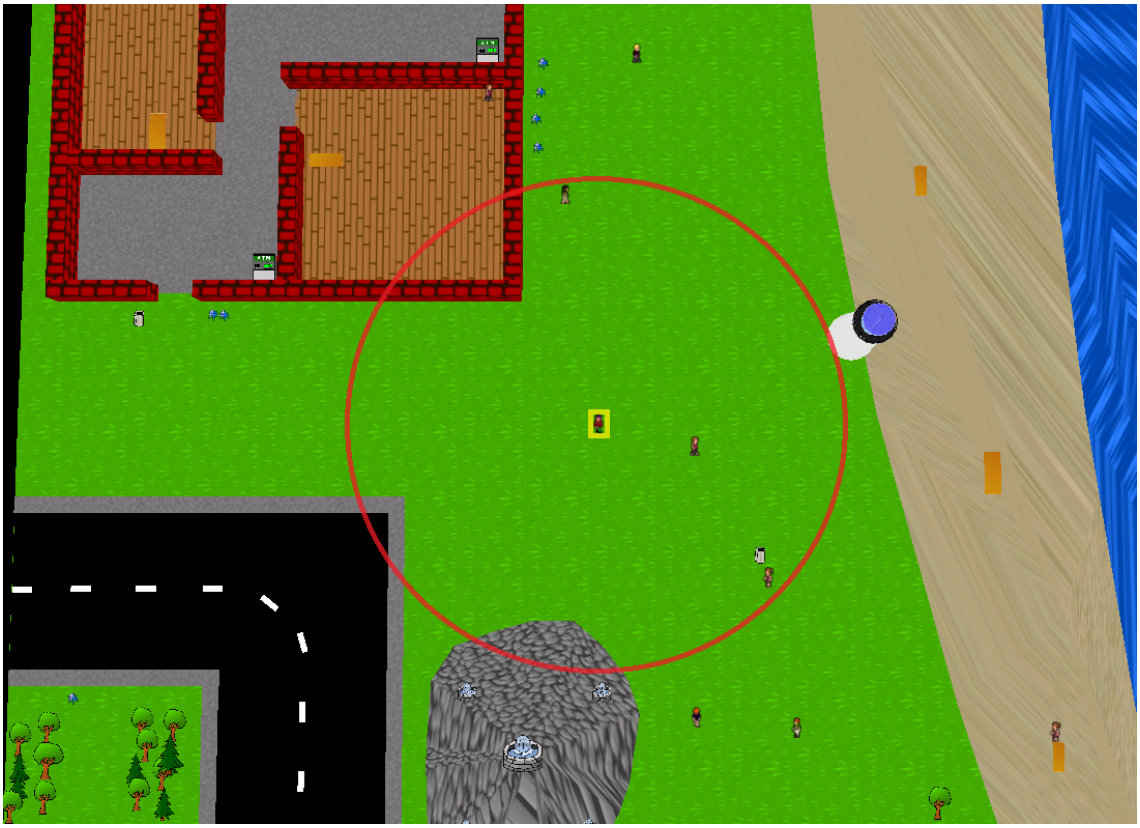

Figure 2–2: Sample game screen and interest radius

followed by a write on the same object. When two players want to pick up the same object, one action will receive the lock on the item first, determine that it is still on the ground, place it in the inventory of the player's character and then release the lock. The lock is then granted to the second action which will detect that the item is no more on the ground and thus abort. Similarly, assume one action breaks a bottle and another picks it up. If the break is serialized before the pickup, then a broken bottle is picked up. If the pickup executes before the break, then the break will not succeed as a bottle that is in the inventory of another avatar cannot be broken. In

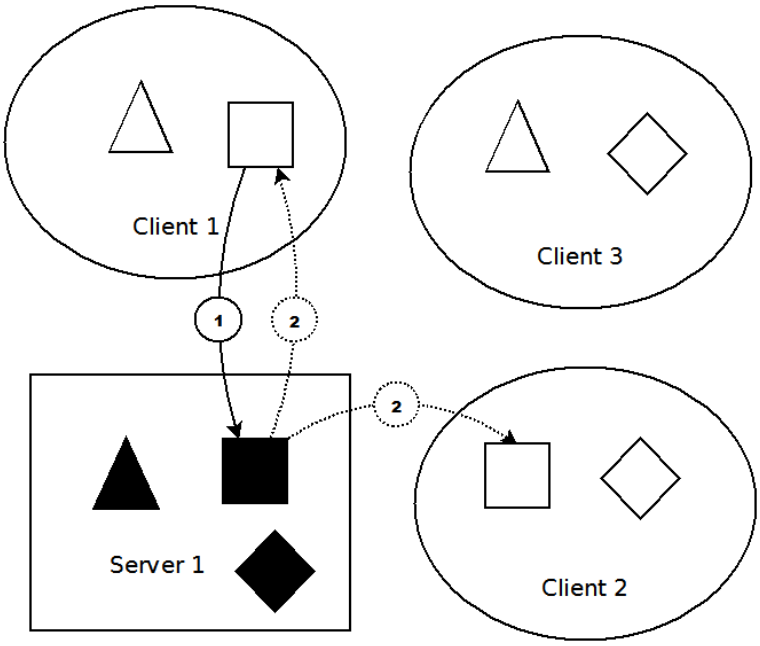

Figure 2–3: Sample Master/Proxy execution

general, we assume all executions to be serializable at the server, which holds the latest and correct game state.

Update propagation. Successful updates need to be propagated to all replicas. Such update propagation can be eager (synchronous), that is, within the boundaries of the transaction performing the update on the master, or lazy (asynchronous), that is, only after the transaction at the server commits [16]. Given the sheer amount of updates and the possibly large number of replicas, eager replication is not feasible in the general case [6]. Instead, current game engines push the updates only after commit at the server, but they do so as fast as possible. Nevertheless, the replicas at the clients are potentially stale, lagging behind the correct state. If an action updates several objects, then the changes should be applied atomically at the clients. If

updates are propagated on an object basis, then it is possible that the object updates related to a single action do not occur as one logical action at the client's game world. This can be avoided by sending all updates related to an action in a single message and then apply them together. However, this might be inefficient as some clients might have replicas of some but not all of the affected objects. Alternatively, the client might defer applying updates to an object until all update messages related to an action have arrived at which point they are applied to the replicas simultaneously.

Round-based execution. Many games are implemented in a round-based fashion [32]. At the beginning of a round, clients send their actions to the server. The server serializes, executes and commits the actions, and then sends all changes at the end of the round. In order to provide fairness, the server has some mechanism as to guarantee that in each round all updates have an equal chance to be serialized first. This mechanism avoids that fast clients see changes first and react to them before slower clients would be able to do. Using this round-based approach, the state at the clients is typically stale by one round as a player, when submitting its own actions, cannot see the actions that are submitted during the same round.

Optimistic execution. The fact that update propagation is asynchronous means that a client does not receive immediate feedback after executing an action. This can be problematic for actions that are not idempotent. For instance, a player may be trying to buy an item which has several units in stock. If there is no immediate purchase confirmation, the player may think that the purchase failed, try again and inadvertently buy multiple copies of the same item. In order to hide the delay to perform actions, some engines execute actions optimistically at the client

replica before receiving the successful changes. If the action fails at the server or has a different effect, the optimistic execution is rolled back [18]. Such behavior might be acceptable for some actions. This also suggests that for certain actions, a synchronous notification is necessary to ensure that clients get a confirmation.

2.2.2 Read Operations

Most actions also have read operations, and understanding what objects are actually read and how the reads are performed is crucial to provide the proper level of consistency.

Action reads. Consider the following actions:

- Moving an avatar typically first reads the current position of the character to verify the destination is reachable. Generally, only the player owning the character can move the character, so there are no conflicting updates on the character position.
- When a player wants to drink water from a bottle, the content attribute of the bottle is first checked: only if there is still water in the bottle, can the action succeed. The value of the content attribute is reduced, and a player's energy level is increased (implicitly reading the energy level attribute first).
- In order to pick up an item, first a read on the item and the inventory of the avatar is performed, then the position of the item is set to the inventory of the avatar, and the inventory is extended by one more item.

These read operations are explicitly written into the action code. Hence, we call them *action reads*. Typically, they are first performed on the replicas residing at the client side. The action must be locally verified on these (possibly stale) copies

before it is sent to the masters of involved objects. At the master side, the read operations have to be repeated, as the state at the server might differ from the state at the clients. Thus, the reads at the client can be considered preliminary, simply for validation purposes and to avoid sending unnecessary actions to the server. However, an action deemed possible at the client might fail or have a different effect at the server side due to differences with the masters state. For instance, although a player saw an item on the ground, the pick up request might not succeed because the state at the server indicates that the item is already in the inventory of another avatar. Similarly, the amount of water actually drunk might be smaller than expected by the client, as somebody else might have taken water from the bottle in-between. This is possible because clients can have stale data. Conversely, notice that an action might fail local validation but in reality be acceptable according to the server state. This side effect is considered tolerable since this more conservative approach cannot result in inconsistencies.

Thus, while the execution at the server is serializable, from the perspective of the client we have the problem of non-repeatable reads. This comes from the fact that each read is executed twice, once on the possibly stale replica at the client and once on the master at the server.

Understanding the effects of changed attribute values is crucial to decide which level of consistency is actually desirable. In some cases, failure or a different effect of an action is acceptable. For instance, the player has seen another player nearby on the game field and is aware that the other player might be quicker in picking up the item or taking water from the bottle. In other cases, this might not be

acceptable and the system needs to ensure that the client can read the current value of an object. In our approach, we will differentiate between actions with various consistency requirements, optimizing update propagation when possible, being more stringent when necessary.

Client reads. Action reads are read operations that are explicitly encoded into the action. However, a player can always observe the state of all its replicas at any time, whether they are stale or not. The player will use this information when deciding what action to perform. Essentially, the player is reading certain semantically relevant attributes before submitting an action. The key aspect is that any value from any object visible to the player can potentially be read and affect the judgment of the player. These reads are implicit and are not part of the action code. Therefore, each action has a *client read set*, determined by the player, which is read before the action is submitted. For instance, consider a player who decides to move to a location in order to pick up an item. Clearly, the move action does not read or update the item but only the position attribute of the player's character. Yet, the decision to move the character is directly dependent on the position of the item, as determined by the player. The action read set contains the current position of the player's avatar, the client read set contains the location of the item.

Because this read set varies depending on the context of the action and also on the strategy followed by the player, it is not possible for the system to determine it. There exists many reasons to move a character and it is generally not possible to determine the set of attributes that influenced the player to submit a certain action. The most conservative approach would be to assume the client read set to contain all

possible reads, that is the entire observable state by the client with all its replicas. This would however severely limit concurrency and performance of the system. Thus, client read sets are ignored by current game engines.

The existence of this client read set complicates the problem of non-repeatable reads. Consider the transactions T1 and T2 from the Schedule 2–1. T1 is a transaction which writes A and T2 writes A and B. Assuming we are using master locking, each transaction receives the appropriate locks at the server. With only these writes, there would be no conflict since T1 is serialized before T2. Now suppose that the client's decision to perform T2 is directly dependent on its read value of A. This client read is local and performed on its replica and therefore does not lock the server. Note that this read was actually performed *before* the decision to perform T2. But since logically, T2 was dependent on A, this client read is considered as part of T2. In order to preserve isolation, T2 now has a read-write conflict with T1. If that client read is performed before T1 is able to commit and propagate its change to the replica used by the client, then the schedule is non-serializable. In order to serialize this schedule, we need to treat the client read the same as regular read operations which are locked within the transaction boundaries. If the client read on A from T2 is executed before T1, then T2 holds a lock on A which would serializes T2 entirely before T1. If the client read on A from T2 is executed after T1 has acquired a lock on A, then T2 must wait until T1 has finished executing before reading the updated value of A (see Schedule 2–2). The client would then have to decide if it wants to perform T2 based on this updated value of A.

However, as mentioned before, client reads are performed by the client outside of transaction boundaries. At the moment the read is being made, we cannot determine whether it will be part of a transaction or not since it is part of the decision-making process of the player. If it is part of an action, then it must be considered as a read for that transaction when determining serializability. If it is not, then the read alone is not considered a transaction. Therefore, if we were to acquire locks on every read a player makes, there is no guarantee that there will be an accompanying action which will depend on that read and release the lock. Locking client reads would then either have a timeout if there is a bound on client reads (i.e. a client does not initiate actions based on reads performed a long time ago) or that the client can explicitly declare that it is about to make client reads. In that case, we can lock reads made from that specified point on and release them when an action has been performed or when the client explicitly release them.

Our solution will handle client reads accordingly depending on the complexity of the action.

2.2.3 Distribution

Most current games run on large server clusters and not on a single server. However, their distribution model is very simple. Each server hosts an individual instance of the game or a well-defined part of the game world. Players can only see objects and other players that reside in the same game instance or part of the world. Thus, each client is connected to only one server that hosts all the relevant master copies. Each server can then implement the above architecture without coordination with other servers.

T1	T2
begin	begin
	C: client-read(A)
S: lock(A)	
S: write(A)	
S: unlock(A)	S: lock(A)
commit	S: lock(B)
	S: write(A)
	S: write(B)
	S: unlock(A)
	S: unlock(B)
	commit

Table 2–1: Transaction schedule with local client reads

T1	T2
begin	begin
S: lock(A)	
S: write(A)	S: lock(A)
S: unlock(A)	S: lock(B)
commit	C: client-read(A)
	S: write(A)
	S: write(B)
	S: unlock(A)
	S: unlock(B)
	commit

Table 2–2: Transaction schedule with locking client reads

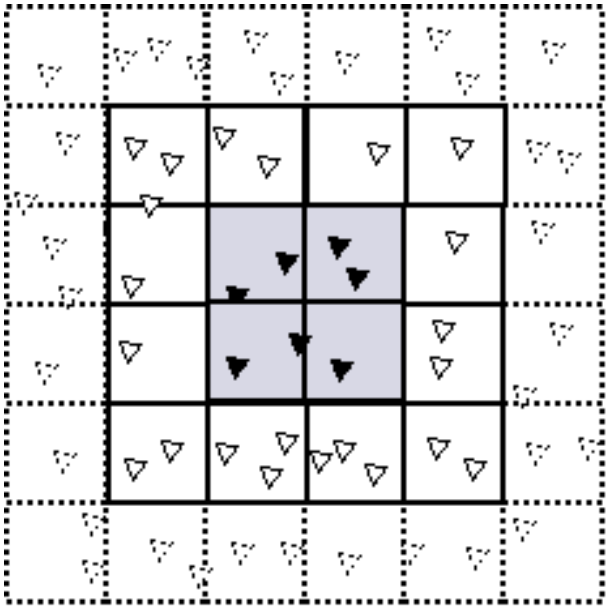

Figure 2–4: Cell-based architecture

Recently, several approaches have been developed to treat the world as a continuous space [9]. Each server holds the master copies of some objects. In particular, one way to assign distribution of game objects is by using cells. Cells form non-overlapping regions of the whole game world. The owner of each cell holds the master copies of objects that are contained in the cell. Clients can host replicas of objects whose masters reside on different servers. This could be the case, for instance, if an avatar is close to a cell boundary. Such an architecture allows for a better game experience due to the continuous world, and is more flexible in regard to load balancing, as cell size can be adjusted dynamically. Masters are migrated whenever cells change size or when a player's character moves from one cell to another.

In order to do interest management, each server typically holds replicas of objects that might be interesting for the avatars it manages. Figure 2–4 shows a cell-based

architecture centered around one server. The shaded center area is the ownership area of the server. The masters of the objects in this area (dark triangles) are managed by the server. The surrounding area delimited by continuous lines is the interest range of the server. It represents the maximum range for which any object inside its ownership area can interact with. The server holds a replica of objects in this range (light triangles). Any object outside of these bounds are not relevant to the server (dotted area with dotted triangles). Please note that dynamic objects can change location and therefore change status.

Actions can access distributed objects, i.e., objects whose masters are located on different nodes. Thus, atomic execution across all servers becomes a deeper issue. It can be achieved by using two-phase commit but this appears too expensive for most action types in MMOGs. Instead, we must use our relaxed action models to handle distributed actions.

2.3 Mammoth: a Multiplayer Game Research Framework

Mammoth is a fully functional MMOG developed in Java [24]. Its main purpose is to serve as a testbed for experimentation in game-related technologies such as distributed systems, fault tolerance, databases, artificial intelligence, content generation, networking and concurrency.

2.3.1 Object Hierarchy

Mammoth is a typical MMO game consisting of a continuous game world populated with various "world objects" (see Diagram 2–5). Static objects are *immutable* and their attributes are initiated at the start of the game. They are also non-interactive and thus not a concern in the context of this thesis. Static objects are

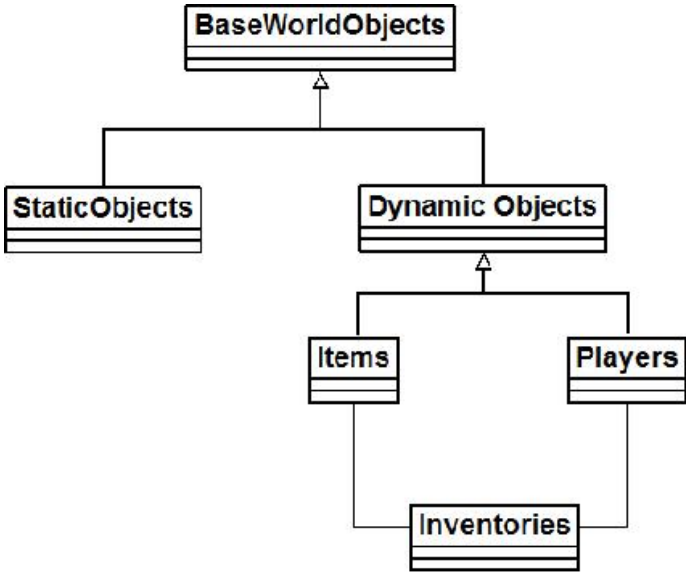

Figure 2–5: Simplified object hierarchy

usually background objects or obstacles such as trees and walls. Dynamic objects are *mutable* and their data can change during the course of the game. Player characters are considered *active* since they can be controlled by a client or AI. Items are objects which are not controllable. Some of them are pickable and can then be put in an inventory or on the ground. Each character has its own personal inventory and certain items (such as bags) also have one.

