Persistence in Massively Multiplayer Online Games

Kaiwen Zhang
McGill University
Quebec, Canada
kzhang9@cs.mcgill.ca

Bettina Kemme
McGill University
Quebec, Canada
kemme@cs.mcgill.ca

Alexandre Denault
McGill University
Quebec, Canada
adenau@cs.mcgill.ca

ABSTRACT

The most important asset of a Massively Multiplayer Online Game is its world state, as it represents the combined efforts and progress of all its participants. Thus, it is extremely important that this state is not lost in case of server failures. Survival of the world state is typically achieved by making it persistent, e.g., by storing it in a relational database. The main challenge of this approach is to track the large volume of modifications applied to the world in real time. This paper compares a variety of strategies to persist changes of the game world. While critical events must be written synchronously to the persistent storage, a set of approximation strategies are discussed and compared that are suitable for events with low consistency requirements, such as player movements. An analysis to better understand the possible limitations and bottlenecks of these strategies is presented using experimental data from an MMOG research framework. Our analysis shows that a distance-based solution offers the scalability and efficiency required for large-scale games as well as offering error bounds and eliminating unnecessary updates associated with localized movement.

1. INTRODUCTION

MMOGs or MMORPGs, short for Massively Multiplayer Online (Role-Playing) Games, is a genre of online games which has seen tremendous growth since the start of the 21st century with now over 11 million paying subscribers for revenues exceeding $1 billion annually [8]. The games' main focus lies in creating vast worlds where players can interact with each other and economies with real-life financial consequences are developing on virtual property [15]. MMORPGs typically have each client control a single character in the world, referred to as the player. Characters can acquire and improve skills, pick up items in their inventory, etc. Thus, the state of the game world is constantly evolving as players interact in the game. These games are meant to be played for a long period of time, with users spending several months or years playing on a single character. The games usually run on a client/server architecture, where clients control their players, sending any actions to the game server. The server serializes the actions, adjusts the game world, and sends changes to all affected players (e.g., nearby players). The client software then renders the new game state.

Since those games are designed to run continuously for several years, the hardware supporting the game will inevitably fail or be forced to shut down. In those situations, it is crucial that the game be restored to the last state it was in. Therefore, persistence becomes an important element for MMORPGs. If the game state is dynamically stored during game play, it can be restored upon restart of a crashed or failed server. It is usually a fairly simple task to issue a complete snapshot of the world immediately before shutting down the server. The difficulty arises when we consider the case where termination is unpredictable (for instance, server crash). Ideally, there should be no distinction between the two cases: the stored state should be equivalent to the actual game state in main memory at the time of shutdown/failure. We will see that this ideal may not be realizable due to scalability and efficiency reasons. Under that context, we define the following goals for persistence in MMORPGs:

- The persistent data should, at the very least, be consistent. In other words, it must form a state which is plausible in the game. For instance, no unique item should exist in two separate locations (duplicated).
- The solution must be efficient. MMORPGs are real-time systems and clients expect fast responses to their actions. The persistence layer inevitably adds overhead to the server. The objective is to minimize the additional cost as felt by a client when executing an action.
- The solution must be scalable. MMORPGs typically support thousands of simultaneous clients [7]. We must support such capacity with persistence.

Basically all commercial MMORPGs provide some form of persistence for recovery and maintenance purposes [10], although available information about the technology used is scarce. We are not aware of any work that approaches persistence from a systematic viewpoint or analyzes persistence options. This paper is a first step in this direction.

The choice of storage is an important decision. In this paper we focus exclusively on a relational database. Since the world itself is composed of different object types, the relational model is a good fit [17]. Primary keys and primary key indices allow for efficient updates of objects. Furthermore, the database provides benefits for other purposes, such as fast querying for analytical reports and statistics. Other alternatives include storing the data to a flat text file, such

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NetGames '08 Worcester, MA, USA
Copyright 2008 ACM 978-1-60558-132-3-10/21/2008 ...$5.00.

as CSV or a memory-mapped file. However, persistence requires constant updates on very specific parts, which cannot be accommodated efficiently. A more interesting alternative would be to only append changes to a log. This allows for fast writes, but recovery is more cumbersome.

