Applying Database Replication to Multi-player Online Games*

Yi Lin
McGill Univ.
Montreal
ylin30@cs.mcgill.ca

Bettina Kemme
McGill Univ.
Montreal
kemme@cs.mcgill.ca

Marta Patiño-Martínez
Univ. Politecnica de Madrid
mpatino@fi.upm.es

Ricardo Jiménez-Peris
Univ. Politecnica de Madrid
rjimenez@fi.upm.es

ABSTRACT

Multi-player Online Games (MOGs) have emerged as popular data intensive applications in recent years. Being used by many players simultaneously, they require a high degree of fault tolerance, scalability and performance. In this paper we analyze how database replication can be used in MOGs to achieve these goals. In data replication, clients can read data from any database replica while updates have to be executed at all available replicas. Thus, reads can be distributed among the replicas leading to reduced response time and scalability. Furthermore, the system is fault-tolerant as long as a replica is available. However, we are not aware of any previous study on the application of database replication to MOG. In this paper, we present a system, MiddleSIR, which provides database replication support. We illustrate different replication protocols implemented in the system along an example, explaining how data consistency and fault tolerance can be achieved. From there, we design a small multi-player typing game to demonstrate how to apply database replication to MOG. We will discuss how different replication protocols affect the semantics of the game. Our experiments show that database replication can provide good scalability and performance in both Local Area Networks (LAN) and Wide Area Networks (WAN).

1. INTRODUCTION

1.1 Multi-player Online Games

Multi-player Online Games (MOG) represent a very popular application area which requires technology support from many different domains, such as databases, graphics, and networks. The general architecture for MOG is the client-server architecture. Servers are responsible for storing and managing all the data related to game state (such as players' credits, equipment, and position of players and objects). Clients are responsible for displaying these data graphically. MOGs are a very data intensive application. As players perform actions, the game state continuously changes, and all these changes should be visible to all players in real-time. Currently, many online games have a central server to which all clients are connected. This limits the fault tolerance, scalability, and performance of the games.

Fault tolerance is a crucial issue since any down-time of the game server has a negative effect on the "game experience". For example, when the server running Blizzard's World of Warcraft crashed on Jan 21, 2006, thousands of players complained, which finally led to player compensation [13].

Furthermore, as more players join a game, the central server becomes overloaded. It is desirable to scale the system to accept more clients. Current solutions partition the game world into zones. Each player is assigned to a zone and only sees actions within this zone. With this, different zones can be handled independently by different servers, and each server has only a small number of clients. These solutions for scalability limit the total number of players which can play with each other (i.e., only those in the same zone can play with each other). Moreover, if a player wants to move from one zone to another, a complex state transfer and coordination protocol is needed.

Finally, a centralized solution might provide different quality of service to the different players. If a player is close to the central server, it will receive fast response to its queries, while the response times for remote players might be worse.

1.2 Database Replication

Clients interact with the database by issuing *transactions*. A transaction has one or more *read* and/or *write* requests, and terminates with a *commit*/*abort* operation. A transaction is atomic in the sense that either all or none of its write requests will persist in the database depending on whether it commits or aborts.

In a replicated database, there are several copies of the database (replicas) at different sites. A replication protocol is in charge of keeping data consistent at all replicas all the time. That is, after a transaction commits, all copies of the database should have the updates of the transaction. Ideally, a replicated database should behave as if there is only one database. Most of the replication protocols follow the Read-One-Write-All-Available (ROWAA) approach. A write request is executed in all database replicas while a read

*This work has been partially supported by the Spanish Research Council, MEC and MITYC, under projects TIN2004-07474-C02-01, FIT-340000-2005-144, and by the Madrid Regional Research Council under project TIC-000285-0505.

Netgames 2006, Oct 30-31, 2006, Singapore

request only in one database. Reading from one replica is possible because the replicas all have the same state.