2.3.2 Architecture Overview

In order to facilitate experiments, Mammoth is built as a modular framework. Each component provides a distinct set of services and multiple implementations of the same component can be swapped in and out of the system. Components interaction is regulated through interfaces. Figure 2–6 depicts the different components of Mammoth. Of particular interest is the replication engine which uses master/proxy

mechanism with interest management (see Section 2.2.1). This replication uses a network engine for communication between the nodes, which must support publish/subscribe messaging to accommodate replica updates [24]. The idea is to have the master of an object publish updates which will be served to its subscribers, namely the replicas. The replication engine is cell-based and allows for distribution of the master proxies (see Section 2.2.3). The persistence manager is our implementation of our solution and is in charge of maintaining a persistent image of the game state on stable storage. The controller component is in charge of handling actions and will be directly involved in our solution along with the replication engine.

Note that Mammoth does not make the distinction between clients and servers, since any node in the system can manage masters. Furthermore, cell owners are not required to manage the masters of objects in its cells, they are only expected to perform interest management for players located in its ownership area. For the sake of clarity, we will continue to refer to nodes holding masters as servers and nodes requesting actions as clients. Our solution is orthogonal to these properties.

2.4 Other Proposed Solutions

In this section we present related works proposing other solutions for handling actions and persistence.

Action protocols. The action based protocols employed in [18] also check consistency at the action level. Their approach relies on optimistically executing actions locally and reconciling when inconsistencies are detected. Updates can be omitted in favor of local simulation. Thus, the client requesting the action can locally update its replica after sending its action to the master. The game-aware

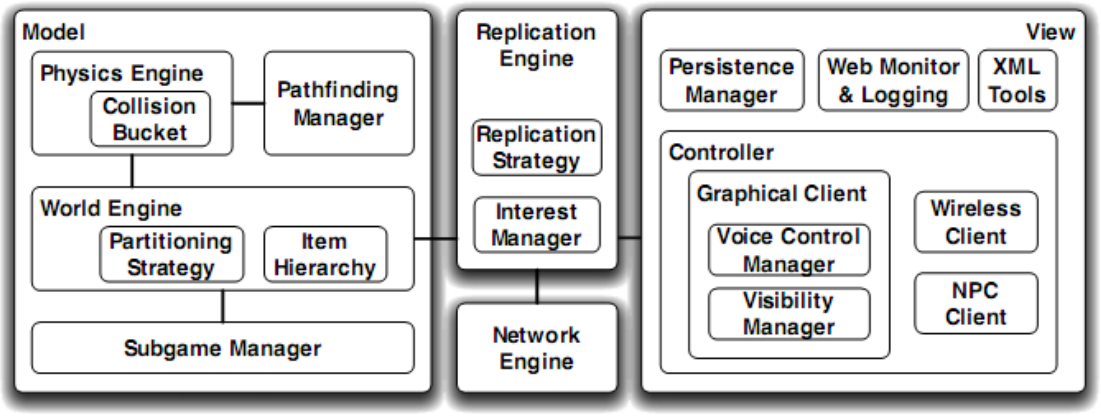

Figure 2–6: Components of the Mammoth framework

portion of the paper offers a reduction on the number of messages sent to clients by maintaining only an incomplete state of the game required, which is similar to our concept of interest management. However, our models are orthogonal to this concern; the models will propagate the updates to all interested parties and can work with any interest management technique.

SEMMO [19] is another consistency protocol which takes game semantics into account. The solution relies on clients performing the computations locally and uses the central server as a coordinator for determining the serialization order of the actions. The distributed architecture of SEMMO is partitioned as in our cell-based approach. The protocol is optimistic in that action requests are sent to the server and replies contain the conflicting reads and actions that the client must evaluate before executing its action. This protocol does not consider the problem of client reads. At the time the client decided to perform an action, it may have based its decision on stale data. Even if conflicting reads and actions will be sent back to

the client, it still does not give the opportunity for the client to decide whether or not the action is still desirable to execute. Instead, it will be executed based on the conflicting data received. Client reads isolation is thus not considered.

Colyseus [6] uses primary-copy replication in its distributed architecture. It supports a rich query interface used for interest management (determining which set of replicas each client holds). Colyseus considers the problem of missing replicas where a client's view is missing an object that should be visible. Such concern is addressed by interest management. Colyseus also considers missing or late updates simply by specifying game-specific bounds on the staleness of replicas. The paper also only considers inconsistency at the replica level and not in the context of transactions and the resulting inconsistencies between the masters involved in an action.

Our solution has similarities with the various degrees of isolation, such as read committed and serializability, that are offered by traditional database systems [4]. In both cases, the developer is offered a trade-off between performance and consistency. Our approach goes further as we not only consider isolation but also durability and the degree of staleness of replicated objects. An interesting analogy exists in the way the levels are developed. The different levels of isolation were proposed when locking was the predominant concurrency control mechanism, and each level can easily be implemented through a certain locking pattern (no, short or long read locks). The stronger the level of isolation, the more costly is the locking scheme. In our case, we assume a certain execution model, namely a traditional client/server model where clients maintain replicas of objects that are interesting for their avatars. Given this

model, we develop increasingly stronger levels of consistency that can be implemented using increasingly complex and costly coordination protocols.

Our solution also has similarities with weak consistency models in replicated databases [13] where read-only replicas have some form of bounded staleness. Such staleness factors have been explored widely in the database community.

Persistence. Standard checkpointing techniques have been evaluated for persistence in MMOGs [31]. This evaluation does not make use of game semantics and only considers the rate of updates. Their motivation for persistence stems from the fact that although only a small subset of actions require transaction guarantees (and thus durability), persistence is still required for other action types such as movement, hence the need of global persistence. However, we differ in that we take into account game semantics and only offer durability as required by the action.

Basically all commercial MMORPGs support some form of persistence [23, 33]. In particular, the game EVE Online only logs transactional updates [17]. Guild Wars periodically stores each character in a database as a BLOB [28]. [30] presents a platform that logs activities of simulated player characters in MMOGs for data mining purposes.

Consistency has been widely discussed in the context of distributed server architectures [9, 8] or peer-to-peer infrastructures[25, 22] but not in regard to the interaction between a server and its back-end persistent storage system.

CHAPTER 3
Transaction Models for Actions

In Section 2.1.1, we have seen that actions vary greatly in complexity and importance. Clearly, given this diversity in action types, there is no one-fits-all solution to consistency. Instead, there are ample opportunities to optimize for performance whenever possible while providing strong consistency whenever needed. The final goal is to provide a system that is scalable, supporting the huge amount of players of MMOGs, to provide the real-time response times needed for interactive, enjoyable game play, and to provide the level of consistency needed to support complex game semantics.

We achieve this by defining game-specific consistency categories and providing a suite of protocols to implement them. Game developers can then decide for each action the appropriate consistency category, and the game engine should ensure that it is achieved. This will allow them to choose the right trade-off between performance and consistency. One key observation is that low complexity actions usually have a much higher volume than higher complexity actions. For instance, a player frequently moves his/her character but will seldom execute complex trading transactions. By executing low complexity actions using efficient protocols providing low consistency, the game engine can achieve scalability and performance.

3.1 Consistency Categories

Games consist of various types of actions with different consistency requirements. Thus, we believe that a multi-level approach to game consistency provides the best trade-off between consistency and performance: some actions can be executed with less consistency while others need higher levels of consistency. In this section we present consistency categories that refer to the various levels of transactional properties possible for actions. Lower categories are more efficient and scalable while higher levels offer more consistency guarantees. Each of the five categories have been designed to serve a different purpose in game semantics. We will explain what each category provides, its usefulness and examples of actions that require it. We will also show action handling protocols that can satisfy the requirements of each category while maximizing its performance and scalability.

The three lowest category levels differ in the way they handle write operations. In particular, they define the staleness levels of replicas. The three highest levels differ in the way they handle read operations. They differ in the way in which they allow stale data to be read and thus differ in the level of isolation they provide.

Our approach is split into two parts. The first discusses how write operations are handled. In particular, we propose to use lower levels of consistency for some attribute types in order to increase scalability without significantly reducing the game play experience. The second part discusses how read operations are handled. Here, we propose stronger levels of consistency for certain actions in order to better maintain game integrity.

Our consistency categories are based on the operations performed by actions and the degree to which the values read at the client side may differ from the server side state. The granularity for operations are object attributes and not entire objects in order to provide more fine-grained concurrency. As concurrently executing actions might belong to different consistency categories one has to be careful that low consistency actions do not negatively affect high consistency actions. Thus, attributes also need to be assigned to consistency categories in order to know the maximum consistency level that can be provided to an action involving this attribute.

Definition 1. *An attribute x is said to be of category c if c is the lowest category level for which there exists an action which writes x.*

In this section, our definition of consistency refers to *isolation* and *replica atomicity*. Isolation considers the effect of stale reads and how they are handled. Atomicity here refers to the state of a master and its replicas. An action is considered atomic at the replica level if all the replicas of all the masters involved will apply the same state change as its masters. Most commonly, this is maintained by having the masters send exact state change updates to all of its replicas, as described in Section 2.2.1. Atomicity can therefore be violated if not all replicas have updated their state and/or if the replicas did not update their state correctly. We do not yet consider a server failure in this context. As update propagation is asynchronous, the failure of the single server can always lead to violation of atomicity as the server might fail after sending an update to some but not all of the replicas. Fault-tolerance and durability will be treated separately in Chapter 4 on persistence. The categories

Category	Description	Examples
Exact	All involved parties must see the same object versions; synchronous action execution and replica propagation.	Trading items between players.
High	Actions have the intended effect according to specified critical attributes.	Buying an item: item price is critical.
Medium	The effect of an action may be different than intended.	Picking up an item.
Low	Action may be successful; replicas see an approximation.	Player movement.
None	Action may be successful; replicas are not guaranteed to observe the effect.	Player turn (orientation).

Table 3–1: Consistency categories

described below also only consider the single server case; the issues specific to distribution will be treated in Section 3.2. The consistency categories are resumed in Table 3–1.

3.1.1 No Consistency

The lowest consistency category provides no guarantee and follows a "best-effort" *maybe* semantic model. The action is not guaranteed to be executed and replicas are not guaranteed to receive and/or apply the update. There is no isolation. This consistency category should be used for minor actions which usually involve only the player's character and which other players do not necessarily need to be aware of. These actions are typically easily repeatable to compensate failures. For instance, consider the act of rotating a character. Normally, a client's camera view is independent of the character, therefore rotation has no importance on gameplay. If the action fails, it can be repeated until success. This category is also useful for social actions. MMOGs use different "emotes" that characters can use (such as dancing,

smiling, etc.). These have no effect on the game other than graphically and can be started and stopped at anytime. Those actions should be handled at the lowest level to minimize their overhead on the system.

The action read set of a no consistency action is expected to only contain the attributes that are also written. Furthermore, a no consistency action is expected to only update those attributes for which the submitting player is the only one who can change them. Thus, there is no problem with conflicting write operations.

3.1.2 No Consistency Protocol

Actions are sent using an unreliable message channel (e.g., UDP) (see 1 in Figure 3–1). The server might execute them only if it has enough resources to do so. No concurrency control is needed for these actions. The state changes can then be propagated unreliably to the replicas (2).

This is the most efficient and scalable solution for action handling. With no guarantee at all on the consistency of the action, the operations are simply sent as messages through the network.

3.1.3 Low Consistency

Similar to no consistency actions, low consistency actions are expected to only have attributes in their action read set that are also written. Like no consistency, low consistency also does not guarantee that all updates are propagated to all replicas, but it does provide a bound on the staleness of the data. This is useful for large volume actions that need to provide some guarantee in regard to their visibility.

The most common action type that falls into this category is the movement of a character leading to a new position value. The idea here is that while clients may

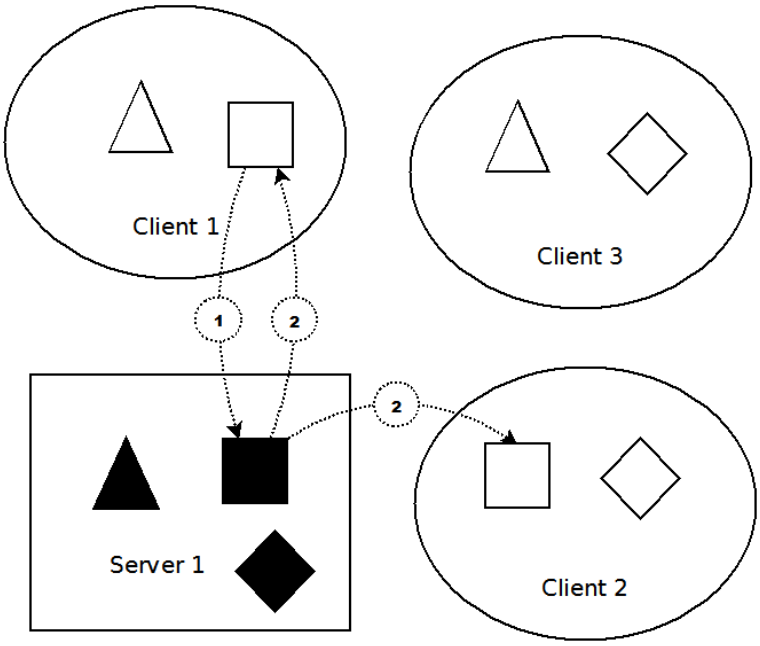

Figure 3–1: No consistency protocol

not perceive the exact position of a character at any time, they always know its approximate location, possibly defined through a specific error bound. For instance, the position seen at any replica should not be off by more than x movements or a distance of y units from the current position at the server.

Another example is the progressive increase of energy points while continuously drinking from a fountain. A player who drinks from the fountain regains energy at regular intervals. Replicas for this player do not need to observe every intermediate energy level while the player is drinking. However, there must be a bound on the staleness of the energy. Thus, there is a guarantee that the replicas will eventually match the true energy level once drinking (the energy increase) has stopped.

In order to provide error bounds between the perceived state of an attribute and the real state, no actions in the no consistency category may update such attributes as their propagation is not guaranteed. Thus, low consistency actions can only be defined over attributes that are in the same category level or higher.

3.1.4 Low Consistency Protocols

In order to achieve the requested error bound, it is not possible to only use unreliable message delivery or drop actions arbitrarily. If the error bound is given as a percentage of received updates, only a fraction of the updates needs to be sent reliably. The other updates are sent with unreliable delivery or can be dropped in overload situations. If the error bound is given as a threshold difference from the true value, then an update is propagated reliably once the difference in values between the master copy and the last reliably propagated value passes the threshold.

A special form of bounded position approximation is provided using dead reckoning mechanisms [5]. Instead of sending every position change, the player's final destination and the average speed is sent to the replicas. The client software then locally generates the individual position changes through dead reckoning. While this does not guarantee a consistent view on the character's position, the replicas will be only slightly off the true position of the character.

Even if a low consistency action changes attributes of more than one object, there is no need for the remote calls to the masters to be propagated together. Essentially, the remote calls to each object are treated separately. However, before the server executes a low consistency action, it requests exclusive locks on all attributes to

write. This is needed for transactions with higher consistency levels. The locks are released immediately after the operations and do not span multiple message rounds.

Although only few action types might fall into this category, we expect that actions of this type build a large portion of all actions in the game. Thus, having some possibility to optimize that action type and adjust it to the workload of the system can be of great benefit. In fact, the most common action type is player movement.

3.1.5 Medium Consistency

The medium consistency category reflects the way current systems typically handle actions as described in Section 2.2. Updates are propagated immediately after commit, or in round-based approaches, at the end of the round. Thus, replicas have stale data but it is stale by at most one round.

Actions can contain arbitrary reads and full isolation is provided at the master level. This means that if we look only at the read and write operations of these actions at the masters, the execution is serializable. But since clients can read stale data locally, the outcome at the server might be different to what the client expected due to the anomaly of unrepeatable reads.

This consistency level is appropriate for actions that affect more than one data item, as it provides consistency at the server and can be implemented fairly easily. Replica atomicity is achieved as all replicas will receive and apply all necessary updates from the masters involved.

Stale action reads are usually acceptable. This is because action reads are read locally only for validation purpose. They are then re-read at the server where they

must be validated again. Therefore, an action which is invalidated will be rejected at the server level. This rejection outcome is reasonable for many cases. For instance, in the case of picking up an object, it will only fail if another player attempts to pick up the item concurrently. In this case, this player must also be close to the item. Thus, each player is aware that a conflict is possible. Even though locally, the client observed the item as pickable, failure of the action is acceptable due to the nearby presence of another player. If the stale action reads do not invalidate the action, the outcome is still acceptable because these reads were only used for validation. Action reads are not part of the decision-making process of the client and therefore their staleness can only result in rejection of the server. To put it differently, if the value of an attribute is part of the decision-making process and is read by an action read, it must mean that another read of the same attribute exists outside of the transaction boundaries as a client read (see Client Reads in Section 2.2.2).

3.1.6 Medium Consistency Protocol

When a player submits a medium consistency action, the client software performs the action reads locally to check whether the action is possible on the client's state. If this is the case, the action is forwarded to the masters of the involved object (see Figure 3–2, 1). The server executes the action within the boundaries of a transaction using strict two-phase locking. After transaction commit (or at the end of the round), each master sends the changes to all replicas (2). Each object sends its updates separately. The client software ensures that updates executed within one action are applied together. Concurrent reads either see all or none of the changes associated with an action. This protocol is simple and keeps the state at the clients

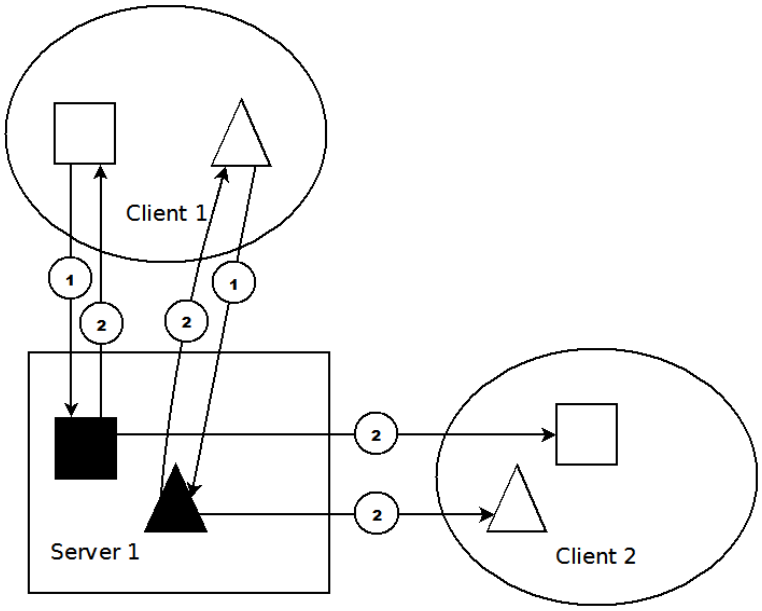

Figure 3–2: Medium consistency protocol

as accurate as possible. Stale reads occur when replica updates of a previously committed action are propagated to the client after it has forwarded another action to the server.

