Consistency Models for Cloud-based Online Games: the Storage System's Perspective

Ziqiang Diao
Otto-von-Guericke University Magdeburg
39106 Magdeburg, Germany
diao@iti.cs.uni-magdeburg.de

ABSTRACT

The existing architecture for massively multiplayer online role-playing games (MMORPG) based on RDBMS limits the availability and scalability. With increasing numbers of players, the storage systems become bottlenecks. Although a Cloud-based architecture has the ability to solve these specific issues, the support for data consistency becomes a new open issue. In this paper, we will analyze the data consistency requirements in MMORPGs from the storage system point of view, and highlight the drawbacks of Cassandra to support of game consistency. A timestamp-based solution will be proposed to address this issue. Accordingly, we will present data replication strategies, concurrency control, and system reliability as well.

1. INTRODUCTION

In massively multiplayer online role-playing games (MMORPG) thousands of players can cooperate with other players in a virtual game world. To support such a huge game world following often complex application logic and specific requirements. Additionally, we have to bear the burden of managing large amounts of data. The root of the issue is that the existing architectures of MMORPGs use RDBMS to manage data, which limits the availability and scalability.

Cloud data storage systems are designed for internet applications, and are complementary to RDBMS. For example, Cloud systems are able to support system availability and scalability well, but not data consistency. In order to take advantages of these two types of storage systems, we have classified data in MMORPGs into four data sets according to typical data management requirements (e.g., data consistency, system availability, system scalability, data model, security, and real-time processing) in [4]: account data, game data, state data, and log data. Then, we have proposed to apply multiple data management systems (or services) in one MMORPG, and manage diverse data sets accordingly. Data with strong requirements for data consistency and security (e.g., account data) is still managed by RDBMS, while data (e.g., log data and state data) that requires scalability and availability is stored in a Cloud data storage system (Cassandra, in this paper). Figure 1 shows the new architecture.

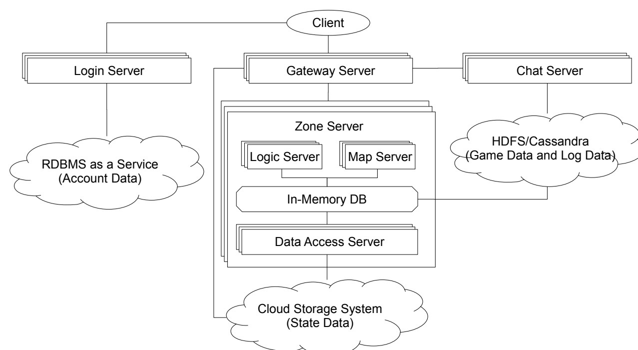

Figure 1: Cloud-based Architecture of MMORPGs [4]

Unfortunately, there are still some open issues, such as the support of data consistency. According to the CAP theorem, in a partition tolerant distributed system (e.g., an MMORPG), we have to sacrifice one of the two properties: consistency or availability [5]. If an online game does not guarantee availability, players' requests may fail. If data is inconsistent, players may get data not conforming to game logic, which affects their operations. For this reason, we must analyze the data consistency requirements of MMORPGs so as to find a balance between data consistency and system availability.

Although there has been some research work focused on the data consistency model of online games, the researchers generally discussed it from players' or servers' point of view [15, 9, 11], which actually are only related to data synchronization among players. Another existing research work did not process diverse data accordingly [3], or just handled this issue based on a rough classification of data [16]. However, we believe the only efficient way to solve this issue is to analyze the consistency requirements of each data set from the storage system's perspective. Hence, we organize the rest of this paper as follows: in Section 2, we highlight data consistency requirements of the four data sets. In Section 3, we discuss the data consistency issue of our Cloud-based architecture. We explain our timestamp-based solution in detail from Section 4 to Section 6. Then, we point out some optimization programs and our future work in Section 7. Finally,

25rd GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 28.05.2013 - 31.05.2011, Ilmenau, Deutschland.
Copyright is held by the author/owner(s).

we summarize this paper in Section 8.

