2011 30th IEEE International Symposium on Reliable Distributed Systems

Transaction Models for Massively Multiplayer Online Games

Kaiwen Zhang
Department of Computer Science
University of Toronto, Toronto, Canada
Email: kzhang@cs.toronto.edu

Bettina Kemme
School of Computer Science
McGill University, Montréal, Canada
Email: kemme@cs.mcgill.ca

***Abstract*—Massively Multiplayer Online Games are considered large distributed systems where the game state is partially replicated across the server and thousands of clients. Given the scale, game engines typically offer only relaxed consistency without well-defined guarantees. In this paper, we leverage the concept of transactions to define consistency models that are suitable for gaming environments. We define game specific levels of consistency that differ in the degree of isolation and atomicity they provide, and demonstrate the costs associated with their execution. Each action type within a game can then be assigned the appropriate consistency level, choosing the right trade-off between consistency and performance. The issue of durability and fault-tolerance of game actions is also discussed.**

I. Introduction

Massively Multiplayer Online Games (MMOGs) is a popular genre of online games. These games revolve around large virtual worlds where players control their avatar to interact with others and the environment. The state of the game is constantly evolving and shared by the players. Personal data has value: most of the time spent in the game involves collecting items. Another important aspect of MMOGs is their size. The millions of players which constitute the user base of commercial games are divided into shards, each with a limited capacity measured in thousands of clients. The focus is put on expanding the scale of games which must be supported by the underlying system.

The typical structure of MMOGs have clients each controlling a single character, or avatar, in the world through a graphical user interface. 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 characters. Players also take *actions*, which constitute the basic mean of interaction in MMOGs. This includes moving in the 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 should be observable by others.

Game engines are commonly implemented as a middleware. It receives the requests from all clients, manages the game world, and disseminates updates to the clients. Game state is also made persistent in a database. In principle, a MMOG can be treated as a database application [1]. The world objects are the data and actions are logical sequences of read and write operations on the attributes of one or more objects. Ideally, actions are ACID transactions. Action need to atomicity to be executed in their entirety or not at all. Durability is essential as these game worlds run for long periods of time and need to survive various failures. Players can request actions on the same objects concurrently, requiring concurrency control to provide isolation.

However, several reasons exist as to why MMOGs can not be implemented simply as a set of transactions using a traditional database system. First, most actions are update transactions, and the high rate of transactions often can not be handled by a single database server [2]. Secondly, the system is, by nature, highly replicated. Each player sees part of the game world and the characters residing in that area. This is usually achieved by giving each client copies of the relevant game objects which are updated upon changes [3].

Since these games can have thousands of players playing simultaneously, the update rate and the degree of replication are considerable. Propagating every change to every copy in an eager fashion (within transaction boundaries) in order to provide a consistent view to every player is simply not possible [4]. 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 [5]. Furthermore, they commonly allow many inconsistencies to occur which the players are forced to accept [6]. Durability is mostly handled in a very optimistic fashion: any gameplay shortly before an outage can be lost [1].

This paper uses an application-aware approach to achieve the proper level of consistency despite the large update rates of MMOGs. Consistency requirements can differ widely between actions due to their variation in complexity. For instance, if an action is frequently executed, it might not require a high level of consistency as it can be easily repeatable and can tolerate transient faults. However, if an action involves more than one object, atomicity becomes a greater issue. Furthermore, effects of certain actions depend on the value of read attributes. Isolation is more difficult to maintain as the number of reads increases. Finally, important events with a significant impact on the game, actions that take a long time to be accomplished, or ones using real currency

1060-9857/11 $26.00 © 2011 IEEE
DOI 10.1109/SRDS.2011.13

necessitate a high level of consistency. For instance, moving a character is considered of low complexity because it affects only a single character and is easily repeatable. Trading items between two players is highly complex because both parties must agree to exchange the same set of items. This involves a series of interaction steps between the players.

Clearly, given this variety in action types, there is no one-fits-all solution to consistency. Thus, commercial games employ ad hoc solutions with undefined semantics. Indeed, there are ample opportunities to optimize for performance whenever possible while providing strong consistency whenever needed. The final goal is to design a system which provides the desired player capacity, the client response times needed for enjoyable gameplay, and 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 the appropriate consistency category for each action. This allows developers to choose the right trade-off between performance and consistency, just as they can choose isolation levels in database systems. One key observation is that low complexity actions usually have a higher volume than higher complexity ones. For instance, a player frequently moves its character but seldom performs trading. The game engine can achieve scalability and performance by executing low complexity actions using efficient, weakly consistent protocols.

Our consistency categories have similarities with the various degrees of isolation, such as read committed and serializability, that are offered by traditional database systems [7]. In both cases, the developer is presented with a trade-off between performance and consistency. Our approach goes further as we consider staleness of replicated objects.