3.1.7 High Consistency

The idea of high consistency is to avoid unrepeatable reads for critical attributes. That is, for high consistency actions, we want to alleviate the problem of stale reads for the most important attributes involved in the action.

Definition 2. *An attribute x is a* ***critical attribute*** if the local value v must be semantically comparable to the master value w.

Definition 3. *For a given attribute x and a given action $A(x)$, the value v of x is comparable to a value w of x if the difference in outcome from $A(v)$ and $A(w)$ is acceptable.*

The definition of acceptability depends on the priorities of the game developers: the difference could be acceptable only if it is in the player's advantage, for instance. Note that the comparison is not commutative.

Replica atomicity is provided as for medium consistency actions. In addition, high consistency actions will contain a set of critical attributes and ensure local read isolation of said attributes. For instance: consider the action of buying an item for a certain price. Price is a critical attribute since it is one of the major factors in deciding whether to purchase the item or not. If the price rises without the client being aware of it yet, then as a medium consistency action, the client would still be able to purchase the item if the funds are sufficient. However, the outcome may not be acceptable because the client has paid more than expected for the item. If buying is not a revertible action (i.e. selling the item is only possible for less than the purchase price), then the player will have suffered a negative consequence which is undesirable. However, if the price decreased, then the player will have no problem paying less for the item. On the other hand, an attribute like the item's graphical representation may be considered non-critical if it does not affect the player's appreciation of the item. Schedule 3–2 illustrates this example: an action T1 raises the price of A (attribute A_i) to 2. T2 still reads the value 1 locally. This value is compared to the master read of the same attribute. Since an increase in price is not considered an acceptable difference, the action is aborted: the client should not buy the item at a higher price unknowingly.

High consistency actions must therefore capture the local reads of critical attributes at the client and compare them to the master reads. If the difference is

T1	T2
(S: $A_i = 1$)	
begin	begin
S: lock(A_i)	
S: write(A_i,2)	
(S: $A_i = 2$)	
unlock(A_i)	C: action-read(A_i)
commit	(C: $A_i = 1$)
	S: lock(A_i)
	S: read(A_i)
	(S: $A_i = 2$)
	S: compare(A_i,1,2)
	S: unlock(A_i)
	abort

Table 3–2: High consistency schedule with an unacceptable stale read

unacceptable, then the action at the server must fail. Note that critical attributes are defined per action. Different actions might have different critical attributes.

Particular attention should be given to the consistency category of critical attributes. In lower consistency levels, updates can be lost or the difference of values between replicas and the master might be very high. Therefore, high consistency actions with low or no consistency critical attributes require a large threshold of acceptability relative to any error bounds those attributes might have.

The purpose of high consistency actions is to provide important actions with certain, clearly defined attributes so that the user can reliably perform those actions with an acceptable outcome. Medium consistency differs from high consistency only in that there are no read operations on critical attributes. Update propagation to replicas is handled in the same way for both categories, guaranteeing replica atomicity.

3.1.8 High Consistency Protocol

The protocol is similar to the medium consistency one. The main difference is that an action submitted to the server contains the client replica values of all critical attributes. When the server executes the read operations on the master copy, the master values are compared to the piggybacked client values. If the comparison is acceptable, the action continues, otherwise it is aborted.

In order to perform the comparison, the action code needs to provide a predicate that, given as input the replica and master values either returns true (acceptable) or false (not acceptable).

From a performance point of view, the message exchange is the same as for medium consistency. However, messages have to carry additional attributes and the server has to perform extra predicate evaluation.

3.1.9 Exact Consistency

While high consistency guarantees that actions at the server only succeed if comparable with what the client expects, the client can read stale data, possibly leading to frequent aborts at the server. Reading stale data at the client and making decisions on behalf of this stale data can be considered an optimistic approach – where the final validation at the server might fail.

Exact consistency is the level where full isolation and guaranteed fresh data is provided. Exact consistency is only possible when client confirmation is integrated into the action itself. This category is designed specifically for complex actions and typically involves more than one player. An example is trading where items and money are exchanged between two players. Given the complexity and the importance

of this type of action for game play, performance is not that crucial. In contrast, all players need to have the accurate state for decision making and want to be assured that the action succeeds properly at all involved parties.

During action execution, the involved players are provided with the latest state of all attributes on which the action performs read operations. Thus, the players make their decision on the correct state of the attributes, and the write operations at the server will succeed as expected by the clients. Full serializability considering all read and write operations is provided. In this context, eagerness also plays a role. In all lower categories, changes are sent asynchronously to all replicas. In contrast, with exact consistency, all involved players receive the updates within the boundaries of the transaction, i.e., eagerly. Thus, eager execution on all replicas of involved players is provided.

Consider, for example, trading between players. Such an action requires multiple steps: players must make offers to one another and then agree and confirm the action. The transaction then takes place and items or currency are updated. Consistency here is critical because if the trade ends up in one player's favor, it is unlikely that this player will want to trade back to rectify the mistake. Furthermore, an action like that is usually isolated from the rest of the game, players will focus their attention entirely on the information presented and used by the action. We can then synchronize all those objects during the action and ensure the players accept to finalize the action.

Exact consistency is only possible for actions that do ask for confirmation. This way, we can determine when and what objects need to be synchronized before the

player wants to commit. It is also the only way we can guarantee that low or no consistency attributes will have exact values at the replicas.

Note that we assume that exact consistency actions usually involve multiple clients and expect client interaction from all involved clients. Such actions must already wait for the players' input, therefore the impact of the delays incurred from the consistency mechanisms is lessened. Conversely, using lower consistency categories for such complex actions would yield small performance benefits only.

3.1.10 Exact Consistency Protocol

We use a pessimistic approach consisting of two steps: a request and a confirmation (see Figure 3–3 for an example involving one object and two clients). After receiving the action request from a client (1), the server sets a long shared lock the involved object in order to avoid concurrent updates. It then propagates the current state of the object to the replicas of all clients involved in the action (2). Once the client(s) have received the latest state, they have to confirm the action back to the masters (3). The server then performs the write operations at the master, and the changes are propagated synchronously to all involved client replicas (arrows #4). Thus, clients receive immediate confirmation about the success of the action. Other replicas are updated asynchronously.

3.1.11 Client Reads

So far, all read operations were restricted to action reads explicitly written into the action code. The problem is that the system has no means to detect the exact client read set. In principle, client read sets can be handled by the high and exact consistency categories. On the other hand, medium and lower consistency categories

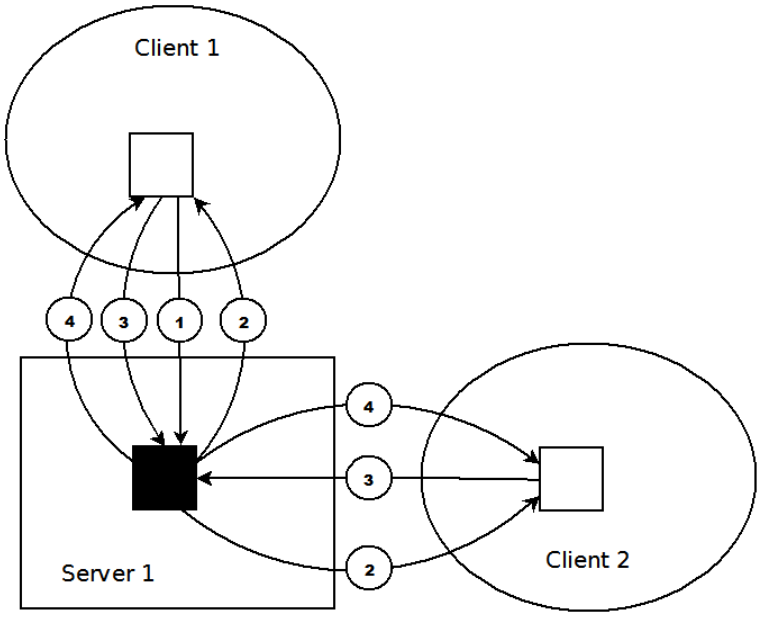

Figure 3–3: Exact consistency protocol

do not take them into account and must therefore be able to tolerate any stale client read.

The medium or lower consistency categories provide no mechanism for detecting stale client reads and dealing with them. Any inconsistencies caused by client reads must therefore be tolerated. Consider Schedule 3–3. A is an object with attributes A_i and A_j, where A_i is the condition of the object ("broken" or "not broken") and A_j is the location of the object (e.g. "on the ground"). T1 and T2 are medium consistency actions executed concurrently on object A. T1 is an action updating the object A. The update affects the attribute A_i rendering the object "broken". T2 is a pickup action on A which updates A_j to put the object into the player's inventory.

In T2, the client makes a local read on A_i and sees that the object is not broken, since the effects of T1 have not been reflected at the replicas yet. It therefore

T1	T2
begin	begin
S: lock(A_i)	
S:write(A_i)	
(S: A_i = "broken")	
S: unlock(A_i)	C: client-read(A_i)
commit	(C: A_i = "not broken")
update replicas(A)	S: lock(A_j)
	S: read(A_j)
	(S: A_j = "on the ground")
	S: write(A_j)
	(S: A_j = "in the player's inventory")
	S: unlock(A_j)
	commit
	update replicas(A)

Table 3–3: Medium consistency schedule with stale client reads

decides to execute action T2 to pick up the object based on that data. The update successfully modifies A_j and sets it to the player's inventory. After the replicas are updated by both T1 and T2, the client will realize that it picked up a broken object. This was not intended as the client only wanted to pick up the object based on the fact that it's not broken. In this situation, this inconsistency is tolerable because the player can compensate by simply dropping the item. If T1 updated replicas before the client read from T2, then the client would have not picked up the object in the first place, therefore the difference in state between the two possibilities is minimal.

For high and exact consistency, there are mechanisms in place to handle client read sets. Potentially, every attribute of every replica held by the client is part of the client read set. With high consistency, one can make all attributes critical requiring the state at the server to be identical to the state at the client. If one attribute value

differs, the action fails. With exact consistency, the server can lock all attributes of all objects for which at least one of the involved clients has a replica, and then propagate the latest state for all these replicas to all involved clients. Although possible, both approaches seem impractical because clients can host large parts of the game state, while it is likely that only a small portion actually affects their decision.

Thus, what is needed is a mechanism to make the client reads explicit and add them to the action read set. This can be achieved with a flexible query language that allows the game developer to specify a reasonable set of objects that could affect an action. The query language must allow for such objects to be detected dynamically during run-time. For example, for a pickup action, all attributes of the item could be put into the action read set as critical attributes. They will then be handled properly via the high consistency protocol. However, such explicit declarations are highly game-specific and will require a proper coding of the corresponding actions. The development of such a query language is out of the scope of this thesis. For example, scripting languages used to model AI behavior could be used here to dynamically determine the read set of an action [32], since AI-controlled characters must make realistic assessments about the game state before taking actions.

In summary, the set of client reads that should be considered for the execution of a high or exact consistency action has to be made explicit in one way or the other, and thus, be included in the action read set. This allows the high consistency category actions to include those client reads as critical attributes, while exact consistency category actions will synchronize the client's view of the read replicas. However, medium or lower consistency actions do not provide any mechanism to deal with

client reads and must be able to tolerate their staleness. In general, the players have to be aware of the asynchronous nature of the games and that a large set of their actions might have a (slightly) different effect as what was anticipated.

3.2 Distribution

As explained in Section 2.2.3, object masters can now reside on different sites. Therefore, additional care is required to deal with distributed actions, i.e. actions which involve masters on different servers. In particular, the problem of master atomicity is introduced with distribution. This form of atomicity relates to the execution at the master level. Since masters can be located in different servers, synchronization between the different masters is now required to ensure atomic execution of the action. In contrast, replica atomicity refers to the propagation of the action from one master to all of its replicas (see Section 3.1). We now look how our protocols can be extended to handle those issues.

Execution model. We distinguish actions of different distribution degree. First, a *simple distributed action* may only update objects whose masters all reside on the same server but it can read objects whose masters reside on different servers. Nevertheless, we assume that the server holding the masters of all objects in the write set also holds at least replicas of the objects in the read set. This is a reasonable assumption as the server needs these replicas for the purpose of interest management. Simple distributed actions only need to be sent to the server who holds the master copies of updated objects. The action is completely executed at this server.

Complex distributed actions write objects whose masters reside on different servers. Thus, these actions require a distributed execution. Each server holding

a master copy to be updated executes part of the action. The action read set is split and each server performs the reads needed to execute its updates. The action code must be written accordingly and specify for each read r the set of write operations WS_r that depends on it. A read operation r must then be executed at each server that executes a write $w \in WS_r$. Again, we assume that a server has replicas of all objects that it has to read for its sub-action. We assume that the client software coordinates complex distributed actions sending the individual sub-actions to the affected masters.

Master reads. In a single server system, an action executed at the server is always guaranteed to read correct values as the server has all master copies. With distributed actions a read operation might be on an object where a server only has a replica. As update propagation is lazy, the replica at the server might be stale. Relying on this stale information can quickly lead to inconsistencies. Assume for instance two concurrent players want to drink from an originally full water bottle B. Player $P1$ has its master on server $S1$, player $P2$ on server $S2$. The master of the bottle is on $S2$ (see Figure 3–4). When $S1$ executes $P1$'s action, it reads its replica of the water bottle, which is full (2), and gives the avatar the appropriate energy points (1). At $S2$, the concurrent action is executed, providing $P2$ with the same energy points and changing the state of the water bottle to empty (1' and #2'). When $P1$'s sub-action of emptying the bottle now arrives at the master, it is executed (emptying an empty bottle) and succeeds (3). However, the final execution is inconsistent because two players received the full energy points of drinking from

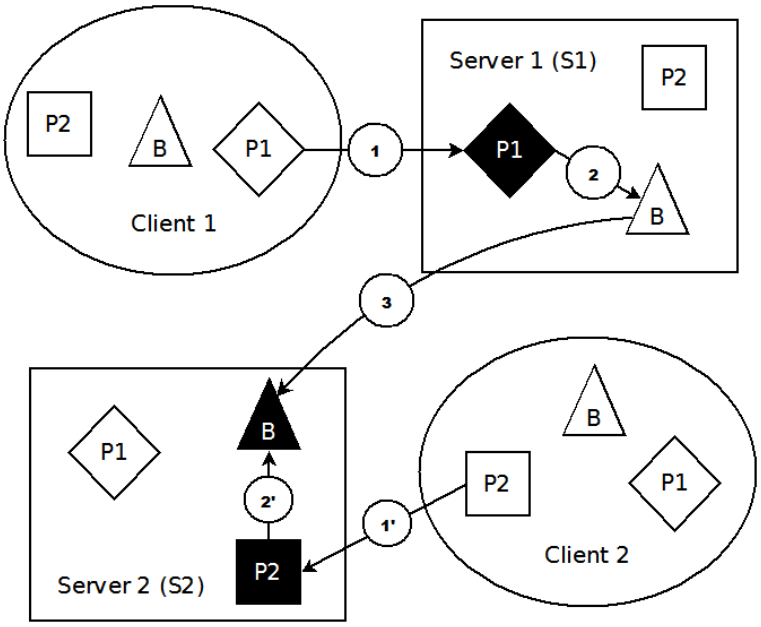

Figure 3–4: Master reads inconsistencies

the same bottle. In contrast, in a single server system, the action to be serialized last will find an empty bottle and not assign the player any energy points.

This problem is particularly prominent in complex distributed actions, since different servers may read the same attributes but not agree on the same values due to stale reads. For simple distributed actions, inconsistencies cannot occur within an action since reads, even stale, are performed on the same server. However, a more general problem arises even for simple distributed actions when the game has assertions which must always be satisfied. One action might read the energy levels x and y of two cooperating players, and change x, while another might read x and y and change y. A general constraint might exist that the sum of x and y should be above a certain threshold. If the masters of the two players reside on different servers, each action is executed on one of the servers. As the replicas are stale and

the concurrent action is not visible, both actions might succeed but the end result could lead to the sum of the energy levels being below the threshold.

No and low consistency actions make no effort to verify that they are applied consistently across the servers. Masters can perform their required reads on their server's replicas. Therefore, no and low consistency actions do not guarantee uniform reads across servers.

This form of inconsistency is much more severe than the optimistic reads at the clients as the game state as a whole becomes inconsistent. Thus, it has to be avoided for medium and higher consistency actions. This can be done by requiring master reads. That is, reads are always performed on the masters, even if they are not being updated by an action, and the read values provided to servers that require those reads.

Master atomicity. A second challenge is to guarantee the atomicity at the master level of complex distributed actions. All or none of the masters need to commit the action. Low and no consistency categories do not guarantee master atomicity, which further differentiate them from higher consistency categories.