2. CONSISTENCY REQUIREMENTS OF DIVERSE DATA IN MMORPGS

Due to different application scenarios, the four data sets have distinct data consistency requirements. For this reason, we need to apply different consistency models to fulfill them.

Account data: is stored on the server side, and is created, accessed as well as deleted when players log in to or log out of a game. It includes player's private data and some other sensitive information (e.g., user ID, password, and recharge records). The inconsistency of account data might bring troubles to a player as well as the game provider, or even lead to an economic or legal dispute. Imagine the following two scenarios: a player has changed the password successfully. However, when this player log in to the game again, the new password is not effective; a player has transferred to the game account, or the player has consumed in the game, but the account balance is somehow not properly presented in the game system. Both cases would influence on the player's experience, and might result in the customer or the economic loss of a game company. Hence, we need to access account data under strong consistency guarantees, and manage it with transactions. In a distributed database system, it means that each copy should hold the same view on the data value.

Game data: such as world appearance, metadata (name, race, appearance, etc.) of NPC (Non Player Character), system configuration files, and game rules, is used by players and game engine in the entire game, which can only be modified by game developers. Players are not as sensitive to game data as to account data. For example, the change of an NPC's appearance or name, the duration of a bird animation, and the game interface may not catch the players' attention and have no influence on players' operations. As a result, it seems that strong consistency for game data is not so necessary. On the other hand, some changes of the game data must be propagated to all online players synchronously, for instance, the change of the game world's appearance, the power of a weapon or an NPC, game rules as well as scripts, and the occurrence frequency of an object during the game. The inconsistency of these data will lead to errors on the game display and logic, unfair competition among players, or even a server failure. For this reason, we also need to treat data consistency of game data seriously. Game data could be stored on both the server side and the client side, so we have to deal with it accordingly.

Game data on the client side could only synchronize with servers when a player logs in to or starts a game. For this reason, causal consistency is required [8, 13]. In this paper, it means when player A uses client software or browser to connect with the game server, the game server will then transmit the latest game data in the form of data packets to the client side of player A. In this case, the subsequent local access by player A is able to return the updated value. Player B that has not communicated with the game server will still retain the outdated game data.

Although both client side and server side store the game data, only the game server maintains the authority of it. Furthermore, players in different game worlds cannot communicate to each other. Therefore, we only need to ensure that the game data is consistent in one zone server so that players in the same game world could be treated equally. It is noteworthy that a zone server accesses data generally from one data center. Hence, we guarantee strong consistency within one data center, and causal consistency among data centers. In other words, when game developers modify the game data, the updated value should be submitted synchronously to all replicas within the same data center, and then propagated asynchronously across data centers.

State data: for instance, metadata of PCs (Player Characters) and state (e.g., position, task, or inventory) of characters, is modified by players frequently during the game. The change of state data must be perceived by all relevant players synchronously, so that players and NPCs can respond correctly and timely. An example for the necessity of data synchronization is that players cannot tolerate that a dead character can continue to attack other characters. Note that players only access data from the in-memory database during the game. Hence, we need to ensure strong consistency in the in-memory database.

Another point about managing state data is that updated values must be backed up to the disk-resident database asynchronously. Similarly, game developers also need to take care of data consistency and durability in the disk-resident database, for instance, it is intolerable for a player to find that her/his last game record is lost when she/he starts the game again. In contrast to that in the in-memory database, we do not recommend ensuring strong consistency to state data. The reason is as follows: according to the CAP theorem, a distributed database system can only simultaneously satisfy two of three the following desirable properties: consistency, availability, and partition tolerance. Certainly, we hope to satisfy both consistency and availability guarantees. However, in the case of network partition or under high network latency, we have to sacrifice one of them. Obviously, we do not want all update operations to be blocked until the system recovery, which may lead to data loss. Consequently, the level of data consistency should be reduced. We propose to ensure read-your-writes consistency guarantee [13]. In this paper, it describes that once state data of player A has been persisted in the Cloud, the subsequent read request of player A will receive the updated values, yet other players (or the game engine) may only obtain an outdated version of it. From the storage system's perspective, as long as a quorum of replicas has been updated successfully, the commit operation is considered complete. In this case, the storage system needs to provide a solution to return the up-to-date data to player A. We will discuss it in the next section.