In summary, the paper makes the following contributions:

- We provide a thorough analysis of the execution model and consistency requirements of current MMOGs.
- We provide a set of consistency categories that are useful in the context of MMOG semantics; our consistency models consider isolation, atomicity and durability.
- We show how the properties of each category are leveraged to design efficient coordination protocols for both single server and distributed models.
- We show how our approach can be integrated into an existing game engine and show performance results.

II. Game Execution Model

In this section, we present the typical game architecture and execution model of MMOGs. We first focus on a single server system. Section VI discusses distribution.

A. Replication Architecture

Each player hosts the client software of the game. It renders the game world and receives input from the player. Clients 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 change. This can be supported by a primary copy replication mechanism [3], [4]. That is, the game world consists of a set of objects, each of them being a collection of attributes. The server holds a *master copy* (or master) of each object (dark shapes in Figure 1). The attribute values of the master contain the latest and correct state of the corresponding object. The game state C is defined as the union of the states of the masters of all objects.

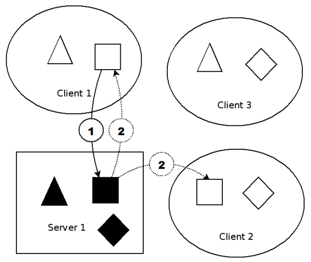

Figure 1. Sample action execution

Each client can then hold a read-only *replica* for an object (white shapes in the Figure). At any given time, the state of a replica might be lagging behind that of the corresponding master, missing some of the changes that have already been installed at the latter. A client holds replica only for objects that it is interested in. The composition of a client's *replication space* is determined dynamically by a process called *interest management*. Various methods exist for determining a player's interest [8]. For example, one can define a circle around a player: all objects within this range are considered interesting and must be included in the replication space. The perceived state of all replicas held by a client is called the *client view*.

B. Action Execution and Update Propagation

An action is a sequence of read and write operations performed on attributes of objects found in the world. We define the write set $WS(A)$ as the set of all objects updated by action A. In the following, we ignore the fact that actions can create new objects. Furthermore, we consider only actions that contain at least one write operation, as basically all actions belong to this category.

As part of an action A initiated by a client, the client reads the state of some of its replicas before submitting the action to the server. The server then performs a sequence of read and write operations on some of the masters. As a result of the action, the master copies of the objects in $WS(A)$ now possess a new state which must be disseminated to the replicas. Figure 1 illustrates a sample execution. The client submits the action to the server (arrow 1), which executes it and disseminates the changes (arrows 2).

C. Read Operations

Understanding what objects are read and how they are read is crucial to recognizing the required consistency levels.
Action reads. Consider the following actions. Moving an avatar first reads the current position of the character. Drinking a bottle checks its content attribute: the action succeeds only if there is water remaining. These read operations are explicitly written into the action code, hence called *action reads*. Some of them are first performed on the replicas residing at the client side. In particular, *validation reads* are used to verify whether an action is possible. For example, when a player tries to pick up an item, the local client software first checks the local replica state of the item. If it is on the ground, it forwards the action to the master. Otherwise, the action is immediately aborted. At the master side, the validation reads have to be repeated, as the state at the server might be different. Thus, the reads at the client can be considered preliminary, useful to avoid sending unnecessary actions to the server. However, an action deemed possible at the client might fail due to differences with the masters state. For instance, although a player saw an item on the ground, the pickup request might not succeed because the state at the server indicates that the item is already in the inventory of another avatar. In general, it is important to note that action reads performed on replicas usually always have the potential of being stale.
Client reads. Additionally to the action reads that are explicitly encoded in the action, a player continuously observes the state of all its replicas, whether they are stale or not. The player uses this information to decide what action to perform. Essentially, the player is reading certain semantically relevant attributes before initiating an action on its client software. 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 not part of the action code. Thus, 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 to pick up an item. The move action itself only reads the position attribute of the player. Yet, the decision to move is influenced by the position of the item, as established by the player. Thus, the action read set contains the current position of the player's avatar, and the client read set contains the location of the item.

III. TRANSACTIONS

In principle, an action can be split into three parts. A client first performs action reads and client reads on its replicas. The server then executes reads and writes on the master copies. Finally, clients apply the changes from the server.

We can consider the full execution in two ways. First, we can view the entire execution as one global transaction. Accordingly, there exists a set of read operations on replicas held by clients, a sequence of read and write operations on the masters, and a set of write operations on replicas. Secondly, we can treat the execution at clients and server separately. All operations on masters are considered as a *server transaction*, while all operations on replicas for one client are a separate *client transaction*.

If possible, the global transaction should be executed under full transactional properties: atomicity, isolation and durability. Unfortunately, this would be costly, requiring distributed locking, eager propagation of updates, and a distributed commit protocol. No game architecture uses such approach. In our solution, we relax consistency by considering correctness criteria for the server transaction and those for the client transactions separately.

A. Server Transactions

Maintaining ACID properties for server transactions is crucial as they keep the game world as a whole consistent.