3.2.1 Exact Consistency

In the case of exact consistency actions, the client will request from all servers the latest state of all objects read (see Figure 3–5). In contrast to a single server system, the request might now go to several servers. These requests will lock the master copies at the servers (1). The replicas of all involved parties (servers and clients) are then synchronized (2). Each client sends its confirmation to the masters involved in the action (3) and then receives the updates synchronously (4). Since

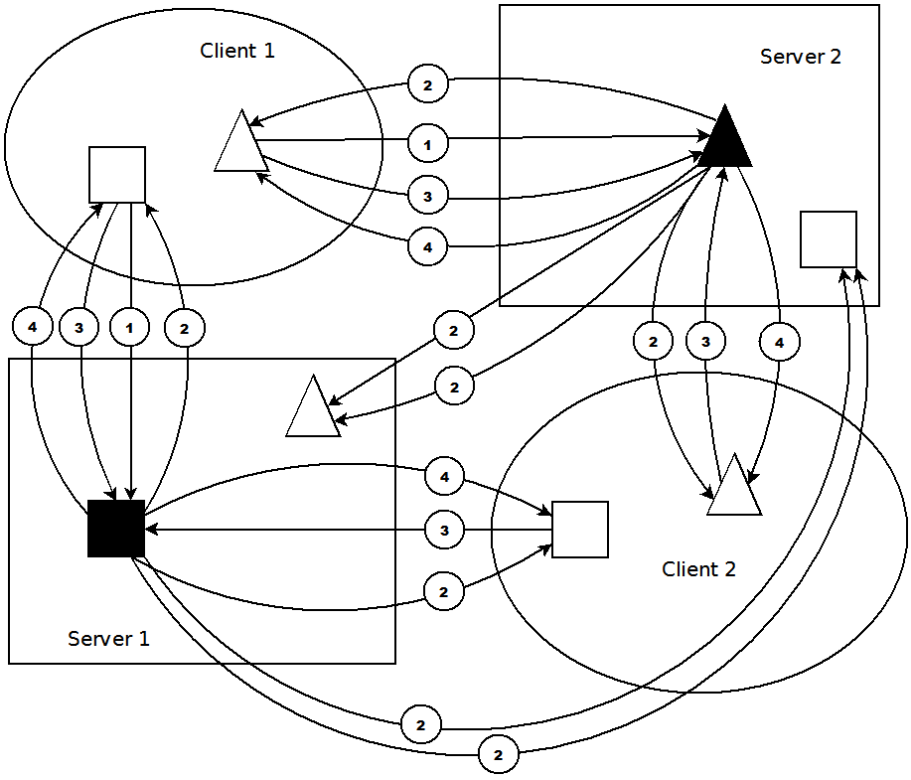

Figure 3–5: Distributed exact consistency protocol

server replicas are synchronized during the action, it is guaranteed that masters read the latest value of other masters from their server replicas.

3.2.2 High and Medium Consistency

Our solution leaves the action coordination to the client. In the worst case, we need the same protocol as exact consistency if the latest complete state of each object has to be transferred to the servers that read the object. However, for a special, but very common action type, a more efficient protocol is possible. This is the case when there is no circular dependency on read operations. Assume the case of picking up a bottle. This action changes two objects, the bottle and the avatar. A bottle can be

picked up if it resides on the ground. That only depends on the state of the bottle itself. That is, the update on the bottle only needs to read the state of the bottle. In contrast, for an avatar to add the bottle to the inventory, the bottle has to be on the ground. Thus, the update on the avatar depends on the state of the bottle.

Thus, the sub-actions for the individual write operations of an action are sorted based on their dependency. First, the client submits a sub-action for objects which do not depend on the state of other objects. The server acquires the appropriate locks and executes the sub-action. If it succeeds, it sends the values of read attributes from its objects needed for other writes to the client. The client then forwards the required reads with its next sub-action to the corresponding server. The server installs the changes, requests the locks and executes the sub-action. This repeats until either all sub-actions succeeded or one has failed. In the first case, the client submits the commit to all servers. In the latter case, the client sends the abort information to all servers where the sub-action succeeded, which roll back their operations. In this approach, receiving the latest state is combined with the submission of sub-actions.

Figure 3–6 shows an example using two objects and two servers. Client 1 submits the sub-action to the first object (1), which returns successfully with read values necessary for the rest of the action (2). Those reads are then sent to the second object (3). The second master successfully completes the action and sends confirmation back to the client (4). Furthermore, it can already start propagating updates to replicas since it knows it has executed the last sub-action and does not need to wait for other objects (5'). Meanwhile, the client now sends a commit back to the first object (5), which now proceeds to update its replicas (6).

Algorithm 1 Distributed linear-locking algorithm

Require: $\forall i, j \in |O|$ where $i < j : R_{O_j,O_i} = \emptyset$
  $V \leftarrow \emptyset$
  for all $x \in O$ **do**
    $In \leftarrow$ set of $V_i, \forall i \in R_{x,y}$ where $\forall y, y \neq x$
    $Out \leftarrow$ set of $R_{y,x}, \forall y \in O$ where $x \neq y$
    if x is the last element of O **then**
      $subActionAndCommit(x, In, Out)$
    else
      $W \leftarrow subActionAndWait(x, In, Out)$
      for all $i \in R_{y,x}$ where $\forall y, y \neq x$ **do**
        $V_i \leftarrow W_i$
      end for
    end if
  end for
  if every call succeeded **then**
    for all $x \in O$ except the last element **do**
      send commit to x
    end for
  else {the action failed}
    send rollback to executed $x, x \in O$
  end if

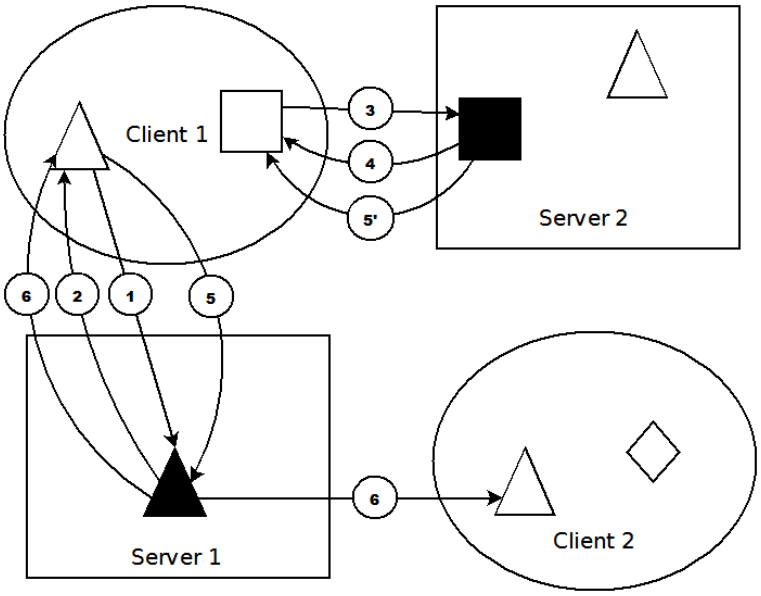

Figure 3–6: Distributed high/medium consistency protocol

The distributed linear-locking protocol in Algorithm 1 details our approach. $R_{x,y}$ is a list of attributes from object y necessary for object x to execute its sub-action. O is a list of objects involved in the action, ordered based on their reads. V_i stores the read value of attribute i. *subActionAndWait* is a remote call which locks and execute the sub-action for a certain object providing an input set of reads necessary and specifying the output set of reads required. After execution, the object must keep the lock and wait for commit or rollback from the coordinator. *subActionAndCommit* is a remote procedure call sent to the last object in the ordered list which does not lock the object. Since it is the last object in the list, it can commit right away if successful. If a call fails, any previous sub-actions must be rolled back and the action aborted. Otherwise, commit is sent to all waiting objects.

CHAPTER 4
Persistence

We now look at how durability for actions can be achieved in MMOGs. Durability is treated separately from the other ACID properties because it is only called upon during failure scenarios. One desirable property for durability is thus transparency. The game should not be aware of the persistence mechanisms in place. The basic idea of persistence is to monitor the game state dynamically and store it to stable storage. When recovery is needed, the persistence data is restored. Ideally, the recovered data is exactly the same data at the time the game was shut down. Exact solutions, however, prove to be too expensive and limit the scalability of the persistence layer, if not the whole system. Our strategy is to handle persistence based on the semantics of the actions. In particular, each consistency category for action handling can be augmented with persistence. The protocols used for persistence are then designed such that any attribute update stored would satisfy the consistency requirements of the action. Doing so can allow us to use more efficient and scalable approximation storage strategies when appropriate.

In a single server environment, the persistence layer monitors the game at the master level. Changes made to the state of the game are captured by the persistence layer, which sends the updates to stable storage according to the consistency requirements of the action. Upon restart of the server, the game is recovered using the persistence data.

When considering distribution, partial failures are now possible where only specific nodes go down and not the whole system. In those situations, we can use persistence for fault-tolerance purposes. Only the data from the affected nodes would need to be recovered. The challenge here is to reconcile and/or minimize any discrepancies between the state of the recovered objects and the rest of the game.

For this thesis, we only consider failures for game servers. Scenarios involving failures of the persistence structure itself will not be considered.

4.1 Persistence Architecture

In order to make persistence transparent to the system, replicas will be used to track changes in objects. The persistence layer will hold replicas of all objects in the game. Those replicas are updated normally like any other replicas. Whenever such an update is received, the persistence layer can decide how to store the update to stable storage.

To cope with the scalability provided by the distribution of the game, the persistence layer itself must be distributed. Multiple persistence servers can be used, each monitoring a different set of objects. Each persistence server will then be connected to some stable storage, which may be distributed as well (see Figure 4–1).

Each persistence server must be assigned non-overlapping and covering sets of objects to monitor. We employ a central persistence server to coordinate assignment. One possible strategy is to use the same cells which are employed for distribution and load balancing (see Section 2.2.3). Each cell is assigned to a persistence server. The advantage of this strategy is its simplicity: it reuses mechanisms already in place. The persistence server simply needs to subscribe to the cell and interest management will

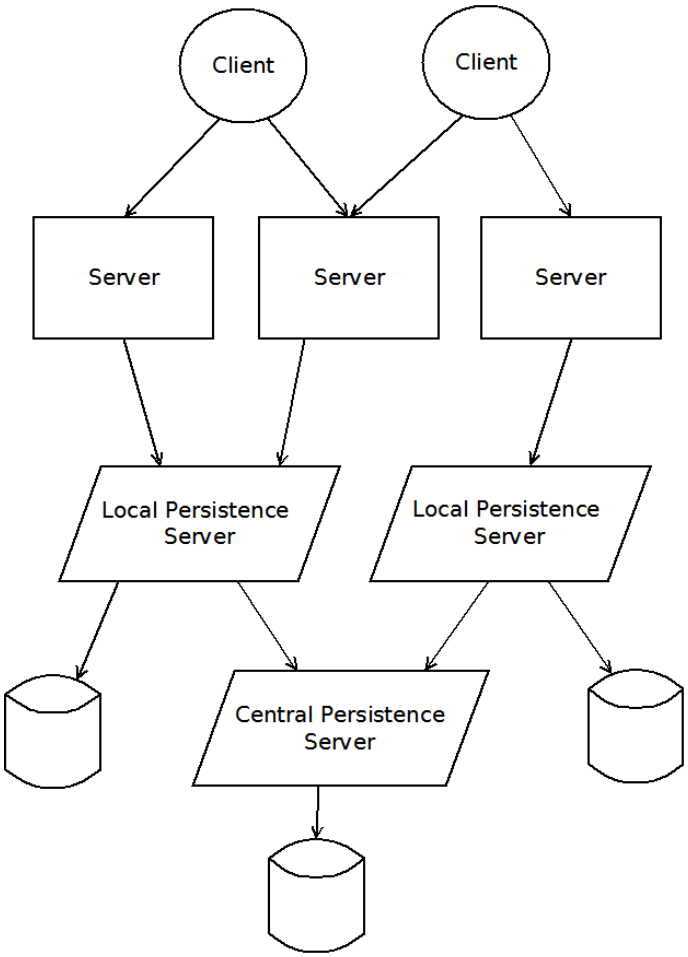

Figure 4–1: Game architecture with persistence

compute which objects are in this cell's boundaries (and thus need to be replicated by the server). Another strategy is to assign game nodes to each persistence server. The persistence server must then be in charge of monitoring every object whose master is located in that server. The advantage here is the ease of recovering a server's content in a fault-tolerance situation. When a server crashes, one persistence server has all

the objects this server had. We do not have to find this information across multiple persistence servers.

4.2 Restoring Data

During a complete restore of the game data, it is also likely that the persistence servers have also been shut down. Therefore recovery must be done from the stable storage data. First, the central persistence server must have an index on the location of each object across its multiple stable storage sites. Another possibility is to centralize all the persistence data back to the central server. When the game servers restart, the system must request objects from the central persistence server, which can then reconstruct the objects from stable storage. Note that it is not necessary for the game system to request for specific objects since objects are created and deleted during the course of the game. Restarting game servers would not be aware of the existence or nonexistence of such objects and it is the responsibility of the persistence service to inform them.

After partial failure of the system, failure detection is needed to determine which object masters have failed. This failure detection must be provided by the system and is not covered in this thesis. Once the set of failed masters is known, the failover procedure can recover those masters from the replicas maintained by persistence servers and reassign master responsibilities to other servers. The game system must interact with the central persistence server and request masters for the failed objects. The central persistence server can then locate the corresponding replicas from its local persistence servers. Those replicas may need some preparation before they are suitable for master duty (see Section 4.3.3). Once ready, the persistence layer sends

Category	Description	Examples
Exact	The persistence replicas must be synchronized with the masters.	Trading items between players.
High	The persistence replicas are consistent according to the critical attributes	Buying an item: item price is critical.
Medium	Stored but may differ due to lost actions	Picking up an item.
Low	Stored using an approximation strategy	Player movement.
None	Does not need to stored	Player turn (orientation).

Table 4–1: Consistency categories for persistence

a copy of the new master to the appropriate server. Local persistence servers are not in charge of masters for the game. The fault-tolerance aspect of persistence will not be covered in detail in this thesis.

4.3 Persistence for Actions

To maximize scalability and efficiency of the system with persistence, actions with varying consistency requirements will be stored differently to satisfy such requirements. Another important point to consider is how the persistence replicas can be reused in case of failure (see Section 4.2). If the persistence replicas are inconsistent with the live data, additional mechanisms need to correct the replicas before they are used as new masters. In terms of performance, our goal is to minimize load sent to stable storage, which can be a bottleneck for our system. Table 4–1 summarizes the material in this section.

4.3.1 No Consistency

Since no consistency does not guarantee reliable update propagation, it is possible that the persistence replica will simply not receive an update. If it does however, it does not need to store it to stable storage, since the minimum requirement of this

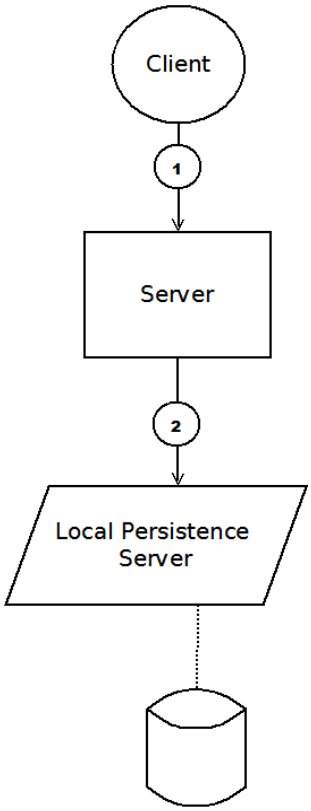

Figure 4–2: No consistency persistence

category is that updates are not received. Therefore recovering a replica that does not reflect the updates from a no consistency action is acceptable. After a client has committed an action on the server (1 in Figure 4–2), the replicas of affected objects will be updated (2). However, the local persistence servers will not write this change to the stable storage.

4.3.2 Low Consistency

At the low level, a master is not required to send updates atomically to its replicas. Upon receiving such updates, the persistence server may decide to further use an approximation strategy on how to store them. As it is acceptable for attributes

not to be exact after recovery of a low consistency action, only some changes need to be persisted. For instance on a movement action, the approximation strategy could ensure that the position is only stored when the last stored position in the stable storage exceeds a certain distance threshold from the actual position. This is a distance-based storage strategy for movement [34]. It reduces the throughput to the stable storage while ensuring an error bound on distance, which is significant in game semantics.

4.3.3 Medium Consistency

At the medium level (and higher), every update must be stored to stable storage since replica atomicity must be preserved. In other words, any update visible to any replica must be reflected in the persistence data. Consider the following scenario: a medium consistency action involves a single object. After commit, the master will propagate the change to all replicas. If the server fails before sending any updates, replica atomicity would be preserved since no one will have observed the action. If some updates have been sent, then replica atomicity is violated. However, as long as a local persistence server replica receives the update, it is possible to recover the effect of the action and later send the missing updates. Therefore, an additional requirement is necessary on the network engine when delivering updates. The publisher/subscriber system is required to either have atomic publications (all or none of the replicas receive the update) or that the persistence replica for a master always receive the update if *any* replica for that master has received an update.

We will now expand the scenario to involve multiple objects, with all the masters residing on the same server (see Figure 4–3). We also assume for now that

the persistence replicas for those objects are located in the same local persistence server. Even with the additional network requirement shown above, inconsistencies can result when updates are not sent completely. After the client has successfully committed the action (1), one of the masters has successfully updated its persistence replica (2), which now reflects its new state on stable storage (3). However, the server crashes and fails to send the update for another master involved in the same action (4). Upon recovery, some masters will reflect the action while others do not. This is inconsistent and equivalent to a partial execution of the action, which violates master atomicity.

We solve this problem by piggybacking replay information in the update. In other words, when a medium (or high) consistency action is sent to a master, it contains enough information to execute the operations of any objects involved in that action. In order to determine whether a replica is missing updates or not, we employ object versioning. Masters increment object versions for each medium (or high) consistency action they are involved in. The action will keep a vector of the current version of each master. Replicas update their version every time they receive an update from such actions as well. When recovering the objects, the local persistence server can determine which replicas are missing updates by comparing their version with the ones stored in the actions. The main idea is that if any object involved in an action has propagated its update to a replica, the action will have been stored on the local persistence server and accessible for replicas of other objects to use. If none of the masters have managed to propagate their updates before crashing,

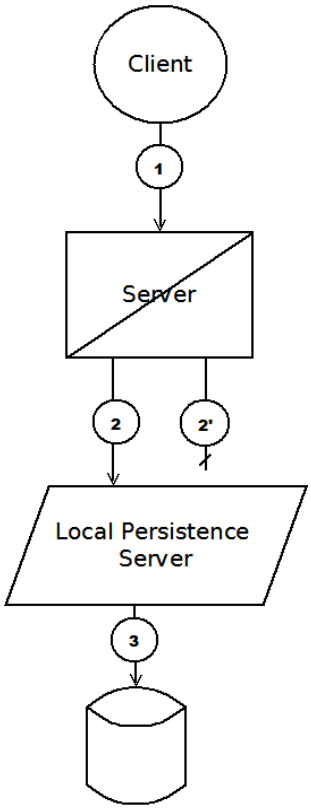

Figure 4–3: Persistence with no atomicity between the replicas of multiple objects

then the action will be missing and will not be recoverable by any replicas. However, since no persistence replicas perceived the action, master atomicity is preserved.