Log data: (e.g., player chat history and operation logs) is created by players, but used by data analysts for the purpose of data mining. This data will be sorted and cached on the server side during the game, and then bulk stored into the database, thereby reducing the conflict rate as well as the I/O workload, and increasing the total simultaneous throughput [2]. The management of log data has three features: log data will be appended continually, and its value will not be modified once it is written to the database; The replication of log data from thousands of players to multiple nodes will significantly increase the network traffic and even block the network; Moreover, log data is generally organized and analyzed after a long time. Data analysts are only concerned about the continuous sequence of the data, rather than the timeliness of the data. Hence, data inconsistency is accepted in a period of time. For these three reasons,

	Account data	Game data			State data		Log data
Modified by	Players	Game developers			Players		Players
Utilized by	Players & Game engine	Players & Game engine			Players & Game engine		Data analysts
Stored in	Cloud	Client side	Cloud		In-memory DB	Cloud	Cloud
Data center	Across	—	Single	Across	Single	Across	Across
Consistency model	Strong consistency	Causal consistency	Strong consistency	Causal consistency	Strong consistency	Read-your-writes consistency	Timed consistency

Table 1: Consistency requirements

a deadline-based consistency model, such as timed consistency, is more suitable for log data[12, 10]. In this paper, timed consistency specifically means that update operations are performed on a quorum of replicas instantaneously at time t, and then the updated values will be propagated to all the other replicas within a time bounded by t + $\triangle$ [10]. Additionally, to maintain the linear order of the log data, the new value needs to be sorted with original values before being appended to a replica. In other words, we execute a sort-merge join by the timestamp when two replicas are asynchronous. Under timed consistency guarantee, data analysts can at time t + $\triangle$ obtain a continuously sequential log data until time t.

3. OPPORTUNITIES AND CHALLENGES

In our previous work, we have already presented the capability of the given Cloud-based architecture to support the corresponding consistency model for each data set in MMORPGs [4]. However, we also have pointed out that to ensure read-your-writes consistency to state data and timed consistency to log data efficiently in Cassandra is an open issue. In this section, we aim at discussing it in detail.

Through customizing the quorum of replicas in response to read and write operations, Cassandra provides tunable consistency, which is an inherent advantage to support MMORPGs [7, 4]. There are two reasons: first, as long as a write request receives a quorum of responses, it completes successfully. In this case, although data in Cassandra is inconsistent, it reduces the response time of write operations, and ensures availability as well as fault tolerance of the system; Additionally, a read request will be sent to the closest replica, or routed to a quorum or all replicas according to the consistency requirement of the client. For example, if a write request is accepted by three (N, N> 0) of all five (M, M>= N) replicas, at least three replicas (M-N+1) need to respond to the subsequent read request, so that the up-to-date data can be returned. At this case, Cassandra can guarantee read-your-writes consistency or strong consistency. Otherwise, it can only guarantee timed consistency or eventual consistency [7, 13]. Due to the support of tunable consistency, Cassandra has the potential to manage state data and log data of MMORPGs simultaneously, and is more suitable than some other Cloud storage systems that only provide either strong or eventual consistency guarantees.

On the other hand, Cassandra fails to implement tunable consistency efficiently according to MMORPG requirements. For example, M-N+1 replicas of state data have to be compared so as to guarantee read-your-writes consistency. However, state data has typically hundreds of attributes, the transmission and comparison of which affect the read performance. Opposite to update a quorum of replicas, we update all replicas while executing write operations. In this case, data in Cassandra is consistent, and we can obtain the up-to-date data from the closest replica directly. Unfortunately, this replication strategy significantly increases the network traffic as well as the response time of write operations, and sacrifices system availability. As a result, to implement read-your-writes consistency efficiently becomes an open issue.

Another drawback is that Cassandra makes all replicas eventually consistent, which sometimes does not match the application scenarios of MMORPG, and reduce the efficiency of the system. The reasons are as follows.