To begin, we assume that actions are designed in a consistent manner: given a consistent game state C, the full execution of an action at the server under isolation transitions C to a new consistent state C'.

Further, *master atomicity* is maintained if all or none of the action succeed at the server. This can be implemented easily via a standard rollback. Atomicity is more challenging in a distributed setting (see Section VI).

Also, isolation is easily provided by using a concurrency control mechanism such as two-phase locking [9]. Assume one action breaks a bottle and a concurrent action picks it up. If the break is serialized before the pickup, then a broken bottle is picked up. If the pickup gets the lock and executes before the break, then the break will not succeed as a bottle that is in the inventory of another avatar can not be broken. Formally, we say that *master isolation* is provided if the execution at the server is serializable. That is, the concurrent execution of a set of actions at the server is equivalent to serial execution of these actions.

Our approach for durability is dependent on the consistency level. We will discuss it in Section V.

B. Client Transactions

The atomicity and isolation properties of client transactions reflect the player's gameplay experience.

We define *replica atomicity* to be given if every update performed at a master is also applied at the related replicas. For performance reasons, we assume that this happens after the action commits at the server and not within the boundaries of the server transaction. That is, our definition of atomicity is more relaxed than the traditional definition.

Replica isolation refers to the notion that if a client initiates an action based on its local action reads and client reads, then the updates determined by the corresponding server transaction and forwarded to the client conform to said reads. In other words, if the client transaction had acted in isolation (performing only local reads and changing the

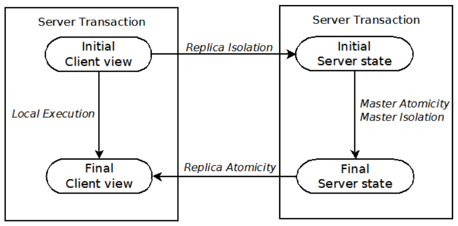

Figure 2. Master/Replica consistency relationship

local replicas) the outcome would be the same as the updates triggered by the server transaction.

For example, assume the purchase of an item. The client reads the price as P from the item replica. If replica isolation is maintained, the price read at the master should also be value P. Suppose not, and the master value is actually $P+1$. Assuming the player has enough funds, the update triggered from this action notifies the client that the item was bought at $P+1$. Since the outcome is different from an execution using the locally read price value, replica isolation is violated.

Looking at this formally, replica isolation does not provide serializability at the global level. The state of the master might be different to the state read at a replica, leading to something similar to non-repeatable reads. However, if the difference does not change the outcome of write operations, then it does not violate replica isolation. This means that our definition of replica isolation considers the game semantics.

Figure 2 illustrates the relationship between client and server transactions. When the server receives an action request, master atomicity and isolation enables the game state to move from one consistent state to the next. The view perceived at the client depends on whether replica isolation and atomicity are supported or not.

IV. Consistency categories

Games consist of various types of actions with different consistency requirements. Current single-server game systems often provide master isolation and atomicity through standard concurrency control and rollback mechanism or by simply executing transactions serially. In contrast, replica isolation and atomicity are often ignored. In fact, it is unclear what semantics are typically provided.

Given the variety of actions, a multi-level approach to game consistency is sensible as it provides the best trade-off between consistency and performance. Lower categories are more efficient and scalable while higher levels offer more consistency guarantees. Thus, application developers can choose for each of their actions the proper consistency level. In total, we identify five categories, and show action handling protocols that can satisfy the requirements of each category while maximizing its performance and scalability.

All single server categories assume master atomicity and isolation and the use of strict 2-phase-locking (2PL) at the server. The three lowest category levels do not offer replica isolation and differ on replica atomicity. The three highest levels offer replica atomicity but differ on replica isolation.

We consider read and write operations to be on object attributes and not entire objects to allow for fine-grained concurrency. Attributes are tagged with a category. An attribute x is said to be of category c if c is the *lowest* category for which there exists an action which writes to x.

A. *No Consistency*

Properties: The lowest consistency category provides no guarantee and follows a “best-effort” *maybe* semantic model. Replica atomicity and isolation are not guaranteed. The server is not required to inform clients about any state changes incurred by no consistency actions.

This consistency category can be used for actions whose importance on game play is very low. These actions are usually easily repeatable to compensate failures. Examples are graphical effects such as rotating a character or showing an emote (such as dancing or smiling).

Protocol: The server executes these actions only when it has enough resources. Assuming strict 2PL, the server locks all objects before accessing them and releases locks at the end of the action. The state changes are propagated to the replicas using an unreliable message channel (e.g., UDP), as replica atomicity is not required.

B. *Low Consistency*

Properties: The idea is to provide a bound on the staleness of the data. Like no consistency, not every update needs to be propagated to every replica, but propagation must take place when the bound is reached. Replica atomicity is therefore provided partially. This is useful for large volume actions requiring some guarantee in regard to their visibility.

The most common action type that fits into this category is the progressive movement of an avatar. While clients might not perceive the exact position of a player 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 possibility is to support eventual convergence to the real position value.