In the distributed case, the above solution needs to be further expanded. If masters can now reside in different sites and be monitored by different local persistence servers, then the replay information for an action needs to be accessible to any local persistence server with a missing update for a replica. This is done by replicating the action on the central server (see Figure 4–4). When an action is sent to the masters (1), they will each propagate its update (as usual) along with the action itself (2). Once the persistence replica receives the action, it will forward it reliably to

the central server (3). The central server may receive the same action several times from multiple sources; duplicates must be discarded. In case of node failure (2'), the persistence server holding a replica of a failed master will request any missing action from the central server (5). The server can then replay those actions on the replica to bring it to a consistent state suitable for master duty and store the changes to the local storage (6).

In the example shown in the Figure 4–4, an action involving object A (version 2) and B (version 3) was successfully committed. However, only the left server holding the object A has managed to propagate the action to its local persistence server. The right server, holding B, fails before sending the updates. For fault-tolerance purpose, a new master of object B must be created. The persistence replica, which is still at version 2 for B, sends a request to the central persistence server for any missing action. The replica receives a copy of the action, which it can apply to bring B to version 3. The replica can now be used as the new master and the action update can now be logged on the stable storage for the local persistence server holding object B.

Algorithm 2 Medium consistency persistence recovery strategy

Replica of object x sends request(x, v_x) to the central persistence server
Upon receiving request(x, v_x) at the central persistence server:
    Central persistence server replies with A, where $a \in A \iff v_a > v_x$
Upon receiving A at replica site:
    $\forall a \in A$, apply a to x

Algorithm 2 details how recovery is performed. The replica sends its current version to the central server. The central server then looks for any action containing this object with a more recent object version. Those missing actions are then sent back to the replica which must apply them in the correct order.

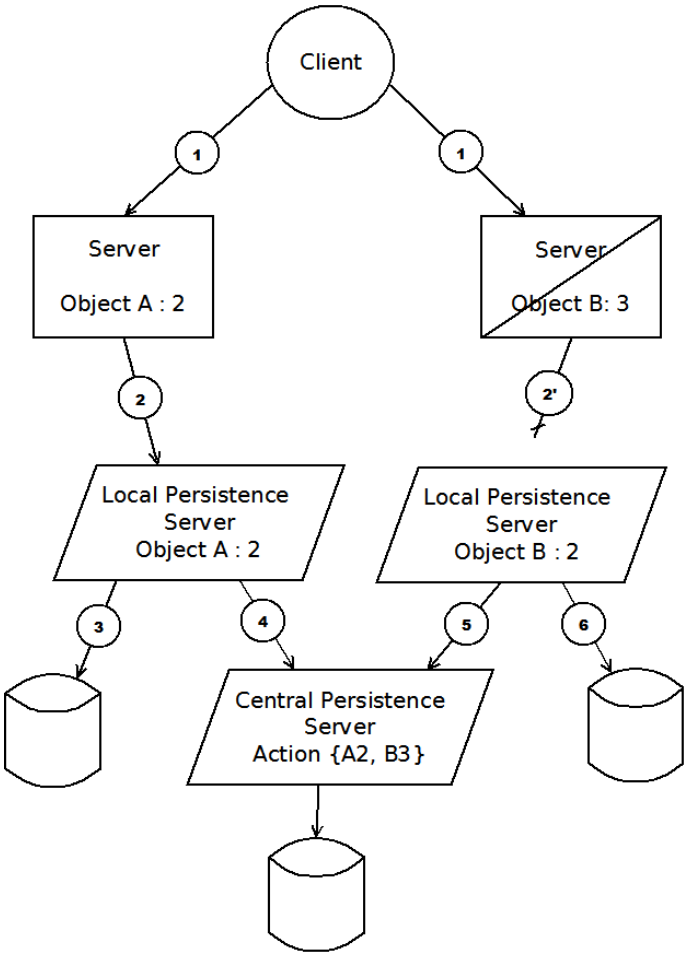

Figure 4–4: Medium consistency persistence

4.3.4 High Consistency

High consistency is treated exactly the same way as medium consistency. The only additional point to consider are the values of critical attributes when replaying an action. If a critical attribute is low or no consistency, then it is possible that the value at the persistence replica differs from the value read at the client's replica when the action was first executed. In this situation, the attribute of the persistence replica

must be set to the read value before applying the action. This is possible because the discrepancies came from a lower consistency action with no replica atomicity requirement and the read value was one possible value a replica could have had so it is acceptable to update the persistence replica with that value. If the attribute is of medium consistency or higher, then the discrepancy must come from an update which was missing at the persistence replica. In this situation, the persistence replica should have received another action to apply (from the central server) before applying the high consistency action.

4.3.5 Exact Consistency

Since the action is treated synchronously, all that is required is for the persistence replicas to be treated as participants. This means the persistence replicas will be updated eagerly. No object versioning or replayability is required since the replicas will be synchronized to the masters.

4.4 Dealing with Lost Actions

Normally if an action is unrecoverable, it is also the last action on the participating objects before their masters fail. Therefore having the persistent replicas takeover while ignoring this action results in no inconsistencies. There is, however, a situation where the missing action is not the last action on an object. Due to asynchronous update propagation, it is possible for a medium or high consistency action, e.g. on an object X, to have committed, but the master has not sent out its updates yet. A second medium or high consistency action now affects the same object (X) and another object (Y) that was not part of the first action. Suppose now that the node holding the master of X crashes before sending out updates for

both the first and second actions, while the master of Y manages to propagate the updates to the central server. Recovering X is now problematic because the first action is unrecoverable but the second action is recoverable. If there is any dependency between the first and second action, it appears that the second action would not be applicable on the persistence replicas. Figure 4–5 shows an example where the first action A, which affects a single object (X) on the left server, has been committed (1). The second action B affects the same object X as A and another object (Y) on the right server. It is therefore applied on both servers (2). The left server crashes before the master of X is able to send the updates for A and B (3), while the master of Y at the right server successfully propagates B to its replicas (3'). Action B can be propagated to the central server (4) and to the local stable storage (5). When the left persistence server needs to recover masters from the left server, it is only able to retrieve action B from the central persistence server (6). Action A is therefore lost.

We want to argue that the lost action can be safely ignored. The reason is that if changes in regard to A were not reflected at the persistence layer, they were not reflected in any other replicas, including the client performing the action B (see the additional requirement shown in Section 4.3.3). If the second action is a medium consistency action, it is acceptable for read values to differ from the masters to the replicas of the client. Since the client did not see the updates of the first action as the persistence server itself does not have the update to A, the outcome of B may have been different from the client's expectations. When recovering, the persistence replicas will now read values prior to A, which will be similar to the client's view.

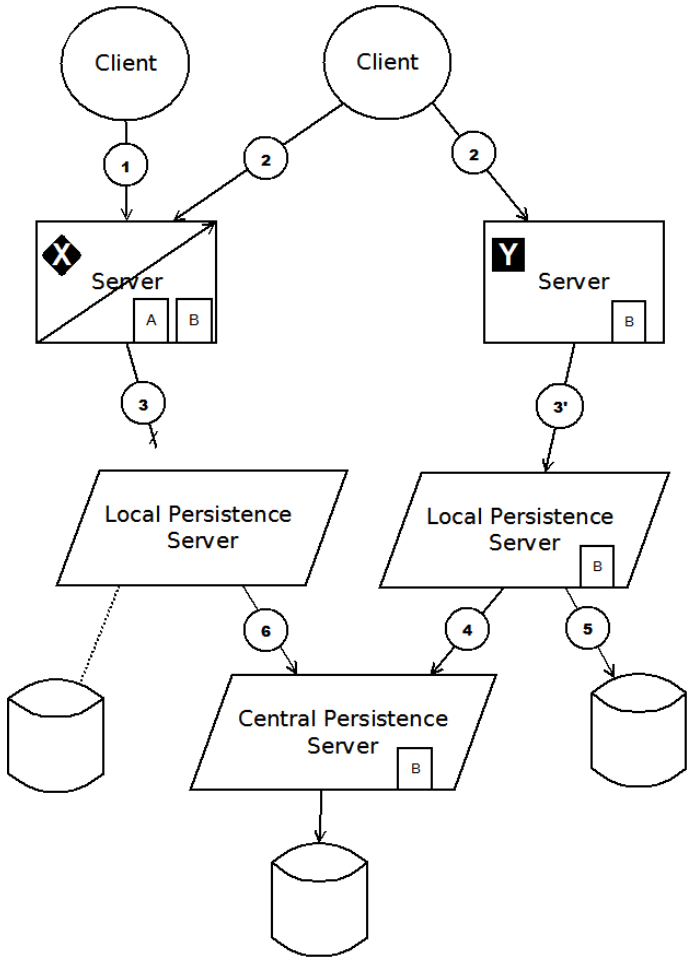

Figure 4–5: Lost action example

If the second action B is a high consistency action, then we must consider the critical attributes. During the original execution of B, the client has not seen the updates of the first action A as mentioned above. Since action A was executed before B, any changes made by A at the master of X would have been considered in the predicate evaluation of the critical attributes of B. Therefore, if A made any

changes which would have caused the critical attributes reads of B to be unacceptable, B would have been rejected and not committed. The situation would have been avoided since B would have not been propagated in the first place to the persistence replica. However, if B is still executed despite the effects of A which are unobserved by the client, it means the outcome of the action is considered acceptable. Therefore reapplying the action B after losing A is possible since A did not make any changes with substantial impact on B, as defined by the critical attribute acceptability criterion.

CHAPTER 5
Action Development and Implementation

Our consistency categories highlight some of the concerns developers must deal with, such as client reads and replica atomicity. The weight of the concerns depend heavily on the complexity of the actions (see Section 2.1.1) and their volume (frequency). Furthermore, consistency categories come with increasing levels of consistency which also incur more overhead on the system. However, the relative performance of each consistency model again depends on the action itself. Therefore, designing and implementing an action can be thought of as a development process [26]. Figure 5–1 is a flowchart showing the important phases in the development cycle of an action:

I Requirements: Requirements for the action need to be collected. In particular, any performance and consistency requirements must be clearly defined.

II Consistency Model Design: Given the requirements, the appropriate consistency model must be applied when designing the action.

III Implementation: The action is implemented within the game.

IV Performance Analysis: The action is tested to meet the requirements.

V Relaxation of Requirements: If the requirements are too stringent for the action to match, they could be relaxed. By doing so, the action can possibly be redesigned with a lower consistency model and yield a more efficient implementation.

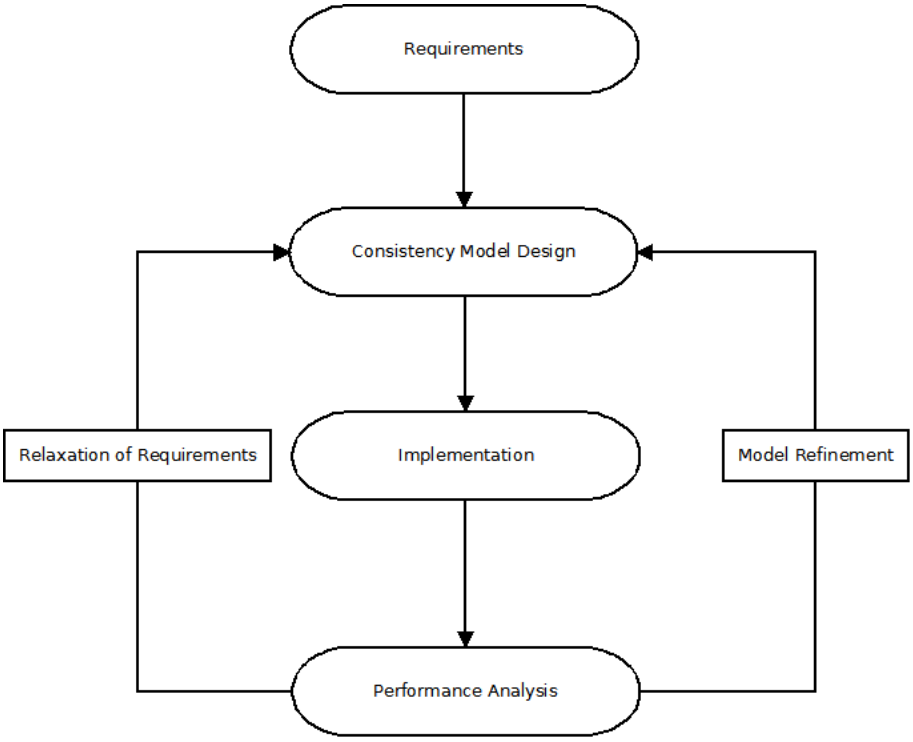

Figure 5–1: Action development process

VI Model Refinement: It is possible to make improvements on the consistency model itself intended specifically for the action, resulting in more improvements possible on the implementation.

Model refinement refers to the fact that certain actions may have very specific requirements which do not quite fit any of the consistency requirements. For instance, an action might require master atomicity but not replica atomicity, which would classify it between low and medium categories. If applying medium consistency is not a viable option, then the model must be redefined to accommodate these more specific requirements. New optimization possibilities can open up with these specialized models.

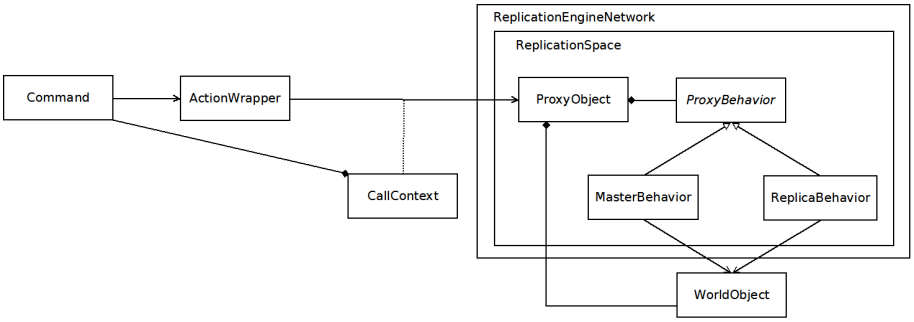

Figure 5–2: Action components

In this chapter, we will present implementations of a wide variety of actions within Mammoth. We will first introduce the basic framework and architecture necessary to support the actions, as well as persistence. We will then go over three actions in detail: "Move Player", "Pickup/Drop Item" and "Activate Item". We will show how we can use semantic requirements to find the appropriate consistency category for each action. This will lead us into the next chapter where our experiments will compare different implementations of the actions using different consistency categories.

5.1 Mammoth Integration

The main features of Mammoth are described in Section 2.3.2. We will first show the structure for action handling and later introduce persistence.

5.1.1 Action Initiation and Coordination

Figure 5–2 shows the interaction between the major components required for creating and executing an action.

- Command: Mammoth uses the Command pattern to encapsulate information necessary from the client to execute an action [14]. Command objects are then locally queued by the client software before execution.
- ActionWrapper: When executed, commands call the appropriate method from ActionWrapper. This client-side code contains the core of the action logic and serves as the coordinator. It is in charge of calling the correct sub-actions on each object, collecting the replies and committing the action. In case of failure, the ActionWrapper is also able to interpret exceptions and notify the user appropriately.
- CallContext: In each call to an object, the ActionWrapper piggybacks a CallContext object, which contains information necessary for that object to execute its sub-action. It is also used by the called object to relay back information. The initial Command data is included in the CallContext.
- ProxyObject: ProxyObjects are in charge of correctly executing sub-actions received by the ActionWrapper. Its behavior depends on whether it is a master or a replica as determined by its attached ProxyBehavior.
- ReplicaBehavior: A ReplicaBehavior allows the replica to perform local reads if desired, but any sub-action with write operations must be sent to the master. Replicas can create a Remote Procedure Call (RPC) message which instructs the master to perform the appropriate method.
- MasterBehavior: A MasterBehavior usually tries to acquire a lock on the master, perform the necessary operations, wait for commit and then propagate

updates to the replicas. In Mammoth, locks are granted on a per-object basis, not per-attribute.

- WorldObject: This is the actual object, used both by replicas and masters, to store the current state of the object and its attributes as perceived by that proxy.
- ReplicationSpace: This is a container for all proxies (masters or replicas) currently held by the node (server or client). The content of the ReplicationSpace is dictated by interest management and distribution. Replicas are added via new interest subscriptions while remaining replicas of objects of no interest are evicted. Load-balancing and fault-tolerance migrate masters from server to server.
- ReplicationNetworkEngine: The network engine handles communication between the nodes. It is capable of publishing updates and sending remote calls between replicas and their master.

For our consistency models, we are mostly concerned with the ActionWrapper, the CallContext and the ProxyBehaviors. The ActionWrapper can vary depending on the level of coordination required. Higher consistency categories differ in that component in order to produce different master atomicity guarantees. ProxyBehaviors can affect the way replicas receive updates. It is therefore of interest to lower consistency categories for replica atomicity. CallContext is used to communicate critical reads (for high consistency actions) and master reads.

5.1.2 Persistence Components

Persistence introduces two types of nodes to the system: local persistence server and central persistence server. Their components and their interaction with the rest of the system are detailed in Figure 5–3.

For most actions, update propagation is asynchronous (see Section 3.1). After the action has committed, the master sends updates to all its replicas, including replicas held by local persistence servers. The *CallContext*, which includes data on the action, is piggybacked on the update message. When a local persistence server has received updates for all of its replicas related to an action, it applies the updates atomically (see Update Propagation in Section 2.2.1). When doing so, the *Object Listener* of each replica is notified and stores the attribute changes on the *Local stable storage* using the appropriate strategy for the action type. The action as whole also notifies the *Action Listener* which sends the action data (CallContext) to the *Central stable storage* for medium and high consistency actions.