Recent research [19, 5, 8] has shown that ROWAA based database replication can improve fault tolerance, scalability, and performance of the system. Fault-tolerance means that in the case a replica crashes, the data is still available because clients can connect to one of the available replicas. Performance is achieved since a client can read data from the a database replica that is geographically close to it (it is referred to as a local replica). Scalability can be achieved since the system is able to handle increasing numbers of read requests by adding new replicas to the system. Note that the system cannot scale if all requests are writes, because in ROWAA a write request must be executed at all databases.

1.3 Applying Database Replication to MOG

Looking at the properties database replication can provide, it seems very promising to apply replication technology to MOGs. Being fault-tolerant, MOGs can provide continuous service to their clients. Furthermore, clients can be distributed to different MOG server replicas. Thus each MOG server only needs to serve a few clients similar to when zone partitioning is used. In contrast to zone partitioning, however, all clients can access the same (replicated) game world. Regarding performance, each client can connect to a close replica, thus the response time for all clients can be short and relatively similar. In MOGs, there are many updates (player actions). These have to be executed at all servers and must be synchronized with read operations. Data replication provides techniques to propagate and apply all updates efficiently at all replicas. Furthermore, it provides concurrency control for any read/write synchronization.

Although it seems promising to use database replication for MOGs we are not aware of any system that attempts to develop a MOG architecture based on database replication. Thus, it is very unclear how exactly replication technology can be applied to MOG in a real scenario, what the challenges will be, how the response time and scalability will look like, and if different replication protocols have an impact on the design of MOG. In this paper, we answer these questions by designing a small multi-player typing game using database replication.

We first present a framework, MiddleSIR which accommodates several typical replication protocols in Section 2. Then, in Section 3 we illustrate those replication protocols by using an example execution, explaining how these protocols achieve data consistency and fault tolerance. In Section 4 we show the design of the multi-player typing game on top of MiddleSIR. Section 5 shows the results of our experiments in both LAN and WAN. In Section 6 we discuss some related work. Section 7 concludes the paper.

2. MIDDLESIR SYSTEM ARCHITECTURE

In [8], we developed a framework, MiddleSIR, which allows the implementation of different database replication protocols. Figure 1 shows the architecture of MiddleSIR. Each site has a MiddleSIR replica and a database replica. MiddleSIR provides a standard JDBC database interface to its clients, i.e., to the clients, MiddleSIR appears to be the database backend. We implemented MiddleSIR in such a way that it allows easy plug-in of different replication protocols. There are three components (i.e., communication, transaction and connection managers) in a MiddleSIR

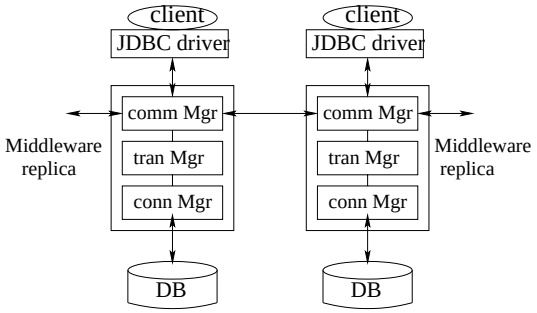


Figure 1: MiddleSIR architecture

replica. When a client submits an operation (such as an SQL SELECT or UPDATE, or a transaction commit), the JDBC driver forwards the operation to the communication manager of a MiddleSIR replica. The communication manager takes care of sending and receiving messages to individual or all sites. The transaction manager implements the replication protocols. Upon receiving messages from its communication manager, it will perform replica control to synchronize the commit/abort of concurrent conflicting transactions. The transaction manager will ask the connection manager to perform any database operations in the database.

The architecture allows for the implementation of different replication protocols. Different protocols have different combinations of communication manager and transaction manager. For example, two protocols may share the same communication manager because their communication mechanisms are the same, for instance they use TCP/IP sockets. Currently, MiddleSIR only works with PostgreSQL. Thus, all protocols share the same connection manager.

If a middleware replica fails, the clients connected to that failed middleware replica will be automatically reconnected to another middleware replica. Thus the system provides transparent fault-tolerance. The failover logic is implemented within the JDBC driver. The driver keeps the IP addresses of other middleware replicas. When it looses the connection to the middleware replica is connected with, the driver connects to another automatically.