Protocol: If the error bound is given as a percentage of received updates, only a fraction of the updates needs to be sent reliably, while the rest can be sent unreliably or dropped in case of overload. If the bound is given as a threshold difference, then an update is propagated reliably once the difference between the value at the master and the last reliably propagated value exceeds the threshold.

A special form of bounded position approximation is provided using dead reckoning mechanisms [10]. Instead of sending every position change, the player’s final destination and average speed is sent to the replicas. The client software then locally simulates each position change. While this does not guarantee an exact view on the player's position, the replicas will be only slightly off the true position.

Again, master isolation is achieved through strict 2PL.

We expect actions of this type to build a large portion of the volume of all activity in the game. In fact, the most common action type is player movement. Thus, optimizing the performance of this action type is significant.

C. Medium Consistency

Properties: The medium consistency category reflects the way current game middleware handles most of the actions. Replica atomicity is provided and requires exact updates to be sent. According to our definition, it is enough to send these updates after the action committed at the server. The bound on replica staleness is therefore the maximum latency for delivering an update from the server to a client.

Actions at this level still do not require replica isolation. Since clients can perform stale reads, the outcome at the server may be different to what the client expects.

This consistency level is appropriate for actions which are important enough to require exact updates. The inconsistencies which arise from the lack of replica isolation must be tolerable or predictable. For example, two nearby players send a request to pick up an item. At the server, one action is serialized before the other. The second action will be rejected because the item will already be picked up. Upon receiving this result, the client of the second action can justify the outcome due to the close proximity of the other player, as it can deduce that other player must have picked up the object.

Protocol: When a player submits a medium consistency action, the client software performs local validation reads. If successful, the action is forwarded to the server. The master executes the action under master isolation and atomicity. After commit, each master reliably sends the exact changes to all associated replicas.

D. High Consistency

Properties: Unexpected outcomes are sometimes undesirable. This is the case if critical attributes are affected by an action. For example, price is a critical attribute when buying an item. Normally, one does not want to buy an item for more than the locally visible price, but it is acceptable if the item is cheaper. With medium consistency, however, the item will be bought independently of the exact price at the server as long as the client has enough funds.

Thus, the high consistency category introduces the concept of *critical attributes* and guarantees a limited form of replica isolation on such attributes. Our isolation property is relaxed along two dimensions.

First, we do not require strict repeatable but only *comparable reads*: the difference between the value at the master and the value at the replica must be *acceptable* for the action. This depends on the game semantics. For instance, if the price attribute at the client is larger than at the server then the difference is acceptable, otherwise it is not acceptable, and the action should fail as a repeatable read is violated.

Second, we only offer replica isolation *safety*. If the difference is unacceptable, the action must be rejected. High consistency essentially reduces any inadmissible executions into a rejection, avoiding inconsistencies.

Protocol: The protocol is similar to the one for medium consistency. The main difference is that an action submitted to the server contains the client replica values of all critical attributes. When the server reads the master copy, the value is compared to that of the client. If the difference is acceptable, the action continues; otherwise it aborts. In order to perform the comparison, the action code needs to provide a predicate that, given the replica and master values as input, 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. Furthermore, action aborts can occur due to the violation of attribute comparability.

E. Exact Consistency

Properties: The high consistency protocol is optimistic as possibly stale values are read at the client and validated at the server. This can lead to many aborts. Exact consistency avoids stale reads, and thus, provides full replica isolation. Guaranteeing that data is fresh requires the integration of client confirmation into the action itself. This category is designed specifically for those complex actions with multiple client input steps. An example is trading where items and money are exchanged between two avatars. Given the importance of this type of action for gameplay, 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.

We also enforce a stronger condition than replica atomicity. 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, atomic execution on all replicas of involved players is provided.

Protocol: We use a pessimistic approach consisting of two steps: a request and a confirmation. Upon receipt of the request from a client, the server sets long shared locks on all involved objects in order to avoid concurrent updates. It then propagates the current state of these objects to the replicas of all clients involved in the action. Once the client(s) have received the latest state, they have to confirm the action. The server then performs the write operations at the masters, and the changes are propagated to all involved client replicas. Thus, clients receive immediate confirmation on the success of the action. Others are updated asynchronously.

V. Durability

Games run for months and years. Therefore, game state has to be made persistent, i.e., written to stable storage, in order to recover from system failures. Persisting all changes before commit would lead to an immense overhead, similar to sending updates synchronously to replicas. The consistency categories described above reflect the importance of write operations and the attributes that are written. Thus, we define durability categories along the same levels.

No-durability attributes: We do not need to persist changes of no consistency actions, as they are not crucial to the game.

Low durability attributes: As clients can see bounded staleness for low consistency attributes, we can use the same staleness bounds for making changes persistent. Whenever the server decides to send an update reliably to the replicas, this update is made persistent.