In case of failure, the *Fault-Tolerance Layer* of a game server must initiate takeover of failed masters. The fault-detection mechanism and the reassignment of the masters is outside of the scope of this thesis. Once the set of failed masters has been determined, the server must request the central persistence server for a copy of the masters. This request is forwarded to the *Persistence Recovery* module of the local persistence server in charge of monitoring those objects. For each object, the local persistence server sends a request for missing actions to the central persistence server. Upon receiving those missing actions (if any), the persistence employs a special *Persistence Action Wrapper* which can replay actions for the target object.

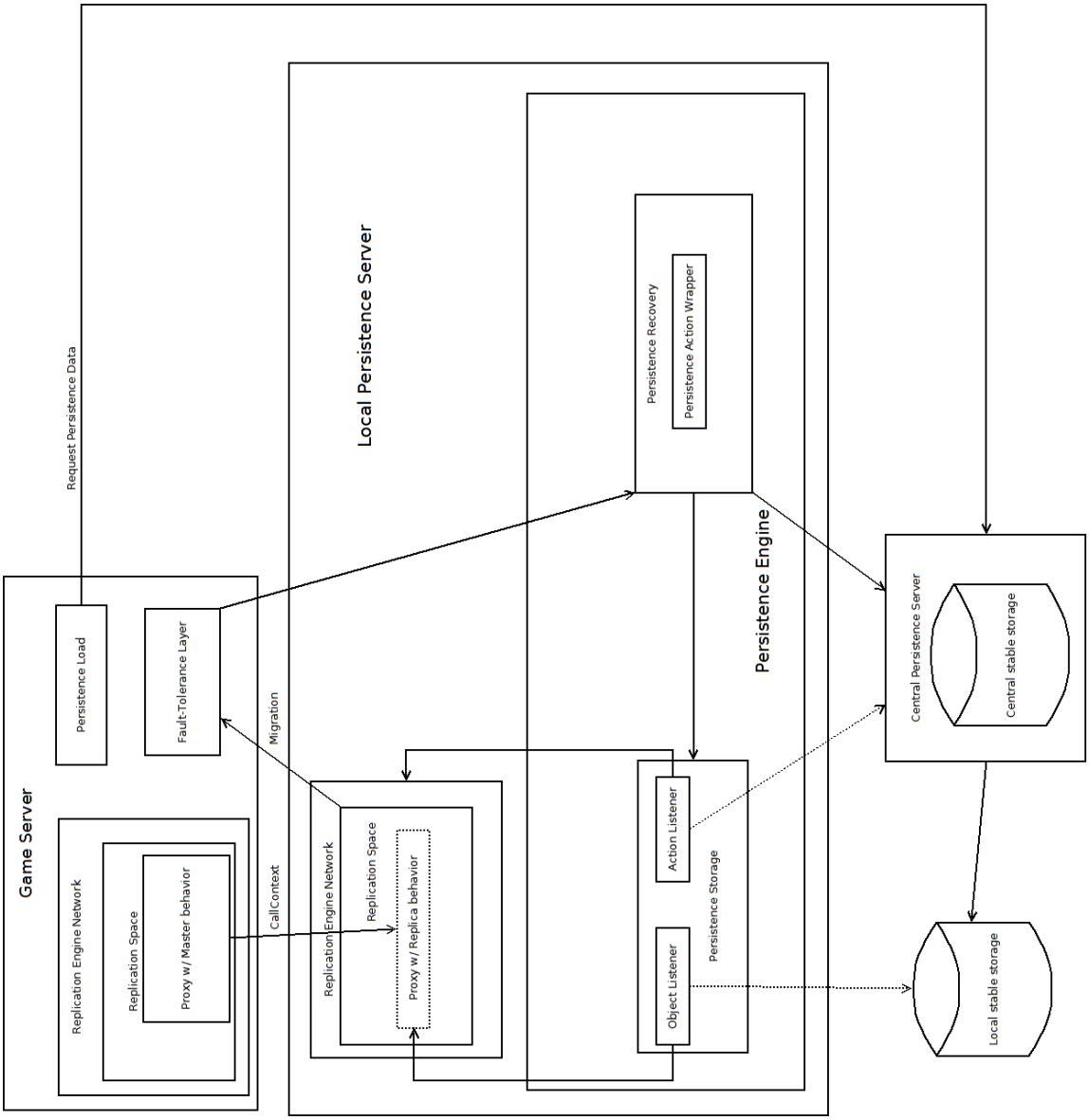

Figure 5–3: Persistence components

The restored object is then migrated to the game server, which can now use it as master.

System startup. The central persistence server must be started before any local persistence server since it is in charge of coordination. The local persistence servers are then started in order to access the data of their stable storage. Once the persistence layer is running, the game servers can be started. The game servers can request objects from the persistence data. Furthermore, the central persistence will assign each game server to a local persistence server. That local persistence server is then in charge of declaring its interest radius as the cells owned by that game server. Therefore, each local persistence server contains replicas for the masters of its monitored server. In other words, each local persistence server is in charge of monitoring objects from a single game server and storing actions involving any objects from that server. Currently, the central persistence server only does one-to-one assignment between game and local persistence servers.

Stable storage. The complete initial state of the world is stored in an XML file called a "map". This includes *immutable* elements which are not to be monitored by the persistence layer (see Section 2.3.1). It is loaded first when the whole system starts. Persistence data is loaded subsequently from local persistence servers to bring this initial state up-to-date.

Persistence data is stored in relational databases. Local persistence servers employ a two-layered database structure [34]. A generic `Object` table stores the entire XML representation of each individual object. Additional tables are then used for specific attributes which are frequently updated, such as position. Changes to

attributes which are not specifically represented must be captured by serializing the whole object and storing them in the `Object` table. When an object is restored from the stable storage, it is first constructed from the XML data. Then, any attribute which is stored in its own table will be updated to the latest state.

The central persistence server uses a relational database to store actions in serialized form. An additional table is used to link each action with the version of the objects involved. This is used when comparing a replica version with an action version to determine missing actions (see Section 4.3.3).

5.2 Action Implementation

Concrete actions have been implemented within the presented framework. Those actions have been chosen to provide suitable references to a variety of actions commonly supported by MMOGs. For each action, we will provide a description, motivation for choosing this action and requirements. Although we will discuss which consistency category is suitable for each action, multiple implementations following different consistency categories will be given for performance comparison purposes.

5.2.1 Player Movement

Player movement is one of the most basic functions available to the client in MMOGs. Movement usually comes in two types: direction-based and destination-based. Direction-based movement allows the player to move continuously in a given direction as long as the client desires. This is for instance done by controlling the player's character with the keyboard arrow keys. Destination-based progressively moves the player's character towards the target destination. This target destination is usually selected by clicking on the location with the mouse.

Mammoth uses destination-based movement. Selecting a destination is considered an action, which is sent to the master. The master sets the destination and publishes the destination to all of its replicas. Once a master has a destination, the server holding the master computes at regular intervals the next location based on the master's current location and its destination. If necessary, path finding is used to find the correct path to the destination. If at any time during the movement, the master is blocked and cannot reach the destination by following the given path, movement is stopped.

Player movement is a relatively simple action since it only affects two attributes (destination and position) from the same object. Position updates do read the position of other objects and checks for collisions to determine whether the destination is reachable. However, Mammoth only supports collision detection with immutable objects, thus eliminating the need to read remote objects. This makes lower consistency categories suitable since master atomicity is not a concern. Client reads can be a problem, as players tend to move towards locations of interest. If the state of objects around the intended destination changes, the player may want to move somewhere else. However, player movement is easily repeatable, as players can change their destination at any time. Therefore, they can compensate for those stale client reads by moving towards another destination even if the current destination has not been reached yet.

This makes low consistency the logical choice for player movement. This model allows for bounded approximation on the replica attributes. In this case, we use dead reckoning to locally simulate character position at the replicas. This eliminates the

need to update positions completely. Since player movement is deterministic, the path computed is the same at the replicas and the master, barring numerical errors and latency. This will be compared to a medium consistency implementation where every position update is published, and a high consistency implementation where position is considered a critical attribute when setting destination.

The implementation for the low consistency action consists of the following steps:

I The player selects a destination.

II This destination is sent as a remote procedure call (RPC) to the master. The client does not need to wait for a reply.

III The master sets the destination and updates all replicas with the destination and the master's current position.

IV Replicas update their position and set their destination.

V At regular intervals, the master and all the replicas locally update their position to progressively advance towards the destination. If the path is blocked and the player cannot advance, movement stops.

VI When the destination is reached, movement stops.

Therefore, apart from the initial exchange of destination, there is no further communication from the master to the replicas. Here are the medium and high consistency implementations:

I The client selects a destination.

II This destination is sent as a remote procedure call (RPC) to the player master. The client will wait for confirmation from the master. For high consistency, position read at the replica's is also added to the message.

III The master sets the destination and updates all replicas with the destination and the master's current position. The master also sends a confirmation to the client. For high consistency, the master also rejects the action if the position read at the replica is not within an acceptable threshold of the master's position.

IV Replicas update their position and set their destination.

V At regular intervals, the master updates its position and sends an update containing the position to all replicas. If the path is blocked, the master stops moving and sends an update to replicas to cancel the destination.

VI Upon receiving the position update, a replica will set its destination. It does not perform any dead reckoning.

VII When the destination is reached, movement stops.

At the persistence level, position updates can be logged with a distance-based approximation strategy for the low consistency implementation [34]. For the high and medium consistency implementations, position changes are expected to be stored exactly. However, the action does not need to be sent to the central persistence server since master atomicity is not a concern. Model refinement is also possible here for medium and high consistency. Although the client must perceive the correct position while playing, this is not necessarily the case after recovery of the game. Clients are able to tolerate some reasonable bounds on its position [34], which suggests the use of approximation strategies even when storing player movement position for the medium or high consistency implementations. Our implementations will not consider this refinement and exact updates are used for medium and high consistency.

Moreover, additional model refinement is possible on the destination attribute at the persistence level. After a system restore, the players do not expect to continue moving towards their destination selected before the game was shut down. In fact, clients expect their player to be still when entering the game. Clients can easily re-establish their destination if they wish to do so. Therefore, destination is not an attribute that needs to be reflected in the persistence data, which mirrors a no consistency approach.

Note that player movement is not the only way of updating the position attribute. MMOGs frequently support other means of transportation which may require a different level of consistency.

5.2.2 Pickup/Drop Item

Pickup/Drop Item is a pair of actions commonly found in MMOGs. Picking up objects represents one of the primary ways of acquiring items in MMOGs since many events occurring in games result in objects being generated on the ground for players to take. A client can pick up an object by clicking on it with its mouse. If the object is too far away, the client will first send a move action to put the player closer to the object. Otherwise, the object will be picked and put in the player's inventory. Players can drop items from their inventory, which is accessible by the player at any time. When an item from the inventory is selected, it is first transferred to a state called the "hand". This transition is not reflected in the game state and is simply an intermediate step at the interface level. The drop action is only initiated after the player has dropped the object from the hand to a selected location on the ground.

Figure 5–4: Objects and inventory

Figure 5–4 shows an example from Mammoth of a flower on the ground and a tomato in the player's inventory.

In Mammoth, both the player and the item are affected by a pickup or drop action. Inventories have a maximum capacity which limits the number of items. Inventories are also limited by a maximum weight attribute, with each item having a different weight. Those restrictions must be verified when a player tries to pick up an item. If successful, the player can then add the item to its inventory. The object itself must keep a reference to its location (either on the ground or in an inventory).

Pickup action should be a medium consistency action. Since objects on the ground are visible to all players, concurrent pickup requests are possible. Master atomicity is therefore required to ensure consistency between the players and the

items. We want to avoid situations where two players list the same item as being in their inventory. Replica atomicity is also important to ensure clients perceive the correct objects on the ground to avoid illegal pickups. Client reads is less of a factor because the drop action can reverse the effect of a pickup with almost no consequence. High consistency is therefore not necessary.

Drop action has lower requirements than pickup. This is because objects in the inventory are not visible to other players and can only be interacted with by the person who owns the object. Therefore, concurrent requests are not possible. Nevertheless, exact updates must be sent to replicas so that clients can all observe that the object is now on the ground, so that they can have the opportunity to pick it up. Furthermore, the drop action requires persistence to maintain master atomicity in the presence of failures. Otherwise we would get inconsistencies such as an inventory containing an object which is already on the ground, or an object still belonging to an inventory which does not contain it anymore. The second inconsistency in particular is not repairable by the players since the item is not visible on the ground nor in an inventory. Therefore the drop action must also be treated as a medium consistency action.

Both pickup and drop actions are implemented in the medium consistency category using the linear locking algorithm described in Section 3.2.2. Here are the steps of the medium consistency implementation:

I The client decides to pickup or drop an object.

II For pickup, the current location of the object is recorded. If the player is too far away, a movement action is initiated first.

III For drop, the location where the client wants to drop the item is recorded. If the player is too far away, a movement action is initiated first.

IV The client sends an RPC request to the item master, which includes the player's ID. The client waits for a reply from the item master.

V The object acquires a lock. If it fails, the action is rejected immediately. If it succeeds, the item verifies that the action is valid. For pickup, this means the item is still on the ground. For drop, it means the object is in the player's inventory.

VI If it passes validation, the item sends a successful reply to the RPC message after applying the action on the item master. For pickup, the weight value of the item is returned to the client. The item keeps the lock and waits for a reply from the client.

VII The client now sends a RPC request to the player master. The player tries to acquire a lock. If it succeeds, it will validate the action. For pickup, it will verify that there is enough space under the maximum weight to hold the item, based on the item master's weight read. For drop, it will verify that the item is in the inventory.

VIII If successful, the player master sends an RPC return and commits immediately, publishing an update to replicas. The client then sends a commit to the item, which can now publish an update.

A low consistency design for pickup is also implemented. In order to reduce the number of message rounds, we verify weight restriction without requiring master reads on the player (see Section 3.2). This is done by reading the player's character

replica held by the item master's server. Doing so allows saves the final commit confirmation sent back to the item since we already know the action will be successful at the item master, thus saving one message round and shortening the lock on the item. This, however, requires tolerance on weight inconsistencies or tolerable bound on the staleness of the weight attribute. In Mammoth, weight is currently an immutable attribute, thus stale weight reads are not possible. However, we have still implemented both the low and medium consistency actions for comparison.

Step-by-step, the low implementation of pickup and drop work the following way:

I The client decides to pickup or drop an object.

II For pickup, the current location of the object is recorded. If the player is too far away, a movement action is initiated first.

III For drop, the location where the client wants to drop the item is recorded. If the player is too far away, a movement action is initiated first.

IV The client sends an RPC request to the item master, which includes the player's ID. The client waits for a reply from the item master.

V The object acquires a lock. If it fails, the action is rejected immediately. If it succeeds, the item master verifies that the action is valid. For pickup, this means the item is still on the ground. For drop, it means the item is in the player's inventory.

VI In addition, for pickup, the item master reads the maximum weight and capacity of the local replica of the player's character. If the weight and capacity restrictions are passed, the item immediately commits and publishes updates,

sending a success message back to the client. For drop, the player's replica is read to verify that the player lists the item as being part of its inventory. The item can then immediately commit, publish updates and a confirmation to the client.

VII The client now sends a RPC request to the player master to tell it to add the item to its inventory (or to remove it from the inventory). The player master does not do any verification, even if the item puts the inventory overweight for pickup, or if the player never had the item in its inventory for drop. The client does not wait for any reply from the player master.

For persistence, this low implementation will not guarantee master atomicity. The action will not be sent to the central persistence server. We will compare the low and medium consistency implementations.

5.2.3 Activate Item

Our third action is item activation. MMOGs often contain objects which can be activated by the player to trigger certain effects. In particular, one special mechanism included in most MMOGs is called "Area of Effect (AoE)" [20]. State changes with AoE will affect all characters within radius. We wish to study how AoE relates to our consistency categories; our item activation action will therefore have AoE. In Mammoth, item activation is an extension of the drop item action; in addition to dropping the item, all characters within the specified range of the item's dropped location will increase their maximum weight by one (see the radius around the flower in Figure 5–5).

Figure 5–5: Item activation Area of Effect radius

Item activation with AoE is normally a medium consistency, if not high consistency action. Clearly master atomicity is required, since the action requires that all characters within range are affected. It could also be considered high consistency, since the characters' position is an important consideration when deciding on this action. The client initiating the action not only expects that all perceived characters within the radius to be affected, but also that all perceived characters outside of the range are not affected. Therefore, the client's expected list of affected players must correspond with the actual list of affected players. Notice that the exact position of each character is not necessary, only the information about being in or outside of the AoE range matters.

Implementing those models can be expensive. If there is no bound on the staleness of player position, which is likely if sudden position changes are allowed, the action must obtain master reads of the position of all objects in the game. This is the only way to ensure that the action is completely consistent. For Mammoth, since the only way of changing position is through player movement, we make the reasonable assumption that only players within the interest radius of the client can be affected by the action. The medium and high consistency implementations therefore have the following steps:

I The client decides to activate an object by dropping the object.

II First, the drop action is initiated and all the steps of the drop action are executed, except for the final commit.

III For each player in the client's interest radius, the client sends an RPC request to the corresponding master. The RPC request contains the location of the action effect, which is the item's dropped location, and the radius of the action. After each RPC request, the player waits for a reply before proceeding to the next player. In the high consistency implementation, the client also locally computes whether that player is in range of the action. When sending the RPC request, an additional Boolean variable is included to indicate whether the client expects that player to be affected or not.

IV The player master tries to acquire a lock and verifies if it is in range of the action. If it is, it increases its maximum weight by one. If it is not, it does nothing. In both cases, the player master must send a reply back to the client and wait for commit. For high consistency, the outcome decided by the player

master must agree with the expected outcome. Otherwise the action is aborted immediately.

V Once all players have successfully executed their sub-action, the client can now send a commit to every locked player and the item.

Persistence-wise the action follows the traditional medium/high consistency model; actions must be sent to the central persistence server for replayability. The action will contain the list of players which are affected by the action.