The design of a persistence solution must be optimized for updates rather than reads, as during normal operation, the only interaction with the persistence layer is usually to write the game state. We approach this challenge by using a game-aware approach. First, we propose a very general database design that suits many games, can be adjusted dynamically during game play, and allows for a fast tracking of changes at a low granularity in order to handle the large volume of updates. Secondly, we analyze typical game semantics and show how we can divide the properties of the world into different levels of priorities, each with different options for storage strategies. Finally, we look specifically at saving player positions. Position changes are by far the most common event in MMOGs. Thus, providing efficient storage mechanisms for player positions is of utmost importance. Player positions have relatively weak consistency requirements and it is likely enough to capture the approximate position of a player on stable storage. This allows for ample optimizations, and opportunities to scale. We propose and analyze a set of approximation strategies specifically designed for position updates. We have validated and analyzed the proposed approaches by integrating them into the Mammoth research prototype [1]. We show that by using appropriate approximation strategies the persistent layer is able to scale to thousands of players without being a bottleneck.

2. DATABASE DESIGN

There are several database design alternatives. As a first option, the granularity of persistence can be very coarse, e.g., at the game level. That is, the entire game state is simply stored in a BLOB attribute of a simple table. Then, each write operation has to serialize the entire game state and write it to the BLOB attribute. A finer granularity can defines the game world as a set of objects. The database has then one relation table where each row represents one object. Each row has an object identifier as indexed primary key attribute and as second attribute the object state in binary form (e.g., serialized from the corresponding main memory object). An event changing the game state then only has to serialize and write the object(s) affected by the event [14]. At the other extreme, storage can be at a very fine granularity such as the attribute level. In this case, the database would contain a relation for each object class, whereby each attribute of the class is mapped to an attribute of the corresponding table. For example, the database will likely contain a table `Player` with attributes such as player id, name, position, level, etc. A change only needs to write the attribute values that have been changed. Figure 1 shows these three possible levels of granularity.

2.1 Granularity and game knowledge

As we can see, there is a direct relationship between the database's knowledge of the game and its granularity. A game-aware design has the advantage that each update can be very specific, minimizing the size of the operation. If the storage unit is the entire game, each update needs to write the whole world whereas if storage is at the attribute level we only need to update specific attributes of the affected object(s). The object level needs to serialize entire objects, which could be expensive for objects with many attributes.

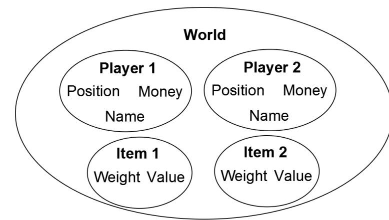

Figure 1: World, object and attribute levels.

But having a specialized database has its disadvantages, too. For each object and attribute, we need special write operations (though object-relational mappers can be used). The fact that the database is designed specifically for a particular game not only limits its support for other games, but also for other versions of the same game. MMORPGs typically undergo frequent changes to keep the games attractive for players. Every time the structure of the world is altered, the database tables might need to be adjusted. However, any altering of existing tables is expensive and can lead to problems. Not all changes and adjustments may be compatible with the previous database design, leading to complex, long, and high-risk redesigns. Both storage at the game and object level do not have such problem.

2.2 Two-layered database structure

Our solution splits the database into a game-agnostic object layer and a game-aware attribute layer. A single table `Object` stores each object of the game in serialized form (with a key attribute for fast search). Furthermore, for each attribute of an object class that can be updated dynamically – we say the attribute is *mutable* – we can create a special table with two attributes, one being the primary key of the object, the other the mutable attribute itself. Typical mutable attributes would be player position, money, etc. Having separate tables for these attributes allows for fast updates since the size of each row is small and only the attributes that were really changed are retrieved and overwritten. Note that it is only necessary to create tables for mutable attributes that are often updated, since our design also allows any attributes to change through the `Object` table. In those cases, the entire object has to be serialized and the corresponding row in the `Object` table has to be overwritten.