Medium and high durability attributes: Medium and high consistency attributes can be stale. Therefore, we can persist all changes asynchronously after commit.

Exact durability attributes: Whenever an exact consistency action is executed, its changes are persisted before the transaction commits in order to provide the transactional durability property for these crucial actions.

When the server crashes and restarts, it reconstructs the game world with the persisted information. An issue arises if a player has received an update on one of its replicas before this update was persisted. For example, it could have observed a new value for the energy level, but this change is not persisted, and thus, is lost when the server restarts. This inconsistency is avoided by first persisting the change, and then propagating it to the replicas.

The persistence layer can be part of the game server or decoupled. In the latter case, the persistence server maintains replicas of all objects. The game server sends updates to these replicas as it does for other replicas. In the case of exact consistency, changes are sent synchronously to these persistence replicas. When the persistence server receives the updates, it writes them to stable storage.

VI. Distribution

In order to scale, current games deploy server farms. A simple distribution model lets each server host an instance of the game or a partitioned region of the game world. Each client resides in one instance or region and is connected to the corresponding server. With such simple model, each server can implement the above categories without any server coordination. Recent approaches go beyond this simple model and offer a continuous space that is maintained by cooperating servers [11]. One way to distribute load is by using cells which form non-overlapping regions of the whole game world. Cells are assigned to servers who hold the master copies of objects residing in that cell. If an object moves from one cell to another, its master copy is migrated to the server of the new 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. To allow for interest management, each server holds replicas of objects that are potentially interesting to the characters within its cell. Figure 3 shows the responsibility range of a server, maintaining the four red cells. The red triangles show objects for which the server holds master copies, black triangles shows objects for which the server holds replicas and empty triangles are ignored by the server.

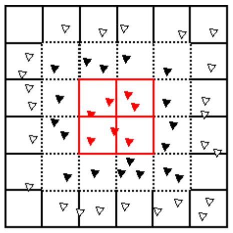
Figure 3. Cell-based architecture

Execution model. *Distributed actions* can write to 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 (called *sub-action*) that refers to these updates. We assume that the client software coordinates distributed actions sending the individual sub-actions to the affected servers. The action code must be written accordingly and specify for each read r the set of writes WS_r that depend on it. If a server requires the value of a read on some object X for which it does not have the master, it can either request that value from the appropriate server or perform the read on its replica of X.

Master isolation. In a single server system, an action executed at the server is always guaranteed to read correct values as the server has all the masters. With distributed actions, a server might perform a read operation on a replica which can be stale as update propagation is lazy. 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. Server $S1$ has the master of player $P1$. Server $S2$ has the masters of player $P2$ and the bottle. $P1$ has to send sub-actions to both servers. When $S1$ executes $P1$'s action it reads its replica of the water bottle, which is full, and gives the avatar the appropriate energy points. At $S2$, the concurrent action is executed, providing $P2$ with the same energy points and changing the state of the water bottle to empty. When now $P1$'s sub-action arrives at $S2$ it succeeds (emptying an empty bottle). However, the final execution is inconsistent because two players received the full energy points. In contrast, a single server would serialize the actions; the last action will see an empty bottle and not provide the player any points.

This form of inconsistency violates master isolation, since the server transaction depends on stale data. It is much more severe than the optimistic reads at the clients since the game state becomes inconsistent. In contrast, inconsistencies related to replica isolation only affect the client view and not the authoritative state of the game.

When master isolation is requested, servers must always retrieve values at the masters: additional synchronization is thus required to communicate read values across servers.

Master atomicity. A second challenge is to guarantee the atomicity of distributed actions. All or none of the servers need to commit the action.

Category	Description	Examples
Exact	Master and replica atomicity, master and replica isolation	Trading between players
High	Master and replica atomicity, master and limited replica isolation	Buying an item at a critical price
Medium	Master and replica atomicity, master isolation	Picking up an item
Low	Limited replica atomicity	Player moving
None	No guarantees	Player turning

Table I
CONSISTENCY CATEGORIES IN A DISTRIBUTED ENVIRONMENT

A. Consistency Levels

As master isolation and atomicity are more costly to achieve in a distributed environment, we determine for each consistency level their requirements. Table I summarizes our revised consistency categories for distribution.

We require master isolation and atomicity for medium consistency and higher, as we want to maintain a consistent game state. For actions which involve a large number of reads, however, it is very beneficial to let the servers read from replicas (either their own or the coordinating client's). This reduces the number of master reads and indirectly the number of servers involved. Therefore, we do not require no and low consistency levels to guarantee master isolation. Then, actions with massive reads have the option to use these low levels and avoid expensive execution. Similarly, we also do not require master atomicity for the lowest two levels. In fact, if an action affects many objects, the clients often do not recall the exact values at their replicas. Thus, inconsistencies do not become visible.