- Unnecessary for State data: state data of a PC is read by a player from the Cloud storage system only once during the game. The subsequent write operations do not depend on values in the Cloud any more. Hence, after obtaining the up-to-date data from the Cloud, there is no necessity to ensure that all replicas reach a consensus on these values.
- Increase network traffic: Cassandra utilizes Read Repair functionality to guarantee eventual consistency [1]. It means that all replicas have to be compared in the background while executing a write operation in order to return the up-to-date data to players, detect the outdated data versions, and fix them. In MMORPGs, both state data and log data have a large scale, and are distributed in multiple data centers. Hence, transmission of these data across replicas will significantly increase the network traffic and affect the system performance.

4. A TIMESTAMP-BASED CONSISTENCY SOLUTION

A common method for solving the consistency problem of Cloud storage system is to build an extra transaction layer on top of the system [6, 3, 14]. Similarly, we have proposed a timestamp-based solution especially for MMORPG, which is designed based on the features of Cassandra [4]. Cassandra records timestamps in each column, and utilizes it as a version identification (ID). Therefore, we record the timestamps from a global server in both server side and in the Cloud storage system. When we read state data from the Cloud, the timestamps recorded on the server side will be sent with the read request. In this way, we can find out the most recent data easily. In the following sections, we will introduce this solution in detail.

4.1 Data Access Server

Data access servers are responsible for data exchange between the in-memory database and the Cloud storage sys-

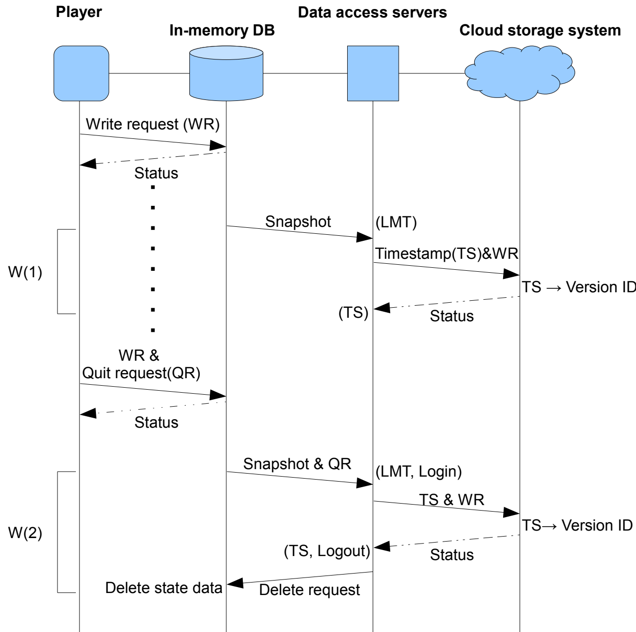

Figure 2: Executions of Write Operations: W(1) describes a general backup operation; W(2) shows the process of data persistence when a player quits the game.

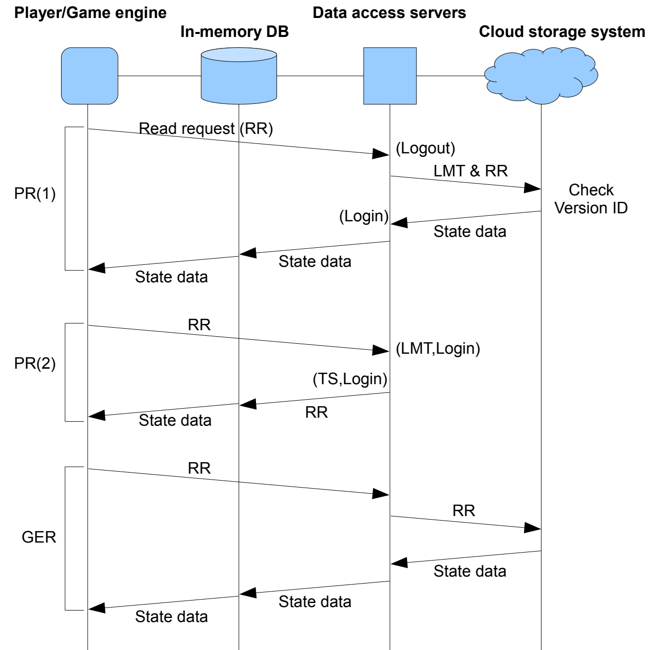

Figure 3: Executions of Read Operations: PR(1) shows a general read request from the player; In the case of PR(2), the backup operation is not yet completed when the read request arrives; GER presents the execution of a read operation from the game engine.

tem. They ensure the consistency of state data, maintain timestamp tables, and play the role of global counters as well. In order to balance the workload and prevent server failures, several data access servers run in one zone server in parallel. Data access servers need to synchronize their system clock with each other automatically. However, a complete synchronization is not required. A time difference less than the frequency of data backup is acceptable.