3. REPLICATION PROTOCOLS

In order to build a MOG on top of a replicated database, it is good to know how database replication works. In a database system, all operations are executed within the context of transactions. A transaction is a sequence of read and write operations on data items of the database. From the application point of view, it represents a logical unit of work. The application indicates the termination of a transaction with a commit request. Transactions provide atomicity, i.e., either all of their operations succeed and the transaction commits, or the transaction aborts, and does not change any state. In the abort case, if the transaction has already executed some write operations, they have to be undone. Furthermore, the concurrent execution of transactions has to be controlled such that no inconsistencies occur. The concurrency control module of a database guarantees that the execution of concurrent transactions is equivalent to a serial execution of the same transactions. This issue is more complicated in a replicated environment, since concurrent

updates can occur at different replicas. For example, assume transaction T_1 sets the value of a data item x to 1 and transaction T_2 sets the value of x to 2 (We say that T_1 and T_2 have a *write/write* conflict in this case). It is possible that T_1 and T_2 are submitted to two different sites A and B concurrently. If T_1 commits before T_2 at site A while T_1 after T_2 at site B, the final values of x after both T_1 and T_2 commit at site A and B will be different (i.e, x is 2 in site A while 1 in site B). It is the task of replica control to handle conflicts occurring at different sites. Some replication protocols schedule T_1 and T_2 to execute sequentially in a unique order at all sites. Others will only allow either T_1 or T_2 to be able to commit at all sites.

[3] categorizes different replication protocols based on two factors, i.e., where updates can be executed and when the changes of a transaction are propagated to the rest of the replicas. The protocol is called a *primary copy* approach if all write transactions must be executed by a given replica, called the primary. The primary is responsible for propagating the updates to the secondary replicas which apply them. The secondary copies themselves may only execute read operations from clients. By executing all update transactions at a single site, the local concurrency control mechanism of the database will detect and handle any conflicts. In contrast, if transactions with write operations can be executed at any replica, the protocol follows an *update everywhere* approach. Replica control then has to schedule transactions globally. A replication protocol is *lazy* if the changes of a transaction are propagated to the rest of the replicas after that transaction commits Otherwise, the replication protocol is *eager*. All replication protocols are a combination of these two factors, i.e., either lazy or eager, and either primary copy or upate everywhere. Actually, the previous example corresponds to the execution of a *lazy update everywhere* approach. Since lazy update everywhere protocols may produce inconsistencies which are very difficult to resolve, we will not consider them further. In fact, most existing replication protocols fall into the categories of *lazy primary copy* or *eager update everywhere* approaches.

In the following we show how some of these protocols work by using an illustrating example. In the example, three transactions T_1, T_2 and T_3 are submitted concurrently to three sites (called their *local* sites) respectively. In the following we denote as $r_i(x)$ ($w_i(x)$) that transaction T_i has read (write) operation on data item x. A c_i denotes the time where the commit request for T_i is submitted. T_1 is a read-only transaction with a single read $r_1(x)$. T_2:$w_2(x)$ and T_3:$r_3(x), w_3(x)$ are write transactions with a write/write conflict. We will show how different replication protocols achieve data consistency.

3.1 Lazy Primary

Our first approach is a lazy primary copy approach similar to the one proposed by Ganymed [14]. It requires that transactions must be marked as read-only or not at their submission times. All clients are connected to a primary MiddleSIR replica and the individual operations (read/write/commit requests) are submitted via the JDBC driver. Our MiddleSIR version of a primary copy approach allows a client to submit all its transactions to its local MiddleSIR replica. If a transaction is an update transaction, each individual operation will be forwarded to the primary.