When during game (re-)design a new object class is created or an existing class is changed, the game designer has to indicate which mutable attributes are frequently updated. The persistence layer then automatically creates the appropriate tables. When a new object instance is created, it is inserted into the `Object` table and the initial values of its mutable attributes are inserted into the corresponding mutable tables, if present. Object changes usually only affect the frequently updated attributes, and therefore only the corresponding attribute tables need to be updated.

The purpose of each layer is therefore different: the object layer must ensure it contains complete information, no matter how outdated it is, about every object of the world, while the attribute layer must ensure it contains the most recent information about the specific attributes it has been assigned to. Loading after a shutdown or failure then first retrieves all objects from the `Object` table, and then recon-

structs the latest persistent state by reading the values from the mutable tables of the attribute layer.

This structure retains the flexibility of object level design and the efficiency of attribute level design. If the structure of an object were to change, this change is automatically reflected in the object layer and the database can still support the game. Most updates are made on frequently updated mutable attributes and thus at the attribute layer, which can support efficient updates for specific attributes. Still, even if attributes are not tagged as frequently updated, their changes can be reflected, although at a higher cost.

2.3 Database design for Mammoth

We have used the database design for Mammoth, a massively multiplayer online research framework written mostly in Java [1]. Mammoth stores the initial state of the world in a XML file called a "map". Whereas the map holds both mutable and immutable objects, our database only holds mutable objects, i.e., objects that have at least one attribute that may be updated. Our `Object` table stores the entire information about each object in XML format. Although not used currently, the XML format allows for a direct querying and updating of individual parts of the object, in case the database system provides XQuery functionality [4]. Then mutable tables are added for mutable attributes that change frequently. The loading phase first retrieves immutable objects from the map file. Then, mutable objects are rebuilt from the XML entries in the database. Finally, the objects are updated using the entries of the attribute tables.

3. STORING PERSISTENCE

Saving is the most crucial aspect of persistence as it occurs while clients are playing and therefore has a direct impact on them. It is therefore of utmost importance to ensure that saving the state of the game incurs little overhead. Earlier, we established that persistence should ideally keep an exact image of the world at any time. However, this might prove to be too expensive. Thus, we can relax the initial cost of persistence by sacrificing exactitude on certain properties.

3.1 Consistency categories

Indeed, an exact image is not required to maintain consistency. We have identified three priority categories that require different levels of exactitude.

At the lowest level, we have properties which do not need to be stored at all (no consistency). This could be immutable objects, such as non-interactive objects and the environment (walls, trees, etc.). It could also be immutable attributes of mutable objects (name of a player) or inferred attributes (the weight of the content of an inventory). It could also be an attribute which has little or no impact on gameplay or which can be modified to the correct value at any time. For instance, the orientation of a player-controlled character does not need to be logged because it is usually only a graphical effect and can be changed at any time.

At the highest priority level, we have properties which are critical to the integrity of the game and must be stored exactly (exact consistency). The state of an item in the game is generally one such property. The state could, for instance, refer to whether an item is laying on the ground or inside a certain player's inventory. Items can be picked up by players and it is important that players always retain possession of their belongings. Trading is another example. It not only requires strong consistency, but has the additional requirement that the set of update operations be atomic in order to ensure that none of the players trading end up with all the items of a trade. Changes that affect high priority properties should be stored *synchronously*, i.e., the changes are only committed at the game and the results sent to the players when the changes have been made persistent.

In between, we have properties which are logged but need not be exact (low consistency). An example is the position of a player in the world. It is likely sufficient to log a position that is within an acceptable range of the true position. This class of properties is the most interesting because it allows for strategies with a trade-off between exactitude and efficiency. In general, changes can be sent to the storage system *asynchronously*, and thus, persistence does not have a direct impact on the response time perceived by the players.