An important component in data access servers is the timestamp table, which stores the ID as well as the last modified time (LMT) of state data, and the log status (LS). If a character or an object in the game is active, its value of LS is "login". Otherwise, the value of LS is "logout". We utilize a hash function to map IDs of state data to distinct timestamp tables, which are distributed and partitioned in data access servers. It is noteworthy that timestamp tables are partitioned and managed by data access servers in parallel and data processing is simple, so that accessing timestamp tables will not become a bottleneck of the game system.

Note that players can only interact with each other in the same game world, which is managed by one zone server. Moreover, a player cannot switch the zone server freely. Therefore, data access servers as well as timestamp tables across zone servers are independent.

4.2 Data Access

In this subsection, we discuss the data access without considering data replication and concurrency conflicts.

In Figure 2, we show the storage process of state data in the new Cloud-based architecture: the in-memory database takes a consistent snapshot periodically. Though using the same hash function employed by timestamp tables, each data access server obtains the corresponding state data from the snapshot periodically. In order to reduce the I/O workload of the Cloud, a data access server generates one message including all its responsible state data as well as a new timestamp TS, and then sends it to the Cloud storage system. In the Cloud, this message is divided based on the ID of state data into several messages, each of which still includes TS. In this way, the update failure of one state data won't block the submission of other state data. Then, these messages are routed to appropriate nodes. When a node receives a message, it writes changes immediately into the commit log, updates data, and records TS as version ID in each column. If an update is successful and TS is higher than the existing LMT of this state data, then the data access server uses TS to replace the LMT. Note that if a player has quit the game and the state data of the relevant PC has backed up into the Cloud storage system, the LS of this PC needs to be modified form "login" to "logout", and the relevant state data in the in-memory database needs to be deleted.

Data access servers obtain log data not from the in-memory database, but from the client side. Log data also updates in batch, and gets timestamp from a data access server. When a node in the Cloud receives log data, it inserts log data into its value list according to the timestamp. However, timestamp tables are not modified when the update is complete.

Figure 3 presents executions of read operations. When a player starts the game, a data access server firstly obtains the LS information from the timestamp table. If the value is "login", that means the previous backup operation is not

completed and the state data is still stored in the in-memory database. In this case, the player gets the state date from the in-memory database directly, and the data access server needs to generate a new timestamp to replace the LMT of the relevant state data; if the value is "logout", the data access server then gets the LMT, and sends it with a read request to the Cloud storage system. When the relevant node receives the request, it compares the LMT with its local version ID. If they match, the replica responds the read request immediately. If not match, this read request will be sent to other replicas (we will discuss it in detail in the section 5). When the data access server receives the state data, it sends it to the in-memory database as well as the relevant client sides, and modifies the LS from "logout" to "login" in the timestamp table. Note that state data may also be read by the game engine for the purpose of statistics. In this case, the up-to-date data is not necessary, so that we do not need to compare the LMT with the Version ID.

Data analysts read data also through data access servers. If a read request contains a timestamp T, the cloud storage system only returns log data until T-$\triangle$ because it only guarantees log data timed consistency.

4.3 Concurrency Control

Concurrency conflicts appear rarely in the storage layer of MMORPGs: the probability of read-write conflicts is low because only state data with a specific version ID (the same as its LMT) will be read by players during the game, and a read request to log data does not return the up-to-date data. Certain data is periodically updated by only one data access server simultaneously. Therefore, write-write conflicts occur only when the per-update is not completed for some reason, for example, serious network latency, or a node failure. Fortunately, we can solve these conflicts easily by comparing timestamps. If two processes attempt to update the same state data, the process with higher timestamp wins, and another process should be canceled because it is out of date. If two processes intend to update the same log data, the process with lower timestamp wins, and another process enters the wait queue. The reason is that values contained in both processes must be stored in correct order.