Exact consistency. In the case of exact consistency actions, the client will request from all servers the latest state of all objects read. These requests will lock the master copies at the servers. When the client sends the confirmation, it sends to each server that is involved in an update the changes on the replicas the server has to read. Thus, it is guaranteed that the server will read the latest values. Compared to a single server system, message exchange is multiplied by the number of involved servers.

High and medium consistency. Our solution leaves the action coordination to the client. In the worst case, we need an additional round of synchronization to provide every server with the latest state of all masters read similar to what is done with exact consistency. However, a more efficient linear protocol is possible for a common class of actions where there is no circular dependency. Assume the case of picking up a bottle which is only possible if the bottle is on the ground. The change on the bottle (being picked up) only depends on its own state (it has to be on the ground). In contrast the update on the character is conditional on the state of another object (the bottle being on the ground).

Thus, we sort the sub-actions of an action based on their dependency. First, the client submits the sub-action on objects that do not depend on the state of other objects. The server acquires the locks and executes the operation. If it succeeds, it sends the state of the objects needed for other writes to the client. The latter then forwards this state 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 succeed or one fails. 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. All servers release the locks only after commit/abort is completed.

Low and no consistency. For these two levels, the client simply sends the sub-actions to the affected servers. Each server executes its own sub-action independently and commits locally whenever the sub-action is completed. Master isolation is not provided. A server can either read its own, possibly stale replicas, or the client can send the state of its read replicas to the server. Master atomicity is not provided as each server commits independently allowing for partial execution if some of the sub-actions abort. The protocol is efficient as a single message exchange is involved between the client and each server. Locks at the servers are only held for the duration of the local sub-action.

B. Failure Handling and Durability

Persistence can be offered by a single persistence server or by a set of persistence servers, each responsible for the persistence of a subset of objects. Cell servers send object changes to the persistence servers according to their durability categories presented in Section V.

The persistence layer can be used not only for recovery of a failed server but also for failover if immediate restart is not possible. As the objects managed by the failed server have replicas at the persistence servers, they can reconstruct the latest state of each of these objects and transfer new masters to one of the available servers.

The biggest challenge is maintaining master atomicity as servers might fail before sending the changes within their sub-actions to the persistence layer. Providing a correct solution in all failure cases would require a 2-phase commit protocol where all involved parties write appropriate log records during the protocol. Clearly, this is too expensive for anything other than exact consistency as this is significantly more costly than lazy propagation. Such overhead can be avoided if every sub-action contains enough information to replay the sub-action at any server. Therefore as soon as one of the sub-actions is made persistent, the entire action can be replayed in case of server failures.

In case of client failures, the servers involved in a pending action of a failed client may block in higher consistency levels. This can be resolved by using a termination protocol.

VII. Performance Evaluation

The performance improvements perceived by using lower categories depend on many factors, such as expected volume, pattern and computational cost of the actions. In this section, we provide an analysis of two types of action: "Player Movement" and "Pickup Item". We compare different implementations of the actions based on the consistency categories. We only study low, medium and high consistency as the performance of exact consistency is difficult to measure as it involves user interaction. Our evaluation will not include the persistence aspect. A study case for persistence of player movement is offered in [12].

A. Implementation

1) Player Movement: Movement is implemented using the position and destination attributes of a character. When a player selects a new destination, it updates the destination attribute. At regular intervals, the game will read the current position attribute which and update it by a small value to be closer to the destination. This repeats until both attributes have the same value. The player movement only updates the player object. We have implemented low, medium and high consistency versions of this action.

Low consistency. After a client submits a movement request, the server propagates the current position and the destination read at the master to all replicas. Server and clients then perform dead reckoning locally to update the current position of their copies. No further messages are sent. The actual position of the player at the different replicas can vary since dead reckoning produces slightly different results due to clock differences between the nodes. We note that the object is locked each time it is being updated and released immediately. This lock does not span any message rounds.

Medium and high consistency. The client first submits a movement request to the server which initiates the action. Upon position changes at each time interval, the server sends the new position to all replicas which simply apply it. The difference between high and medium consistency is that position is treated as a critical attribute. At the time the player sets the destination, the replica's position is compared to the master's. Since position is a floating point, it is difficult to obtain exact matches due to numerical errors. We set a threshold of 0.1, which does not result in any visual difference for the client.

2) Pickup Item: This actions updates two objects, and thus, allows us to evaluate a distributed environment. A player can pick up an item and place it in its inventory. The action reads the weight and location attribute of the item, and the current carried weight and the capacity of the player. The action only succeeds if the item is on the ground and the current carried weight plus the weight of the item are not larger than the capacity. The action changes the location to be in the inventory of the player, and increases the current carried weight by the weight of the item.

We implement low and medium consistency versions of this action. In our experiments, the player and the item always reside on different servers.

Low consistency. No master isolation and atomicity is required. The client sends both sub-actions concurrently and the servers read from their local master/replica. The servers commit immediately without requiring further synchronization. They send their updates to the replicas after commit.