3.2 Relationship with game consistency

The level of priority is closely linked to game consistency. Although each client may perceive the world slightly differently, a consistent world ensures that certain critical elements are uniform across all clients and the server. Such elements therefore also belong to the highest level of priority for persistence. For instance, two clients may not perceive the same item in their own inventory. On the other hand, elements that can be divergent (e.g., different clients might perceive slightly different positions for a given player) belong to lower levels of priorities [12, 13]. In this case, the persistent data should constitute a *possible state*. In other words, the persistence engine can be treated as a client whose view on the world may differ from the actual world, but the differences are within the limits generally tolerated by clients.

3.3 Storage strategies

Persistence can be achieved using an event-based approach. Data is written when certain events occur, e.g., when the state of a certain object has changed. For high priority events, an event-based strategy can immediately store the change, ensuring that the persistence data is always exact. However, for lower level properties which trigger a lot of events, the event-based strategy will be flooded with notifications. It must be able to handle all these notifications and possibly restrict what exactly is sent to the database.

On the other hand, the server could decide independently of events when to store, for instance by taking snapshots. Timestamping such snapshots allows us to determine how recent the persistent data is. There is no need to keep track of game events. The disadvantage is that it may not store critical updates fast enough. For instance, the server may crash after a player picked up an item and before this event was recorded on the database, thus losing this information. However, for low priority properties which change frequently such a strategy may be an effective option.

4. STORING PLAYER POSITION

The change of a player's position is usually the most frequent event in MMORPGs. Game worlds are typically built on a continuous plane. Figure 2 shows a player walking in a game world. When a player clicks on a destination point (arrow in the figure), the player's movement is represented by a series of small successive position changes towards the target destination, denoted as thin or thick dots in the figure. As position updates are very frequent but have low consistency

Figure 2: Position Updates

requirements, it is possible and desirable to have approximate logging mechanisms for them that keep the overhead at an acceptable level. In this section we discuss a set of storage strategies for the storage of player positions. The strategies can also be used for storing the positions of other objects in the game that can displace on their own.

4.1 Naïve solution

The first solution is the simplest implementation of an event-based strategy, dubbed the “naïve solution”. This solution stores every position change by sending an update to the database with the new coordinates. Storage can be synchronous, that is, players only observe the movement after it is stored in the database. It could also be asynchronous and players are informed about the change concurrently to propagating the update to the database. In Figure 2, this strategy would store each of the dots. While this strategy maintains exact positions, it is also very expensive.

4.2 Timed snapshot

Our second solution is not event-based. The server takes a snapshot containing the position of all players and writes it to the persistence layer. The time between snapshots can be chosen dynamically in order to not overburden the server or the persistence layer. One problem is that writes occur in bursts and the database is not evenly used. Also, the strategy does not take into account that different players have different activity patterns and stores all positions with the same interval. It is more desirable to dedicate more resources to moving players and none to passive ones.

4.3 Fractional storage

Fractional storage is a generalization of the naïve solution. Instead of storing every position change, we store only a fraction of it. This can be done by having each player store a counter which triggers an update for the database at every x position events affecting this player. In contrast to the snapshot solution, fractional storage takes a player’s activity into account and updates the position of a player relative to the frequency of movements. It also updates on a continuous basis, avoiding burstiness. In contrast to the naïve solution, fractional storage can easily avert overload by dynamically adjusting the fraction parameter. In Figure 2, if the fraction parameter is set to 1/5, only thick dots would trigger writing the position to stable storage.

Fractional storage is a game-agnostic improvement over the naïve and snapshot solutions. However, taking game semantics into account, one can do better. If movement is localized, i.e., the player moves around within a small area, it is enough to keep one of these positions in stable storage as the error remains small. In this case, fractional storage performs unnecessary updates. Furthermore, if a player used an action which caused a position change at a large distance (such as teleportation), fractional storage may not store the new position instantly. If a crash occurs after the movement but before its storage, the player will restart at its old position, which may be far from the real position.

4.4 Distance-based

A fourth strategy is to take into account the distance a player traveled, as opposed to the frequency of its movements. A distance-based strategy only stores the position of a player to the database if the distance between the current position and the last stored position is greater than a set threshold. This strategy has the advantage of avoiding unnecessary updates, since local movements do not increase the distance, and have an error bound (as defined by the threshold). It might require more computation overhead than fractional storage since it has to check distances.