As shown in Figure 2, a read-only transaction T_1 can execute and commit locally without contacting the primary (site B). For a write transaction, all of its operations must be submitted to the primary for execution. In Figure 2, the local replica of T_2 is the primary so that the primary MiddleSIR will simply forward $w_2(x)$ to the primary database replica for execution. T_3 is not submitted to the primary so that its local MiddleSIR replica (called secondary MiddleSIR replica) forwards it to the primary MiddleSIR replica. The primary MiddleSIR replica then forwards $r_3(x)$ to the primary database replica for execution. The response of $r_3(x)$ is sent back to the client in Site C and the client submits the next operation $w_3(x)$ which is again forwarded to the primary replica. T_2 is successfully committed at the primary and the primary replica sends the writeset (the transaction changes) to all secondaries where they are applied. $w_3(x)$ is also successfully executed. But when the commit request c_3 arrives, the primary database replica detects a conflict and aborts T_3. The abort message is sent back to the client. That is, all sites apply T_2 while T_3 aborts.

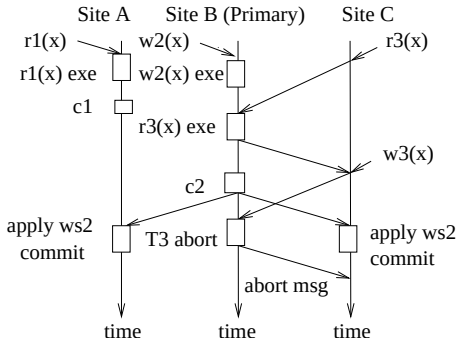


Figure 2: Lazy Primary Copy Example

3.2 Symmetric

[7] presents three eager update everywhere approaches using a group communication system (GCS). These protocols are based on the protocol presented in [5]. In here, we present the basic idea of one of the protocols, *Symmetric*. A client submits all operations associated by a transaction to its local MiddleSIR replica. Read-only transactions can execute and commit locally. Write transactions are executed at all replicas in the same order. For that, the local MiddleSIR replica multicasts the update transaction to all MiddleSIR replicas using a *uniform reliable* and *total order* multicast provided by the GCS. Uniform reliable multicast guarantees that all sites will deliver a message unless the site crashes. Total order multicast guarantees that all sites will deliver messages in the same order. Thus, all MiddleSIR replicas receive write transactions in the same order and execute them on their local databases according to that order.

Figure 3.2 illustrates Symmetric with the benchmark example. T_1, T_2, and T_3 are submitted to their local replicas. Note that all operations of a transaction must be known when the transaction starts. By checking the SQL statements, site A knows that T_1 is a read-only transaction so, it can execute and commit T_1 locally. Site B and C multicast $T_2 : w_2(x)$ and $T_3 : r_3(x), w_3(x)$ in uniform reliable and total order. T_2 is delivered before T_3 at all replicas. Hence, T_2 is scheduled to execute before T_3.

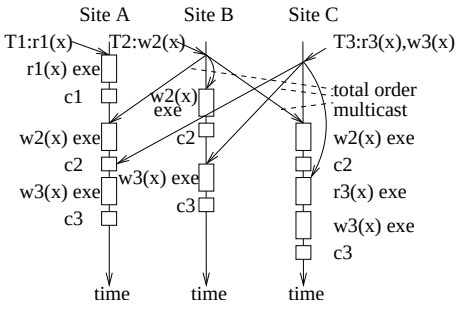


Figure 3: Symmetric example

3.3 SRCA-REP

SRCA-REP [8] is an eager update everywhere approach using GCS which does not have any restrictions on the interface. The basic idea is as follows. A client executes all operations of a transaction in its local database replica before commit (using standard JDBC driver). After a local transaction successfully finishes execution, it submits its commit request. A read-only transaction can commit immedately, For update transactions, the writeset is multicast in uniform reliable and total order to all replicas MiddleSIR replicas. Each replica then checks if there are any conflicting transactions executing concurrently in other replicas. The checking (called *validation*) is performed in all replicas according to the delivery order. If no other concurrent conflicting transaction is validated before, a transaction is successfully validated and allowed to commit. Otherwise, it will be aborted in its local replica and discarded in other replicas. SRCA-REP uses the writeset to detect conflicts among transactions and a timestamp to detect concurrent transactions. A writeset contains the changes made by a transaction identified by the primary keys. Two transactions conflict if the intersection of the primary keys of their writesets is non empty. SRCA-REP records the start time and end time of a transaction in terms of an incremental timestamp in the local replica. The timestamp of a replica will be increased consecutively if one transaction succeeds in validation. If a transaction's start time is between the start time and end time of the other transaction, they are concurrent. For more details, please refer to [8].