Several inconsistencies can occur. The server that has the master of the player can have a stale replica of the item indicating that it is on the ground while it is already picked up. Thus, the server will update the players current carried weight. However, the master of the item will reject its sub-action as the item is no more on the ground. Atomicity is violated. The successful sub-action needs to be rolled back. This compensation is outside of the action's execution and allows for the inconsistent carried weight to be disseminated. Other inconsistencies can occur if the player replica has a stale current carried weight or the item's replica has a stale weight. In this case, picking up the item might put the player overweight. Master isolation is violated. The severity of this inconsistency depends on the semantics of the game.

Medium consistency. Master isolation and atomicity is required and we implement the linear approach. The client first sends the sub-action to the server with the item master, which acquires a lock, verifies that the object is on the ground, changes its state to be picked up, and returns its weight. Then, the client sends the sub-action to the server with the player master, including the item's weight. The server acquires the lock, verifies that the object's weight is less than or equal the remaining capacity, and if so, puts the object into the inventory and adjusts the total weight. When it confirms to the client, the latter sends a commit to both servers. The servers commit the sub-actions and release the locks. If either master determines that the action is not possible, it aborts and sends the corresponding information to the client who ensures the other sub-action also aborts. Our locking scheme is special in that if an action finds the item already locked, it does not wait but immediately aborts.

B. Setup

Our benchmark application is the Mammoth framework [3]. It is built on a distributed replicated architecture

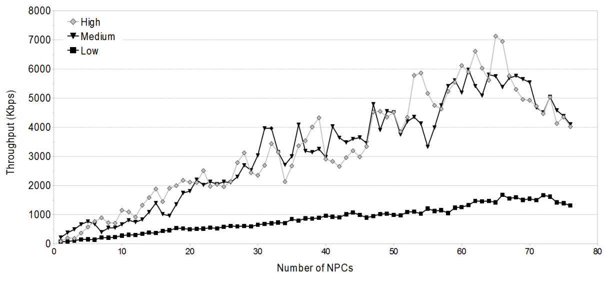

Figure 4. Movement: Throughput vs. Number of NPCs

as described in this paper. The network engine uses a publish/subscribe system built on Apache MINA. To generate actions, Non-Playing Characters (NPC) clients are instructed to continually perform the benchmarked action. The servers were ran on Quad-Xeon processors machines with over 8Gb of main memory. NPC clients were ran on separate virtual machines located on 80 different machines ranging from Pentium 4 to Core Duo processors. All machines were connected in a local network.

C. Analysis

1) Player Movement: Our analysis focuses on throughput and not response time. In fact, the computational overhead incurred by medium and high consistency compared to low consistency is negligible because movement is a simple action and involves only one object. However, the dead reckoning algorithm to determine the next position is costly [13]. We do not consider path-finding costs in our analysis. Instead, our low consistency optimization aims at reducing the network traffic sent to the replicas.

There is a clear difference in the throughput between low and the two other implementations (see Figure 4). This is because players update their position approximately 10-15 times per second. In medium and high implementations, each of these position changes results in an update to all replicas. At around 60 players, performance degradation occurs for medium and high consistency.

The difference between medium and high is negligible. The average message size for high consistency is 875.48 Bytes, while it is 873.08 Bytes for medium. The overhead of appending the local read is small, so the choice between those two levels is more of a concern towards isolation control rather than performance.

2) Pickup Item: Four servers are assigned to equal quadrants. As pickup is a distributed action, response time is our main concern. Figure 5 shows the average execution time for low and medium pickup actions. The additional message round of the medium action accounts for the increased delay (119.34 ms compared to 23.47 ms).

Further experiments have shown that the execution time stays stable for an increasing number of players for low and medium consistency. Even though the number of concurrent

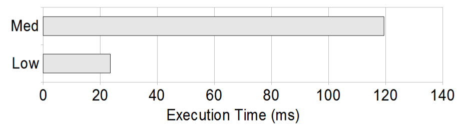

Figure 5. Pickup Item: Average Execution Time

requests increases, any medium consistency request that fails to acquire a lock on an object is immediately rejected instead of waiting. Therefore, actions do not block when concurrent pickups on the same item occur: a high throughput with good response times can be achieved.

Although the base cost is higher, the medium consistency implementation is a good fit for pickup because it scales as well as the low consistency one. We think the performance advantage of the low consistency implementation is not significant enough to compensate for the numerous inconsistencies that might occur.

D. Summary

We have shown how low consistency can be used to optimize the performance of actions which generate a large number of state changes as low consistency can reduce the number of updates sent. Medium consistency can be appropriate for actions with no obvious optimization opportunities. This is the case with pickup item, which does not generate a large number of updates nor read a large number of objects. The difference between medium and high consistency is functional rather than performance-based. Including more critical attributes with more stringent comparison predicates will only increase the rate of rejection, not necessarily affect the performance of the action.