While distance-based storage takes game semantics into account, it is not completely game-aware. Thus, it cannot track situations where distance is not the important factor. For instance, when a player moves from one region to another (e.g., climbing up a mountain), the distance between the two regions might not warrant an update, but the properties of the regions might require one. In similar spirit, the movement might be associated with other changes that run at a higher priority level. For instance, a player might cross a bridge paying a fee. While the distance travelled might be too short to trigger an update, the money exchange will be stored synchronously. To avoid inconsistencies, such crucial movements must be made high priority properties.

5. IMPLEMENTATION AND ANALYSIS

We have implemented a persistence engine into the Mammoth prototype using the database design presented in Section 2.3. The engine provides the storage strategies listed above. Mammoth possesses a well-defined system of listeners for every event in the game. The persistence engine registers listeners on every object in the game in order to realize its event-based strategies. When a corresponding event occurs, the listener gets the control and can perform the according actions. In our synchronous solution, the listener only returns after the necessary changes have been written to the database. In the asynchronous solution, the listener only writes the changes into a main memory buffer and then returns. The changes in the buffer are then processed and written to the database asynchronously in the background. Our performance analysis focuses on storing position updates since they constitute the vast majority of events in Mammoth. In our testbed, the database is on a separate machine as the game server and the DBMS is MySQL.

5.1 Overhead distribution

In our first experiment, we analyze the delay incurred by the event-based strategies. For each event, the *server overhead* consists of analyzing the change and adding it to a main memory buffer, if necessary. The *database overhead* consists of propagating the change to the database.

At the server, the naïve strategy has an overhead of 21 μs. The fractional storage has a delay of 25μs when it adds the change to the buffer, and only 1.5μs when it decides

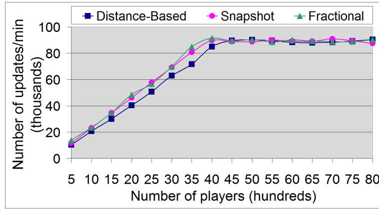

Figure 4: Approx. strategies: worst-case

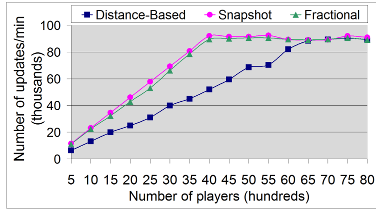

Figure 5: Approx. strategies: random

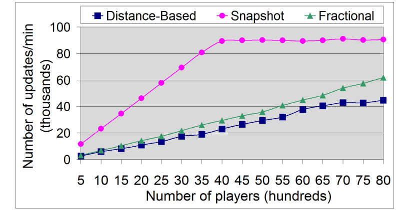

Figure 6: Approx. strategies: "pick item"

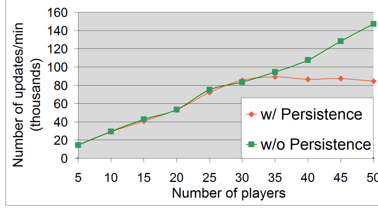

Figure 3: Naïve strategy & rate of position changes

to not make the update persistent. The corresponding values for the distanced-based strategy are 26μs and 4μs. The database overhead for each propagated change is 420μs.

Generally, the overhead at the server is small and tolerable. Thus, event-based strategies are feasible. The extra overhead for approximation strategies, namely incrementing a counter or calculating distance, is negligible. As they add less changes to the buffer, they have less overall server overhead than the naïve strategy. What is really expensive is the database access. It shows the importance of keeping the number of changes that are actually sent to the database small. Furthermore, given the long delay, the interaction with the database should remain an asynchronous background process whenever the game semantics do not require strong consistency.

5.2 Worst-case analysis

In this section, we want to determine the maximum rate of position updates the persistence engine can handle. Clients are simulated by NPCs (non-playing characters). Those NPCs are designed to wander around the world uninterruptedly, since moving without pause generates the most events in the game. In our setting, a NPC sends approximately 50 position updates per second to the server. This scenario represents a worst-case scenario in terms of position updates created but also in terms of locality, since these random players usually do not remain in a confined area.