5. DATA REPLICATION

Data in the Cloud typically has multiple replicas for the purpose of increasing data reliability as well as system availability, and balancing the node workload. On the other hand, data replication increases the response time and the network traffic as well, which cannot be handled well by Cassandra. For most of this section, we focus on resolving this contradiction according to access features of state data and log data.

5.1 Replication Strategies

Although state data is backed up periodically into the Cloud, only the last updated values will be read when players start the game again. It is noteworthy that the data loss in the server layer occurs infrequently. Therefore, we propose to synchronize only a quorum of replicas during the game, so that an update can complete effectively and won't block the subsequent updates. In addition, players usually start a game again after a period of time, so the system has enough time to store state data. For this reason, we propose to update all replicas synchronously when players quit the game. As a result, the subsequent read operation can obtain the updated values quickly.

While using our replication strategies, a replica may contain outdated data when it receives a read request. Though comparing LMT held by the read request with the Version ID in a replica, this case can be detected easily. Contrary to the existing approach of Cassandra (compares M-N+1 replicas and utilizes Read Repair), only the read request will be sent to other replicas until the lasted values was found. In this way, the network traffic will not be increased significantly, and the up-to-date data can also be found easily. However, if the read request comes from the game engine, the replica will respond immediately. These strategies ensure that this Cloud-based architecture can manage state data under read-your-writes consistency guarantees.

Similar to state data, a write request to log data is also accepted by a quorum of replicas at first. However, the updated values then must be propagated to other replicas asynchronously when the Cloud storage system is not busy, and arranged in order of timestamp within a predetermined time ($\triangle$), which can be done with the help of Anti-Entropy functionality in Cassandra [1]. In this way, this Cloud storage system guarantees log data timed consistency.

5.2 Version Conflict Reconciliation

When the Cloud storage system detected a version conflict between two replicas: if it is state data, the replica with higher version ID wins, and values of another replica will be replaced by new values; if it is log data, these two replicas perform a sort-merge join by timestamps for the purpose of synchronization.

6. SYSTEM RELIABILITY

Our Cloud-based architecture for MMORPGs requires a mutual cooperation of multiple components. Unfortunately, each component has the possibility of failure. In the following, we discuss measures to deal with different failures.

Cloud storage system failure: the new architecture for MMORPGs is built based on Cassandra, which has the ability to deal with its own failure. For example, Cassandra applies comment logs to recover nodes. It is noteworthy that by using our timestamp-based solution, when a failed node comes back up, it could be regarded as an asynchronous node. Therefore, the node recovery as well as response to write and read requests can perform simultaneously.

In-memory database failure: similarly, we can also apply comment logs to handle this kind of failure so that there is no data loss. However, writing logs affects the real-time response. Moreover, logs are useless when changes are persisted in the Cloud. Hence, we have to find a solution in our future work.

Data access server failure: If all data access servers crash, the game can still keep running, whereas data cannot be backed up to the Cloud until servers restart, and only players already in the game can continue to play; Data access servers have the same functionality and their system clocks are relatively synchronized, so if one server is down, any other servers can replace it.

Timestamp table failure: We utilize the primary/secondary model and the synchronous replication mechanism to maintain the reliability of timestamp tables. In the case of all replicas failure, we have to apply the original feature of Cassandra to obtain the up-to-date data. In other words, M-

N+1 replicas need to be compared. In this way, we can rebuild timestamp tables as well.

7. OPTIMIZATION AND FUTURE WORK

When a data access server updates state data in the Cloud, it propagates a snapshot of state data to multiple replicas. Note that state data has hundreds of attributes, so the transmission of a large volume of state data may block the network. Therefore, we proposed two optimization strategies in our previous work [4]: if only some less important attributes of the state (e.g., the position or orientation of a character) are modified, the backup can be skipped; Only the timestamp, ID, and the modified values are sent as messages to the Cloud. However, in order to achieve the second optimization strategy, our proposed data access approach, data replication strategies, and concurrency control mechanism have to be changed. For example, even during the game, updated values must be accepted by all replicas, so that the subsequent read request does not need to compare M-N+1 replicas. We will detail the adjustment program in our future work.