VIII. Related Work

In [1], [14], the authors propose consistency protocols that move the execution of actions to the clients. The task of the server is to serialize actions and to forward them to all relevant clients. Instead of using locality-based interest management, the server takes the read and write sets of actions and their transitive closure into account in order to determine which actions conflict, influence each other, and are relevant for which players. The protocols are optimistic and allow the client to read stale data and to execute an action first on this stale data. When the client later receives the correct order of its action, it might have to reconcile. Some of the presented protocols send, similar to our exact protocol, to the client the current state of all objects to be read by an action and possibly conflicting concurrent actions. Only after having received this information can the client send a confirmation back to the server with the final write operations to be performed. However, client reads are not considered as the protocol does not allow the player itself to decide whether the action is still desirable to be executed. Instead, it will be executed based on the conflicting data received. Thus, the protocol appears to provide the level of medium consistency.

Colyseus [4] uses primary-copy replication in its distributed architecture. It supports a rich query interface used for interest management. 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, which is orthogonal to the focus of our paper. Colyseus also considers missing or late updates by specifying game-specific bounds on the staleness of replicas, which we discussed in our models, especially at the lower levels of consistency. The paper also considers inconsistencies only at the replica level and not in the context of transactions or the resulting conflicts between the masters involved in an action.

Our low consistency model share similarities with weak consistency models in replicated databases [15] where read-only replicas have some form of bounded staleness. Such staleness factors have been explored widely in the database community. Materialized views, recently introduced for very large distributed data stores [16], provide concise but somewhat stale read-only replicas. However, while work on materialized views is concerned with keeping the views updated, they do not consider distributed update transactions that span data read from the view and updates to the base table. In [17], the authors discuss a middleware-based database caching system where a transaction can read stale data from the cache and then perform updates on the primary copy. The authors describe staleness limits that restrict how much the read cached versions may differ from the current value at the primary copy. Our medium and high consistency protocols have similarities with this approach but do not offer staleness limits. In contrast, our approach considers several other consistency levels and also considers distributed servers.

Our lower consistency categories rely on client-side computation to achieve the desired scalability. This approach is prone to cheating, as replica isolation is not maintained and allow for faulty client reads. This issue can be addressed by offloading computations to an external arbiter, such as another client [18].

IX. Conclusions

We argue that a generic solution for handling MMOG actions is not scalable. This paper proposes transactional models which offer varying consistency properties suitable for different MMOG actions. Action handling protocols can then be optimized accordingly. Our performance analysis shows that it is up to the developer to analyze each action separately and determine the suitable category. The goal is to make MMOGs scalable while maintaining the minimum consistency required for the semantics of the game.

References

[1] N. Gupta, A. Demers, J. Gehrke, P. Unterbrunner, and W. White, "Scalability for virtual worlds," *ICDE*, 2009.

[2] B. Dalton, "Online gaming architecture: Dealing with the real-time data crunch in mmos," in *GDC*, 2007.

[3] J. Kienzle, C. Verbrugge, B. Kemme, A. Denault, and M. Hawker, "Mammoth: a massively multiplayer game research framework," in *FDG*, 2009.

[4] A. Bharambe, J. Pang, and S. Seshan, "Colyseus: a distributed architecture for online multiplayer games," in *USENIX NSDI*, 2006.

[5] F. Glinka, A. Ploss, S. Gorlatch, and J. Müller-Iden, "High-level development of multiserver online games," *Int. J. Comput. Games Technol.*, 2008.

[6] T. Henderson, "Latency and user behaviour on a multiplayer game server," in *COST264 NGC*, 2001.

[7] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A critique of ANSI SQL isolation levels," in *SIGMOD*, 1995.

[8] J.-S. Boulanger, J. Kienzle, and C. Verbrugge, "Comparing interest management algorithms for massively multiplayer games," in *ACM SIGCOMM NetGames*, 2006.

[9] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza, "Towards scalable and transparent parallelization of multiplayer games using transactional memory support," in *SIGPLAN PPOPP*, 2010.

[10] A. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang, "Donnybrook: enabling large-scale, high-speed, peer-to-peer games," *SIGCOMM*, 2008.

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

[12] K. Zhang, B. Kemme, and A. Denault, "Persistence in massively multiplayer online games," in *ACM SIGCOMM NetGames*, 2008.

[13] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding," *Journal of Game Development*, vol. 1, 2004.

[14] N. Gupta, A. J. Demers, and J. E. Gehrke, "SEMMO: a scalable engine for massively multiplayer online games," in *SIGMOD*, 2008.

[15] A. Fekete, "Weak consistency models for replicated data," in *Encyclopedia of Database Systems*. Springer US, 2009.

[16] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan, "Asynchronous view maintenance for VLSD databases," in *SIGMOD*, 2009.

[17] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma, "Relaxed-currency serializability for middle-tier caching and replication," in *SIGMOD*, 2006.

[18] J. Goodman and C. Verbrugge, "A peer auditing scheme for cheat elimination in MMOGs," in *ACM SIGCOMM NetGames*, 2008.