Naïve solution – Figure 3 indicates the number of position updates the server can handle per minute when we increase the number of NPCs for a system without persistence[1] and one with persistence, both using the naïve solution in asynchronous mode. Without persistence, the rate of updates linearly scales with the number of players as expected. With persistence, the throughput caps at around 90000 position updates per minute which represents the load submitted by 30 players. This represents the maximum capacity the database can handle. Clearly, the naïve solution cannot handle the thousands of players that we would like to support. Thus, the aim of the approximate strategies must be to not write more than 90000 position updates per minute to the database while supporting a larger number of players. For instance, assuming 4000 players, on average 22.5 updates per player per minute can be written. We analytically determine the parameters of the strategies in order to support 4000 players for the worst-case scenario.

[1]The persistence structure is still in place, but the changes queued to the buffer are simply discarded.

Snapshot strategy – Whenever a snapshot is taken, the positions of all players are written to the database. Thus, n players require n writes. The maximum capacity depends on the delay between snapshots. With 4000 players, a snapshot can be taken at most every 2.67 seconds and the data on storage is therefore outdated by up to 2.67 seconds as well.

Fractional storage – The capacity of the fractional storage depends on the fraction of position changes stored. As we can store 90000 updates per minute that are equally distributed over all players (since they all move with the same speed), then each player gets written 22.5 position updates to stable storage, or one update every 2.67 seconds. Since a player makes 3000 position changes per minute and only 22.5 get written, this represents a fraction of 0.0075. Note that the performance is the same as with snapshot-based logging due to the fact that all players make continuous movements. Performance improves if players move intermittently.

Distance-based strategy – The distance-based strategy depends on the distance threshold. In our experiment, a NPC travels a distance of 1 in around 1 second. Thus, assuming that locality is minimalized, if we take a threshold of 2.7 we again end up storing an update per player every 2.67 seconds, and we can support 4000 players.

Summary – With those parameters, all three strategies are able to support 4000 players and have the same precision of 2.67 seconds. The experiment shown in figure 4 confirms the analysis: they perform equally in the worst-case scenario. The distance-based strategy performs slightly better due to the fact that the NPCs do not exactly move in straight lines. Thus, more locality is given and less updates are triggered.

5.3 Other scenarios

The difference between the strategies is more noticeable when we relax the scenario. Fig. 5 shows the results using NPCs with randomized movement that does not enforce worst-case locality. The parameter values are as before. In this situation, the distance-based solution performs better as it logs less updates per player and thus supports more players. The other two strategies remain unaffected.

The next scenario involves NPCs which look for items and move towards them to pick them up. While picking up objects, NPCs will stop for a moment to put the item in their

inventory (worst-case rate of position change is therefore not enforced). In this situation, Figure 6 shows that the snapshot strategy does not yield improved performance, while the others do perform much better. The distance-based solution only performs slightly better than the fractional storage because NPCs will actively wander around the map to search for items. Thus, there is not a lot of locality.

We thus conclude that the performance of the strategies largely depends on the scenario used. The worst-case scenario is improbable; players usually only move across great distances when they travel between major points of interest (e.g., towns), and they usually stay at these hotspots for a considerable time. Furthermore, since MMORPG worlds are meant to be interacted with, it is unlikely that players only move constantly in the game. Therefore, the last scenario is the most realistic. In this sense, the snapshot strategy is not preferable, even though it is as efficient as the others in the worst-case. Amongst the other two, the distance-based strategy is likely better than the fractional storage strategy in most cases due to its efficiency in dealing with locality.

6. RELATED WORK

We are not aware of any work that analyzed persistence in MMORPGs in detail despite the fact that persistence is supported by basically all commercial MMORPGs [10, 18]. [16] presents a platform that logs activities of simulated players in MMOGs for data mining purposes.