Figure 4 shows our example with SRCA-REP. T_1, T_2, and T_3 are submitted operation by operation to site A, B, and C respectively. They all successfully execute in their local replicas and submit their commit requests. Since T_1 is read-only it can commit locally. The writesets of T_2 and T_3 are multicast in uniform reliable and total order. In all replicas, T_2 performs validation before T_3 because T_2 is delivered before T_3. Thus, T_2 succeeds its validation while T_3 fails. Then all replicas apply T2's writeset (if not the local site of T_2) and commit. T_3 is aborted in its local replica.

3.4 SEQ

SEQ [9] is similar to SRCA-REP except that it does not use a GCS. In order to reach the same validation decision at all replicas, SEQ uses a replica as the sequencer. All replicas send their writesets to the sequencer for validation. The sequencer sends the validation results (including writesets if the validation succeeds) to all replias in the validation order. All replicas will apply the writesets according to the validation order. Thus, the data are consistent.

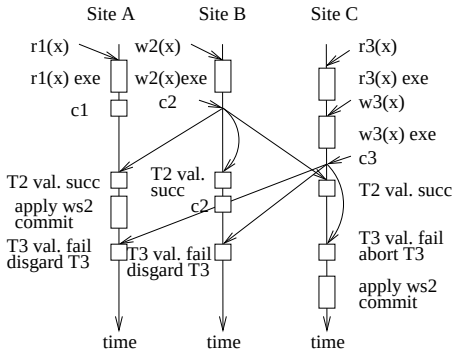


Figure 4: SRCA-REP example

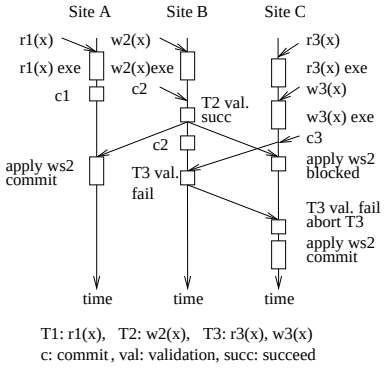
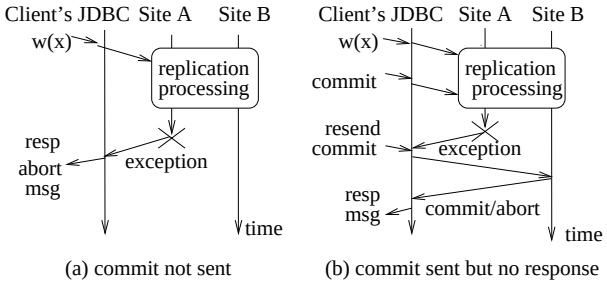


Figure 5: SEQ example

Figure 5 applies SEQ to our benchmark example. Site B is the sequencer replica. T_1 is read-only so it can execute and commit locally. T_2 and T_3 must perform validation in site B. T_2's local replica is the sequencer so that it can validate immediately upon commit request. The validation succeeds and T_2's writeset is sent to all other replicas. Applying T_2's writeset will be blocked in site C since T_3 is writing on the same data item x. Since T_3's local replica is not the sequencer, it must send its writeset to site B for validation. The validation fails because T_2 is concurrent and has already been validated. An abort message will be sent back to site C to abort T_3. After the abort of T_3, T_2's writeset will be applied and committed.