It is noteworthy that a data access server stores a timestamp repeatedly into the timestamp table, which increases the workload. A possible optimization program is as follows: If a batch write is successful, data access server caches the timestamp (TS) of this write request. Accordingly, in the timestamp table, we add a new column to each row to maintain a pointer. If a row is active (the value of LS is "login"), the pointer refers to the memory location of TS; if not, it refers to its own LMT. When a row becomes inactive, it uses TS to replace its LMT. In this way, the workload of a timestamp table will reduce significantly. However, LMT and Version ID of state data may be inconsistent due to the failure of the Cloud storage system or the data access server.

8. CONCLUSIONS

Our Cloud-based architecture of MMORPGs can cope with data management requirements regarding availability and scalability successfully, while supporting data consistency becomes an open issue. In this paper, we detailed our timestamp-based solution in theory, which will guide the implementation work in the future. We analyzed the data consistency requirements of each data set from the storage system's perspective, and studied methods of Cassandra to guarantee tunable consistency. We found that Cassandra cannot ensure read-your-writes consistency for state data and timed consistency for log data efficiently. Hence, we proposed a timestamp-based solution to improve it, and explained our idea for concurrency control, data replication strategies, and fault handling in detail. In our future work, we will implement our proposals and the optimization strategies.

9. ACKNOWLEDGEMENTS

Thanks to Eike Schallehn for his comments.

10. REFERENCES

[1] Apache. Cassandra, January 2013. http://cassandra.apache.org/.

[2] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Lt'eon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research(CIDR)*, pages 223–234, Asilomar, California, USA, 2011.

[3] S. Das, D. Agrawal, and A. E. Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Symposium on Cloud Computing(SoCC)*, pages 163–174, Indianapolis, Indiana, USA, 2010.

[4] Z. Diao and E. Schallehn. Cloud Data Management for Online Games : Potentials and Open Issues. In *Data Management in the Cloud (DMC)*, Magdeburg, Germany, 2013. Accepted for publication.

[5] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM Special Interest Group on Algorithms and Computation Theory (SIGACT)*, 33(2):51–59, 2002.

[6] F. Gropengieß er, S. Baumann, and K.-U. Sattler. Cloudy transactions cooperative xml authoring on amazon s3. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 307–326, Kaiserslautern, Germany, 2011.

[7] A. Lakshman. Cassandra - A Decentralized Structured Storage System. *Operating Systems Review*, 44(2):35–40, 2010.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] F. W. Li, L. W. Li, and R. W. Lau. Supporting continuous consistency in multiplayer online games. In *12. ACM Multimedia 2004*, pages 388–391, New York, New York, USA, 2004.

[10] H. Liu, M. Bowman, and F. Chang. Survey of state melding in virtual worlds. *ACM Computing Surveys*, 44(4):1–25, 2012.

[11] W. Palant, C. Griwodz, and P. l. Halvorsen. Consistency requirements in multiplayer online games. In *Proceedings of the 5th Workshop on Network and System Support for Games, NETGAMES 2006*, page 51, Singapore, 2006.

[12] F. J. Torres-Rojas, M. Ahamad, and M. Raynal. Timed consistency for shared distributed objects. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing - PODC '99*, pages 163–172, Atlanta, Georgia, USA, 1999.

[13] W. Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, 2008.

[14] Z. Wei, G. Pierre, and C.-H. Chi. Scalable Transactions for Web Applications in the Cloud. In *15th International Euro-Par Conference*, pages 442–453, Delft, The Netherlands, 2009.

[15] K. Zhang and B. Kemme. Transaction Models for Massively Multiplayer Online Games. In *30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011)*, pages 31–40, Madrid, Spain, 2011.

[16] K. Zhang, B. Kemme, and A. Denault. Persistence in massively multiplayer online games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games, NETGAMES 2008*, pages 53–58, Worcester, Massachusetts, USA, 2008.