Mapping an object-oriented application to a relational database is a common task in many business applications [3, 11, 2]. The mapping software typically automatically generates a relational schema for a given object-oriented model. Systems such as J2EE then provide a persistence engine that writes changes to the objects automatically to the database at specific timepoints, e.g., transaction commit. The application developer only uses the object model. Such mapping software and persistence layer could also be used for persistence of game worlds. However, current solutions do not allow for approximate solutions which are crucial for MMORPGs given the large update volume. Furthermore, changing the object model, and thus, the database schema, would be very cumbersome. Our solution provides game-aware approximation and dynamic adjustments.

Consistency has been widely discussed in the context of distributed server architectures [6, 5] or peer-to-peer infrastructures[12, 9] but not in regard to the interaction between a server and its back-end persistent storage system.

7. CONCLUSION AND FUTURE WORK

This paper proposes a persistence layer for MMORPGs. We propose a two-layered database design that provides efficient writes and can be dynamically adjusted. We identify three consistency categories for game events that have different priority in regard to persistence. For low consistency events such as position changes, we propose a set of approximate storage strategies with low overhead. Our implementation in the Mammoth prototype validates our approach.

We show that by making the persistence layer game-aware, we can exploit unique game properties to considerably reduce the write load compared to game-agnostic checkpointing, and thus, can scale the system without the need to increase the capability of the database itself.

We see this paper as a first step into developing an efficient storage and recovery component for MMORPGs. In future work we want to improve our current solution by further analyzing the influence of algorithmic parameters such as the storage fraction and the distance threshold, but also engineering parameters such as the number of database connections, or the impact of different database systems. We also want to compare with other storage systems, such as logging, where changes are written sequentially to disk. Finally, we want to look into recovery schemes for systems where the load is distributed across an entire server cluster.

8. REFERENCES

[1] Mammoth - a multiplayer game research framework. `http://mammoth.cs.mcgill.ca/`.

[2] S. W. Ambler. Mapping objects to relational databases, AmbySoft. Inc. White Paper, 1997.

[3] D. K. Barry. Object-relational mapping articles and products. `www.service-architecture.com/object-relational-mapping/`, 2008.

[4] D. Chamberlin. XQuery: a query language for XML. In *ACM SIGMOD*, 2003.

[5] A. Chandler and J. Finney. On the effects of loose causal consistency in mobile multiplayer games. In *NetGames*, 2005.

[6] J. Chen, B. Wu, M. DeLap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *PPOPP*, 2005.

[7] W.-C. Feng, D. Brandt, and D. Saha. A long-term study of a popular MMORPG. In *NetGames*, 2007.

[8] K. Graft. NPD: Online Subs Exceed $1 bln Annually. `www.next-gen.biz/index.php?option=com_content&task=view&id=10377`, May 2008.

[9] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *NetGames'04*.

[10] D. James, G. Walton, B. Robbins, E. Dunin, G. Mills, J. Valadares, J. Estanislao, S. DeBenedictis, and J. Welch. IGDA persistent worlds whitepaper'04.

[11] A. M. Keller, R. Jensen, and S. Agrawal. Persistence software: Bridging object-oriented programming and relational databases. In *ACM SIGMOD*, 1993.

[12] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE Infocom*, March 2004.

[13] F. W. Li, L. W. Li, and R. W. Lau. Supporting continuous consistency in multiplayer online games. In *Int. Multimedia Conf.*, 2004.

[14] A. Riddoch and J. Turner. Technologies for building open-source massively multiplayer games. Worldforge.org, 2005.

[15] N. Robischon. Station exchange: Year one. Whitepaper, 2007.

[16] A. Tveit, Ø. Rein, J. V. Iversen, and M. Matskin. Scalable agent-based simulation of players in massively multiplayer online games. In *Scand. Conf. on AI*, 2003.

[17] G. Wadley and J. Sobell. Using a simple MMORPG to teach multi-user, client-server database development. In *MS Academic Days Conf. on Game Develop.*, 2007.

[18] W. M. White, C. Koch, N. G. 0003, J. Gehrke, and A. J. Demers. Database research opportunities in computer games. *SIGMOD Record*, 36(3):7–13, 2007.