3.5 Fault-tolerance

We mentioned before that when a MiddleSIR replica fails the JDBC driver of a client connected to this replica will automatically be redirected to another MiddleSIR replica. However, there might be a transaction being executed (called active transaction) when the replica crashed. The active transaction either (i) has not submitted a commit request or (ii) has submitted the commit request but has not received the response yet. In case (i), the JDBC driver can just simply return an abort message to the client since the changes made by the transaction have not been written permanently to the database yet (shown in Figure 6.(a)). In case (ii), the transaction may have been committed but the



(a) commit not sent (b) commit sent but no response

Figure 6: MiddleSIR failover

response message is lost, or its commit has not taken place when the replica crashed. Hence, the JDBC driver has to resend the commit request to the newly connected replica to know the decision. This is shown in Figure 6.(b).

In the lazy primary copy approach, the primary copy presents a single point of failure. Ganymed resolves this by using a backup primary. All write requests go through the backup primary site before they execute in the primary site. When the primary site crashes, the backup primary site will take over. When the driver reconnects to a different replica which does not know the outcome of the transaction (case (b) in the figure), this replica can ask the new primary for the outcome The Symmetric and SRCA-REP use uniform reliable multicast. Thus, it is guaranteed that a transaction is received by all surviving sites. Hence, when the driver reconnects to a new replica and asks for the outcome, the decision is known. Since SEQ does not use uniform reliable delivery it might be that the sequencer validates and commits a transaction but no other site receives the writeset of this transaction before the sequencer fails. Thus, the transaction is lost. As with the lazy primary copy approach, we could have redirected all writesets through a backup sequencer to avoid this problem.

4. A MULTI-PLAYER TYPING GAME

In the last two sections, we have shown a middleware-based framework, MiddleSIR, providing replicated database functionalities, and illustrated how different replication protocols achieve data consistency and fault tolerance. In this section, we will show how to design a multiplayer typing game on top of MiddleSIR. Although it is a small game, it is designed to show the basic rationale of how to exploit database replication for multi-player games.

4.1 Game description

The game allows users to practice keyboard typing. Figure 7 shows the Graphical User Interface (GUI) of the game. The GUI is mainly composed of a game field in the center, a score panel on the right, and a typing text area at the bottom. Strings in black color appear at the top of the game field, then move to the bottom at a certain speed, where they disappear from the field. The goal is to type the strings before they disappear. Once a player has successfully typed a string, he/she gets points for the string and the string appears in the color of the player. Then, automatically, a bullet with the same color will be fired from the bottom of the game field to shoot the string and let it disappear.

The system provides a set of strings to players. Typically, players will see the same strings. We have also implemented

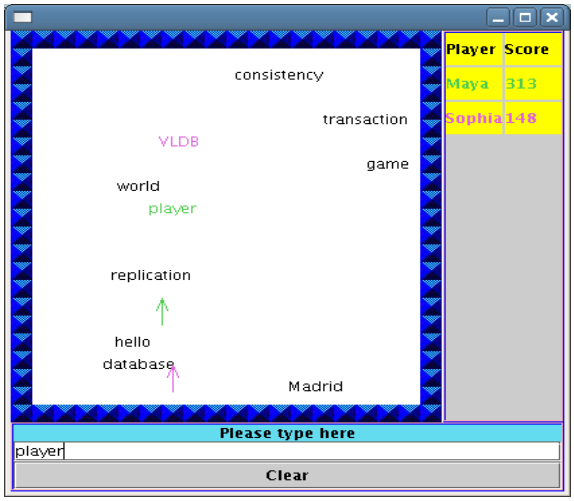


Figure 7: Typing Game GUI

Thread	Task at certain time interval
background thread	Move strings and bullets
updating thread of each player	process characters typed by player
reading thread of each player	read game states (e.g., strings, bullets, scores etc)

Table 1: Analysis of thread tasks at certain time interval

a publish/subscribe scheme where each player can subscribe to a subset of strings. Thus players might have different, but possibly overlapping string sets. For space limitations we do not discuss the publish/subscribe system in more detail. On the score panel, each player will see the scores of other players. Once a player successfully types a string, his/her score will be increased, and all related players' score panels will be updated.

4.2 Analysis based on threads

The typing game is inherently a multithreaded application because it has lots of parallel actions. We identify several threads and their regular tasks in Table 1.