A possible low consistency implementation can decide from the client's view which players will be affected. This is done by reading the replicas. After sending the sub-actions to the selected objects, their master must then verify that they are in actual range of the effect. Only objects who are truly within the radius will be affected. The result is an outcome which is neither completely consistent with the player's view nor the masters' view. It allows players expected by the client to be in range not to be affected, but it does not allow players not expected by the client to be affected. The advantage of this design is efficiency of implementation: the involved masters are decided client-side. The low consistency implementation works as follows:

I The client decides to activate an object by dropping the object.

II First, the drop action is initiated and all the steps of the drop action are executed, including the final commit.

III For each player in the client's interest radius, the client computes locally whether that player is in the AoE range of the action. If it is, it sends a

RPC sub-action to the player master. The client does not wait for any replies for those RPCs.

IV Each player master verifies whether it is in the range of the action or not. If it is not, it ignores the sub-action. Otherwise, it increases its maximum weight by one.

This implementation is stored under standard low consistency persistence rules, with no action data. We will compare the low consistency implementation against the medium and high consistency implementations.

CHAPTER 6
Performance Evaluation

The implementations presented in Section 5.2 are now compared. Evaluating the performance of each implementation allows developers to determine if it is adequate in fulfilling the requirements. If not, the developers either have to relax their consistency requirements to use a lower consistency model, or to refine their current model and optimize further.

6.1 Experimental Setting

Our benchmark application is the Mammoth framework. The network engine uses a publisher/subscriber system built on top of Apache MINA[2]. The network engine (called *Stern*) employs a single hub to which all network messages are forwarded [24]. The hub runs on a separate machine from the server(s).

Non-Playing Characters (NPC) clients are used to generate load [30]. They will be instructed to continuously execute the target action being tested. Each NPC client uses a distance-based interest. The threshold is 6.0, which corresponds to the whole screen for a regular client. In other words, NPCs are subscribed to all objects in the visible area of a regular client's screen.

A MySQL database is used for persistence. Every persistence server is connected to the same database, but for the purpose of these experiments they do not share any data. For each experiment, the number of persistence servers is equal to the number of game servers.

Clients and servers are ran on various machines in a local area network. Although the system is heterogeneous, the bulk of the machines are Pentium Core 2 Duo 2.4GHz machines. Each Mammoth application is allowed 512MB of Java heap space memory.

Specific details on the setup is given for each action. Statistics are collected using the MINA StatCollector and SIGAR [1].

6.2 Player Movement

For player movement, the NPCs used are called "wanderers". They move around the world continuously, changing their destination every 1-2 seconds. NPCs do not use path finding; they only move in straight lines.

We will first compare the rate of messages and throughput. We first focus on the actions without persistence support. Since distribution is not a factor here as actions only involve a single object, the experiment will run on a single server. High consistency has been set to accept very large differences to maximize the number of successful actions and provide a suitable comparison with the other implementations.

Rate of messages. There is a clear difference in the rate of messages between low and the two other implementations (see Figure 6–1). This is because players update their position approximately 10-15 times per second. The rate of position updates vary due to the non-deterministic nature of the NPCs, which are not constantly moving since there may be a delay before selecting a new destination after reaching the current one. In medium and high implementations, each of these position changes results in an update to all replicas. In contrast, the low implementation only updates when the NPCs set their destination (since dead reckoning is

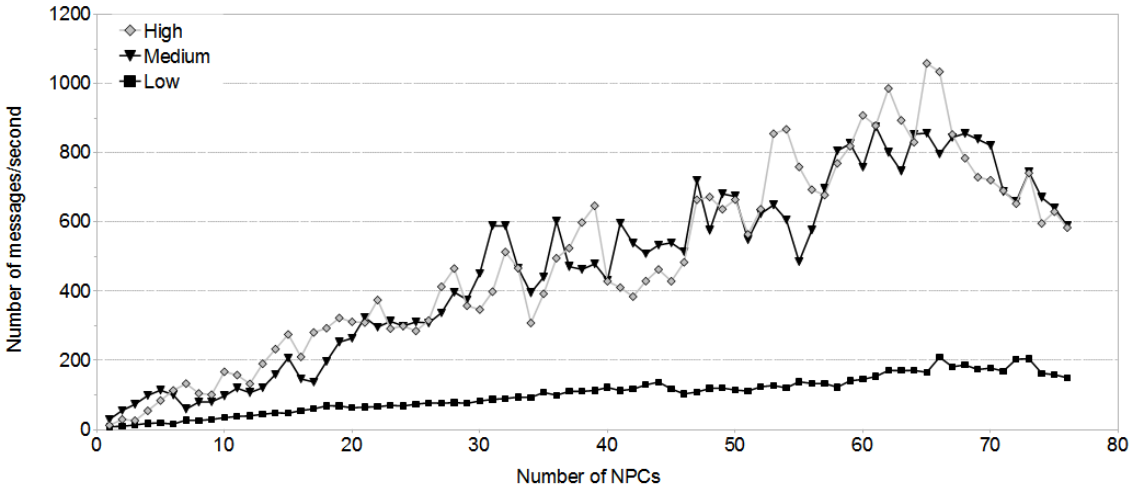

Figure 6–1: Movement: Rate of messages for an increasing number of NPCs

Implementation	Size (Bytes)
High	875.48
Medium	873.08
Low	1061.2

Table 6–1: Movement: Messages size per implementation

performed at the replicas), which is only 1-2 times per second. At around 900 messages per second, the server started to drop messages which resulted in performance degradation for the medium and high implementations. This occurs at roughly 60 players.

Throughput and message size. The throughput graph (Figure 6–2) shows similar performance. Messages for the low implementation have an average size of 1061.2 Bytes (see Table 6–1). This is because the majority of the messages sent are requests to set destination. In contrast, the lower sizes of 873.08 Bytes and 875.48 Bytes for medium and high implementations respectively are attributed to the position updates messages, which are smaller than "set destination" requests. This also

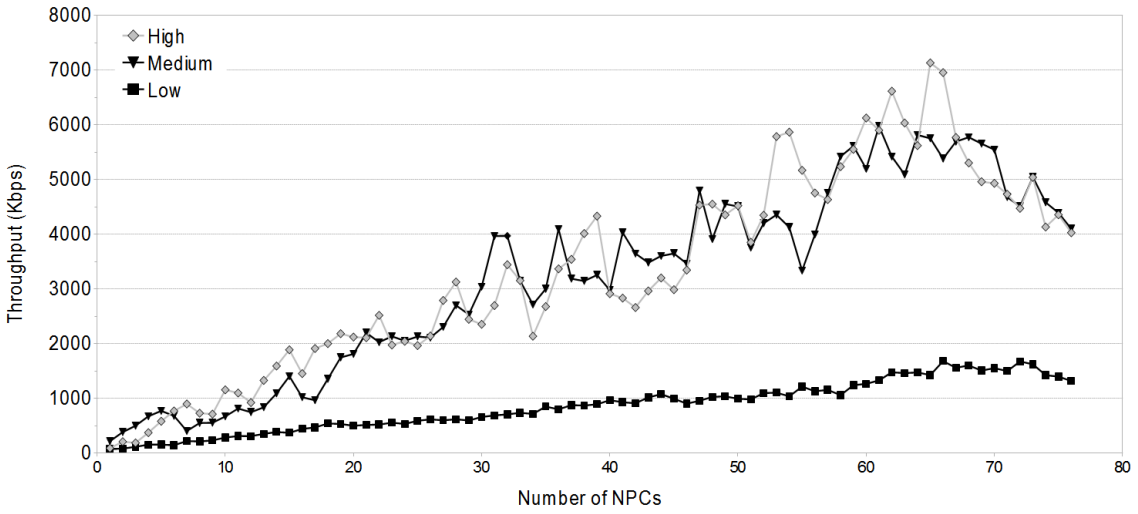

Figure 6–2: Movement: Throughput for an increasing number of NPCs

explains why there is no significant difference in size between the high and medium implementations. Although the high implementation "set destination" messages include the value of the read position, the relatively low number of such messages compared to position updates (which are the same size for both implementations) does not increase the average message size.

CPU load. At the client-side, the CPU load fluctuates between 0.2% and 0.6% for all three implementations at any number of clients. This indicates that the extra load required on clients to locally update positions for the low implementation is not significant or is matched by the higher number of updates to process by the two other implementations. Please note that path finding is not used in this experiment, which can make local computations more expensive. At the server-side, the CPU load reaches 14% at both the medium and high consistency, but only 10% at the

low consistency implementation. This can be attributed to the higher number of messages to process.

Persistence. We also run the experiments with added persistence support. The low consistency implementation uses a distance-based approximation strategy [34], while both the medium and high consistency implementations use an exact strategy. This exact strategy means every position update results in a database update which is around 15 updates/second. The low consistency implementation only triggers 1.5 update/second with a bound of half the game screen. In other words, position updates are sent to the local persistence database only when the actual character position differs by more than half a game screen from the stored position. Although the players are moving continuously, they are not necessarily moving across long distances, which keeps the number of necessary updates for a distance-based strategy rather small.

Summary. There is a large performance gap between low and the two other implementations. The reduced number of updates both in the network and persistence improves the scalability of the system. The additional client load remains negligible.

6.3 Pickup/Drop Item

Since multiple objects are involved, distribution becomes a factor. We will therefore test under a varying number of servers. Since picking and dropping are limited by the number of pickable items in the game (in our case, there are 25 items in the game), throughput and rate of messages are not significant metrics since the rate of actions is bounded by game logic. Instead, we investigate the average

execution time for pickup and drop actions. Since the implementations differ in their communication with masters, we expect to see a difference in performance. The execution time is measured from the time the action is initiated at the client to the time the action returns at the client. In other words, the execution time is from the perspective of the client; the player cannot initiate any new actions while an action is currently being executed.

The NPCs used are called "Random pick up and drop to the ground" NPCs. These NPCs will move around randomly, similarly to the wanderer used above, until they find an object. They will then have a 75% chance to move to the object to pick it up (or a 25% chance to ignore it and continue moving). Once they have an object, they have a 75% chance to drop it on the ground (or a 25% chance to continue moving). We are using the low consistency version of movement.

Execution time. Experiments show that the execution time stays stable for an increasing number of players in the system for both implementations. Even though the number of concurrent requests increases, any request that fails to acquire a lock on an object is immediately rejected instead of waiting. Figure 6–3 compares the message delay for the low and medium implementations of pickup and drop using a single (1) server and a four (4) servers setup. The four servers setup uses static partitioning of the world, meaning the world is divided into four rectangular cells, each server being responsible for the masters of their cell. A master is migrated from one server to another whenever its object's position crosses to a different cell. Persistence is not included for now as it will be dealt with separately.

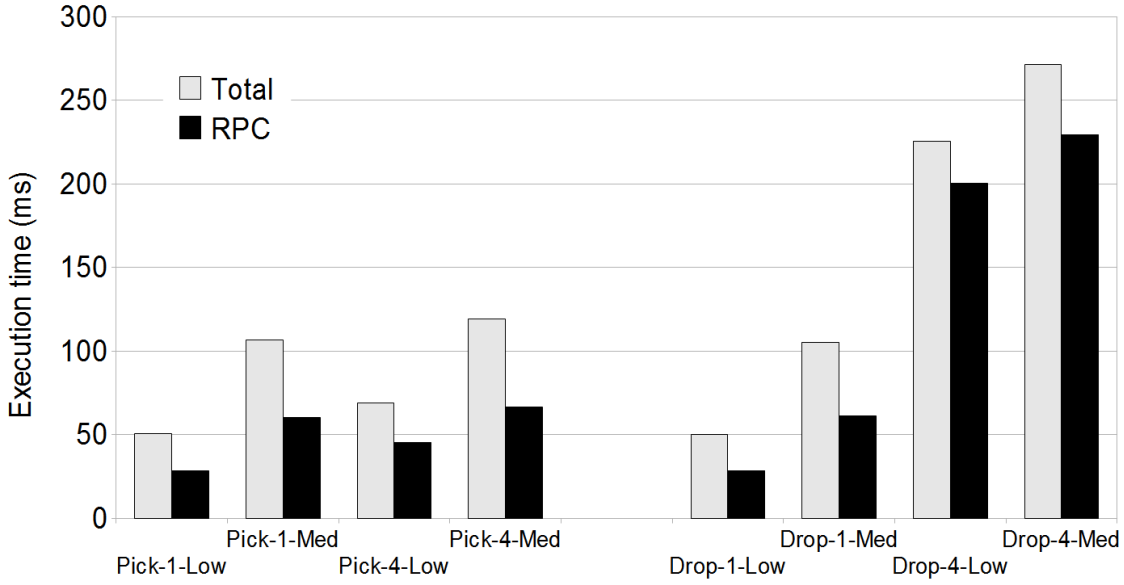

Figure 6–3: Pickup/Drop: Average execution time

RPC (Remote Procedure Call) refers to the time spent executing a remote procedure call and receiving a reply. This measures the time from the moment the request is sent to the moment a reply is received. In the low consistency versions of pickup and drop, the actions send and wait for one RPC call. Since master atomicity is not guaranteed, these actions do not wait for confirmation from the player; as soon as the item confirms that the action is possible to the client, the action is considered complete. For the medium consistency versions, the actions send and wait for two RPC calls. To preserve master atomicity, the client needs confirmation from both objects before sending out commit requests. Therefore, it must wait for the replies of two masters. The figure shows that waiting for RPC replies constitutes a large part of the execution time. Implementations at the medium consistency level have

longer execution times than their low level counterparts due to the increased RPC reply waits.

Figure 6–3 also shows that the performance of pickup and drop actions are almost identical in the single server scenario, which corroborates with the fact that both actions are very similar. However, in the distributed case, the drop action is much longer than the pickup action. This is due to actions being executed on objects while they are being migrated between servers. These actions suffer a heavy penalty since they have to wait for the migration to be completed before proceeding. In our setup, when a player moves to the cell of a different server, not only is the player migrated, but so do all the items the player is currently carrying. Therefore, since pickup involves an item which was already on the ground (and thus cannot be migrating), the only way the action can suffer the migration penalty is if the player is currently moving. For drop, since the action can involve items the player is carrying and can thus be migrating, it is more likely to incur the additional delay.

In our tests, approximately 1% of the pickup actions measured in the four servers case had an average execution time of 2000ms, while the rest were similar to the single server case ($\sim$50ms for low, $\sim$100ms for medium). In contrast, approximately 9.2% of the drop actions measured in the four servers case had an average execution time of 2000ms, while the rest were around the single server average. This suggests that the more migrating objects an action involves, the longer its execution time.

CPU load. The choice of action implementation has no noticeable effect on the CPU load both at the server(s) and clients. The NPC CPU load is on average

0.72%, regardless of the implementation used or distribution. The server CPU load average is 3.9% for the single server case, and 1.8% for the distributed case.

Message size. The average message size for both actions combined is very similar across implementations, with 1015 Bytes for medium and 1000 Bytes to low. The additional size is attributed to the master read values passed for verifying the weight restriction.

Persistence. We also tested the same implementations with persistence support under the same condition. We want to determine whether persistence will add any overhead to the medium consistency actions. Low consistency actions are not affected by persistence since no additional data is required; only an persistence replica for each master is needed. With regards to server CPU load, there is an increase of 0.5%. The average message size for medium consistency pickup and drop increases to 1073 Bytes to take into account the action data. No significant change in execution time is perceived, since the persistence layer is only contacted asynchronously along with other replicas.

Summary. The difference of performance between low and medium consistency is perceivable in the form of longer execution times and is largely in part due to the additional number of master replies required. The drop action is also affected in the distributed case by migrating objects, although this issue is specific to our testing environment. Persistence adds no significant overhead to the system.

6.4 Activate Item

Activate Item is tested under the same conditions as pickup and drop actions. The same NPCs are used to generate the volume of actions. Experiments are ran on

a single server and then on four servers with migration as before. The same metrics used for pickup and drop are used here. Namely, the execution time is the most significant due to the varying number of master replies between implementations. We first test without persistence, which will be considered later.

Impacts of high consistency. The impact of high consistency relative to medium consistency is minimal. The message size increase is not noticeable, since it only contains one additional Boolean attribute. It has no perceivable effect on execution time or CPU load. It will therefore not be included in the rest of this analysis since its performance is almost identical to the medium implementation.

Execution time. In our implementations, the high consistency action sends a sub-action to all players within the interest range of the player initiating the action. In the low consistency version, the player locally checks the position of all character replicas it currently has and sends a sub-action only to eligible players. Therefore, the number of RPC request messages sent is variable and depends on the number of players being contacted.

Figure 6–4 shows the average execution time for the low and medium implementations on a single server per increasing number of players involved. A player is considered involved in the action if a RPC message has been sent to the player. Because the medium implementation waits for replies from each RPC message, the execution time clearly increases with the number of players. The low implementation can also send a large number of RPC messages, but it does not wait for replies for any of them. Thus, its execution time stays stable.

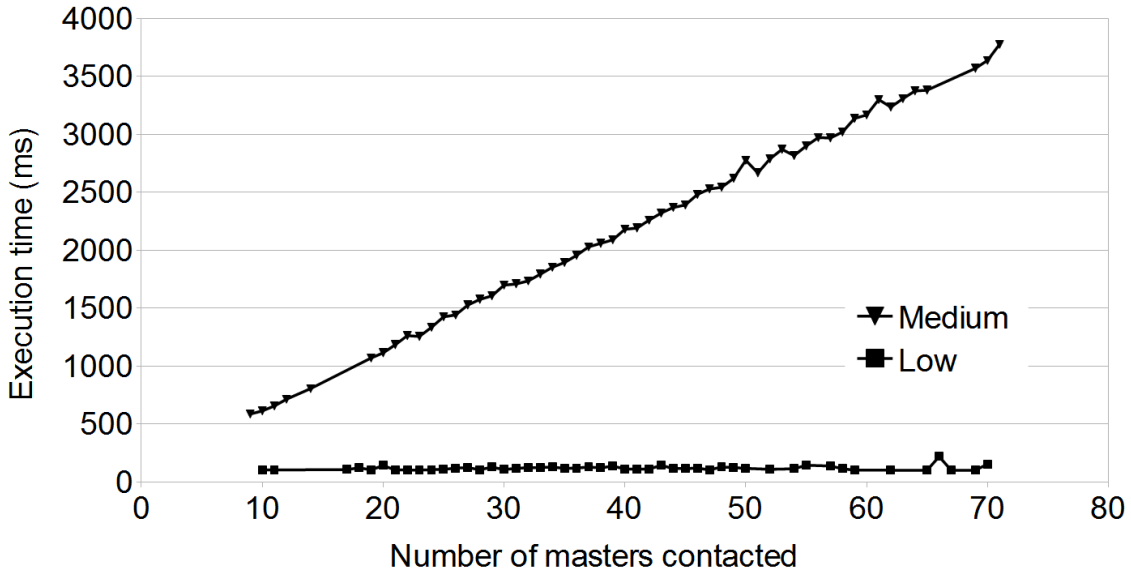

Figure 6–4: Activate Item: Average execution time for an increasing number of players involved

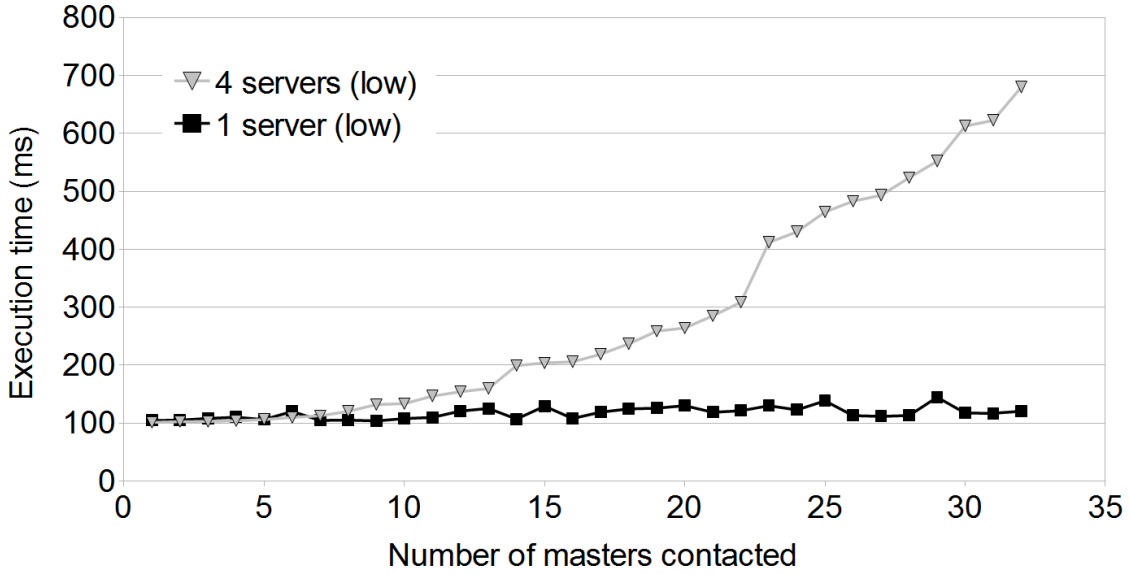

Figure 6–5: Activate Item: Average execution time for the low consistency implementation

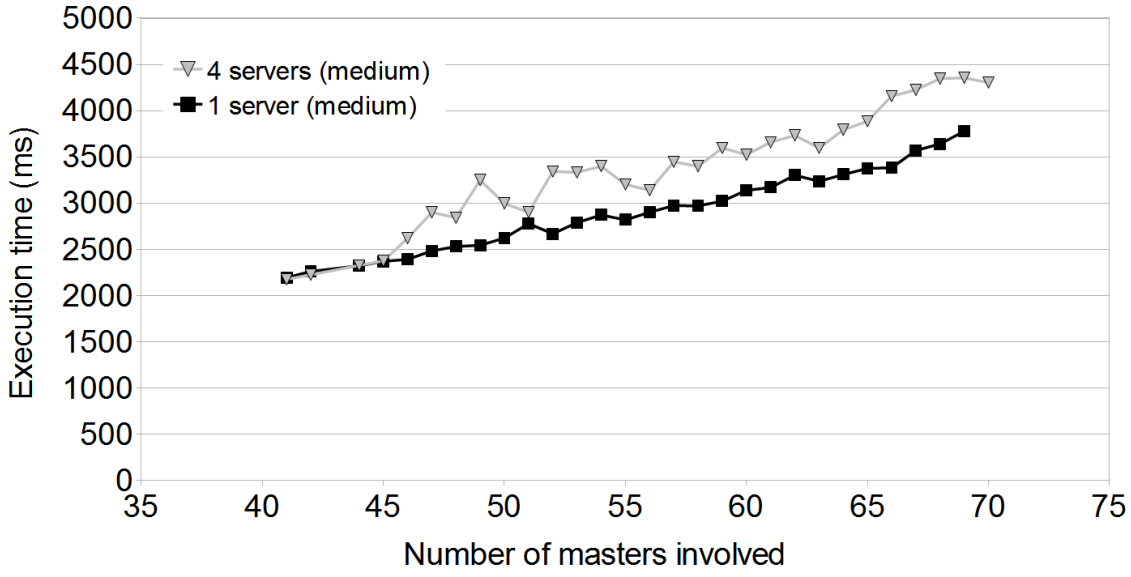

Figure 6–6: Activate Item: Average execution time for the medium consistency implementation

The effect of distribution for each implementation is assessed. Figure 6–5 compares the average execution time for the low consistency implementation between the single server and the four servers setup. The delay incurred by accessing migrating objects has an impact on the distributed case. Since players can migrate at any time by moving between the cells of different servers, an action which involves a greater number of players has a greater chance of encountering migrating objects. Therefore the execution time increases with the number of players involved. Notice also that the migrating penalty is applied even if the low consistency implementation does not wait for RPC return messages. This suggests that the delay occurs at the time of sending the RPC request.

The same analysis is now performed for the medium consistency implementation. Figure 6–6 compares the average execution time for the medium consistency implementation between the single server and the four servers setup. The penalty for migrating objects is still present in the distributed case, but its impact is relatively less than for the low consistency implementation. This is because the medium implementation is already quite expensive, so the penalty does not skew the results as much.

CPU load. The CPU load at the client is on average 1.14% regardless of the implementation or the server setup. At the server(s), the load is 6.86% for a single server, with no significant differences between the implementations. In the distributed case, the CPU load is lowered to 4% since each server has less objects to manage.

Messages size. The message size for all of the implementations are similar, being on average 1023 Bytes. Both the low and medium implementations send exactly the same RPC requests to the character masters.

Persistence. The overhead for persistence support has been measured. In terms of CPU load and execution time, there are no significant differences. The message size for the medium consistency implementation increases to 1083 Bytes to accommodate the action data.

Summary. Due to the variable effect of the action, the execution time depends on the number of players involved in the action. The medium consistency action is much longer and less scalable than the low consistency one. However, the impact of migration is relatively lessened on the medium consistency implementation relative

to low consistency. Furthermore, the overhead introduced by high consistence has very little impact on performance. As for persistence, its impact is mostly based on the increase of message size.

6.5 Results Analysis

The results support the discussion presented in Section 5.2 concerning the choice of suitable consistence category for each action. The specific properties of each action really define what consistency level is required.

For actions with a large volume, such as movement, limiting the message rate is crucial for scalability. Thus, this makes dead reckoning, a low consistency mechanism, particularly suitable for movement as it eliminates the need to send position updates to replicas.

For actions involving multiple objects, the execution time is important because it represents the client's perceived delay after initiating an action. This execution time mostly consists of delays waiting for replies for RPC requests sent to masters. Therefore, performance can be improved when we reduce the number of replies needed from masters. In other words, actions are shorter when they have fewer message rounds. However, medium or higher consistency models increase the number of message rounds to ensure master atomicity and provide master reads. Therefore, it becomes a direct trade-off between consistency and performance.

For the pickup/drop actions, since the number of objects involved is low, a medium consistency implementation is adequate since the number of message rounds is small. However, an action like activate item which has an Area of Effect can involve a larger number of objects, for which it is not acceptable to lock each of them for an

extended period of time. It must therefore be implemented at the low consistency level.

In general, the overhead of high consistency is minimal in the actions we have implemented. This does depend on the nature of the critical attributes, as their predicate evaluation and size can have an impact on the overall performance of the action. The decision between medium and high consistency thus depends more on the semantics of the action rather than performance considerations.

Persistence does not have any impact on execution time, since the core of the work is done asynchronously after the action has committed. The client therefore does not perceive any effect from persistence. It does however increase the size of the messages, which must now carry the action information necessary when replaying.

CHAPTER 7
Conclusion

Good action designs are important for the scalability and consistency of massively multiplayer games, since actions are the primary mean of interaction for players. The consistency categories presented in this thesis are integrated in the action development process. Each category provides a set of consistency guarantees which is relevant for MMOG semantics. Actions have been designed by following the models offered by these consistency categories and implemented in the Mammoth framework. The results have shown the major performance considerations each consistency category entails.

We analyzed in detail the execution model and architecture found in current MMOGs. This analysis treats MMOGs as a data management application, where concepts in database transactions can be applied to actions. We have also shown that generic solutions cannot fully accommodate for some of the most specific aspects of MMOGs. Namely, the replication architecture uses lazy update propagation and induces stale replica reads. Players can make client reads which influence their decision to take an action. Distribution of the game state introduces the issue of uniform master reads, where masters of objects cannot locally access the master state of objects residing in other servers. All of these challenges can cause inconsistencies in the game state which cannot be dealt with efficiently with a standard transactional model.

We presented five consistency categories which cater to the requirements of a variety of actions. For each category, we detailed the consistency guarantees it provides, examples of actions it is suited for, and a coordination protocol optimized for scalability and efficiency. We also show how each category can handle or is limited by distribution and client reads.

Persistence support is integrated to the consistency categories presented. The persistence architecture monitors the game state via Master/Proxy replication. Depending on the consistency category used by an action, the persistence storage protocol differs to provide the same level of consistency the action has during normal execution. This protocol takes into account game server failures, where masters of failed sites need to be rebuilt from the persistence layer. Furthermore, these new masters need to be consistent with the rest of the game state.

Finally, an action development process model is proposed to demonstrate the use of the consistency categories when designing and implementing actions. Three actions with varying properties and complexity have been implemented in the Mammoth framework. For each action, we described their use and importance, highlighting the fact that they will provide a reference point for future action designs. The requirements of each action is listed and the choice of the proper consistency category to apply is justified. Multiple implementations for each action following different consistency categories are revealed. In particular, actions with a large volume rate need to be optimized to reduce the message throughput, making them suitable for lower consistency categories. For actions involving multiple objects, performance mostly depends on the number of message rounds the action contains. There is

therefore a direct trade-off before performance and consistency. While it may be acceptable for an action using a low number of objects such as Drop and Pickup to use medium consistency, actions involving a large number of objects such as Activate Item with an Area of Effect are too expensive and must stay at the lower levels. In the end, the acceptability of the performance of each implementation depends on the requirements imposed by the game developer.

7.1 Future Work

Our work offers a concrete set of transactional models which can be used by game developers to design actions. The more novel aspects of MMOGs described in this thesis are subject to additional research. Persistence can be further explored as well to consider expanded fault scenarios. Here are some of the future work that could stem from our research:

Determining the client read set. One of the biggest challenges left unanswered in this thesis is determining the client read set. Finding the client read set prevents the player from making unintended decisions based on stale reads. One possibility is to investigate the thinking process of the AI of NPC characters. It is plausible that reads performed by an NPC are the same reads a human player would perform prior to making an action. Another possibility is to look at the interaction between the human player and its client's interface. Observing patterns in the behavior of players can allow us to infer the client reads being made. Finally, a flexible query language would allow the developers (or even the players) to specify a read set dynamically generated based on the context of the action.

Reconciling high consistency actions. High consistency actions function on the principle that it is more beneficial for the player to have its action rejected due to stale critical attribute reads rather than execute with an unintended outcome. In some situations however, both possibilities are not acceptable for the player. For instance, this can occur in a time-sensitive situation where the effort and time lost due to a rejected action denies the player other opportunities. In these situations, it may be possible to intelligently adapt the player's action, which was based on stale reads, to be suitable for the actual state of the game rather than rejecting it outright.

Action adaptivity based on performance. A first idea is to adapt based on the current load on the system. Actions can lower or raise consistency depending on the performance currently achievable. Some additional precautions must be taken during the transition between consistency levels. In particular, replicas will probably have to be resynchronized with the master when the consistency level is being raised. Furthermore, any low consistency action with a parametrized approximation update propagation can be adjusted dynamically. For instance, if player movement sends only a fraction of the position updates to the replicas, this fraction can be adjusted in-game. The same idea can be applied for persistence.

Action adaptivity based on context. The consistency requirements for an action can depend on the context of the action while it is being executed. For instance, pickup item could be normally considered as a medium consistency action, but when it is initiated to pickup a very important item, the action must be made exact instead. Developers could manually support this by developing multiple pickup actions using different consistency categories and specifying which action is used for

specific situations. However, developing an intelligent mechanism for evaluating the importance of a specific action instance would be more practical to use.

Extended fault-tolerance for persistence. The persistence model presented in the thesis does not consider failure cases in the persistence engine itself. Furthermore, faults such as late messaging must be considered as well. For instance, a local persistence server sends medium and high consistency action data late to the central server. If another local persistence server requests missing actions from the central server before it receives the data from the other server, the central persistence server will not be able to send all missing actions to the requesting server. We assume right now that the failure detection process of the game allows sufficient time for all actions involving objects on the failed server to be received by the central persistence server.

References

[1] Hyperic SIGAR. http://www.hyperic.com/products/sigar.html.

[2] The Apache Mina Java Networking Library. http://mina.apache.org.

[3] Richard Bartle. *Designing Virtual Worlds*. New Riders, 2003.

[4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM.

[5] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. *ACM SIGCOMM Computer Communication Review*, 38(4):389–400, 2008.

[6] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design and Implementation*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[7] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 6, New York, NY, USA, 2006. ACM.

[8] Angie Chandler and Joe Finney. On the effects of loose causal consistency in mobile multiplayer games. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–11, New York, NY, USA, 2005. ACM.

[9] Jin Chen, Baohua Wu, Margaret Delap, Björn Knutsson, Honghui Lu, and Cristiana Amza. Locality aware dynamic load management for massively multiplayer games. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on*

Principles and practice of parallel programming, pages 289–300, New York, NY, USA, 2005. ACM.

[10] Bill Dalton. Online gaming architecture: Dealing with the real-time data crunch in MMOs. In *GDC '07: Proceedings of the Game Developers Conference*, Austin, TX, USA, 2007.

[11] Screen Digest. Subscription MMOGs: Life beyond World of Warcraft. Technical report, 2009.

[12] Blizzard Entertainment. World of Warcraft® Subscriber Base Reaches 11.5 Million Worldwide. http://us.blizzard.com/en-us/company/press/pressreleases.html?081121.

[13] Alan Fekete. Weak consistency models for replicated data. In *Encyclopedia of Database Systems*, pages 3451–3455. Springer US, 2009.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[15] Frank Glinka, Alexander Ploss, Sergei Gorlatch, and Jens Müller-Iden. High-level development of multiserver online games. *International Journal of Computer Games Technology*, 2008:1–16, 2008.

[16] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[17] H. F. Guðjónsson. The Server Technology of EVE Online: How to Cope With 300,000 Players on One Server. In *GDC '08: Proceedings of the Game Developers Conference*, Austin, TX, USA, 2008.

[18] Nitin Gupta, Alan Demers, Johannes Gehrke, Philipp Unterbrunner, and Walker White. Scalability for Virtual Worlds. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1311–1314, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Nitin Gupta, Alan J. Demers, and Johannes E. Gehrke. SEMMO: a scalable engine for massively multiplayer online games. In *SIGMOD '08: Proceedings*

of the 2008 ACM SIGMOD international conference on Management of data, pages 1235–1238, New York, NY, USA, 2008. ACM.

[20] Florian Heger, Gregor Schiele, Richard Süselbeck, and Christian Becker. Towards an interest management scheme for peer-based virtual environments. *ECEASST: Electronic Communication of the European Association of Software Science and Technology*, 17, 2009.

[21] Tristan Henderson. Latency and user behaviour on a multiplayer game server. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 1–13, London, UK, 2001. Springer-Verlag.

[22] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 116–120, New York, NY, USA, 2004. ACM.

[23] Daniel James, Gordon Walton, Brian Robbins, Elonka Dunin, Greg Mills, Jeferson Valadares, Jon Estanislao, Steven DeBenedictis, and John Welch. IGDA Persistent Worlds Whitepaper '04. White paper, International Game Developers Association, 2004.

[24] Jörg Kienzle, Clark Verbrugge, Bettina Kemme, Alexandre Denault, and Michael Hawker. Mammoth: a massively multiplayer game research framework. In *FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 308–315, New York, NY, USA, 2009. ACM.

[25] Björn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE Infocom*, March 2004.

[26] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36:47–56, 2003.

[27] Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. Towards scalable and transparent parallelization of multiplayer games using transactional memory support. In *SIGPLAN Symposium on Principles and Practice of Parallel Programming (POPP)*, 2010.

[28] Curt Monash. The database technology of Guild Wars. `http://www.dbms2.com/2007/06/09/the-database-technology-of-guild-wars/`, June 2007.

[29] PlaySpan. PlaySpan and Magid Associates release survey on virtual goods market penetration and projected growth in North America. Technical report, 2009.

[30] Amund Tveit, Øyvind Rein, Jørgen Vinne Iversen, and Mihhail Matskin. Scalable agent-based simulation of players in massively multiplayer online games. In *Proceedings of the 8th Scandinavian Conference on Artificial Intelligence*, Bergen, Norway, November 2003. IOS Press.

[31] Marcos Vaz Salles, Tuan Cao, Benjamin Sowell, Alan Demers, Johannes Gehrke, Christoph Koch, and Walker White. An evaluation of checkpoint recovery for massively multiplayer online games. *Proceedings of the VLDB Endowment*, 2(1):1258–1269, 2009.

[32] Walker White, Alan Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportions. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 31–42, New York, NY, USA, 2007. ACM.

[33] Walker White, Christoph Koch, Nitin Gupta, Johannes Gehrke, and Alan Demers. Database research opportunities in computer games. *SIGMOD Record*, 36(3):7–13, 2007.

[34] Kaiwen Zhang, Bettina Kemme, and Alexandre Denault. Persistence in massively multiplayer online games. In *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 53–58, New York, NY, USA, 2008. ACM.