Initially a background process will load a list of strings from a file and publish them. In order to simulate the movement of strings and bullets, the background process updates the positions of strings and bullets at certain time intervals (e.g., increase Y values of strings by 10 pixels every 500 ms).

Upon starting, the game client instance of each player starts an updating thread and a reading thread. The updating thread accepts input from the player. The game semantics guarantees that all words shown on the game field start with different characters. Thus, when a player enters the first character, the update thread can determine which word the player tries to type. After finding a target string, the update thread will check if the following typing is correct. If yes, the correctly typed character will be appended to the string's typing buffer. If the whole string is typed completely, a bullet will be fired and the points for the string will be added to the player's score.

The reading thread is responsible for refreshing the game field of the player. It determines the full game state infor-

mation (such as player scores, strings, bullets, their colors, and their positions) each time interval (e.g., 1000 ms), and presents it on the client screen.

4.3 Modelling based on transactions

Looking at the thread implementation above, the game is actually a reader-writer problem on game state (such as strings, bullets and players). While the state is being read, it might be changed by different events. For example, if a player types a string correctly, the string's color will be changed to the player's color. The points for the string will be added to the player's score while the score is being read. Conflicting writers might also run concurrently. For example, two players may type the same string at the same time. In sum, there is a considerable amount of concurrency control issues. These issues become only more severe if any kind of distributed architecture is used.

However, by using a database system for storing game relevant information, the database system will automatically take care of these concurrency issues. Recall that transactions are atomic and that the database system isolates concurrent transactions via concurrency control mechanisms. Thus any conflicts are detected and resolved, guaranteeing the consistency of the data.

This still holds true if we use a replicated database (such as in Figure 1), since the replication protocol takes further care of concurrency control problems across the entire distributed architecture. That is, it provides the application the illusion that there is only a single logical database. Thus, the game itself does not even need to be aware of the distribution.

4.4 Game and database design

As typical for MOGs, we have implemented the typing game in a client/server architecture. Each game player runs a game client application with an updating and reading thread as described above. There is also an independent application which runs a so-called background thread. We will discuss this application shortly. Both the game client application and the application running the background thread are clients of our MiddleSIR (please refer to Figure 1).

In our design, we store all game state information in the database. The schema of the database is shown in Table 2. There are four tables, i.e., player, string, bullet, and color. A player tuple has attributes like name, score, color ID (i.e., cid), the ID of the string being typed (i.e., psid), the part of the string typed correctly so far (i.e., ptyping). A string tuple has attributes like x and y coordinate values, characters, and a status which determines on which game fields the string should be displayed. A bullet has x and y coordinate values, and the ID of string being shot. A color has 3 integers to represent its RGB value.

We model different read and write requests on these data as read and write transactions, as described in Table 3. We allocate different transactions to different threads identified in Table 1. The background thread will update x and y coordinate values of strings and bullets in a write transaction of type (1). Once a string moves out of game field (i.e., y coordinate value of the string is larger than the height of game field), a write transaction of type (2) will be executed to update the status of the string to be NOT-DISPLAYED. A write transaction of type (2) will be also triggered once a bullet shoots a string (i.e., y coordinate value of bullet is smaller than that of its target string). At the same time, the bullet will be also deleted from the database (i.e., write transaction (3)). The reading thread of each player will execute at certain time intervals a read-only transaction which reads all game state data containing information about strings, bullets, and scores etc. We model typing a string as a transaction of type (5) and typing a character as an operation (i.e., appending the character to the *ptyping* attribute of the player). It is executed by the updating thread of each player. Once a string is successfully typed, a transaction of type (6) is triggered to increase the score of the player, so will a transaction of type (7) to create a new bullet.

Thread	transaction
background thread	(1) update strings and bullet positions (2) update status of string to be displayed or not (3) delete a bullet once it shoots a string.
reading thread	(4) read strings, bullets, their positions, their colors, all scores;
updating thread	(5) update the ptyping attribute until a string is complelely typed (6) increse one player's score; (7) create a bullet;

Table 3: Definitions of different transactions

Note that most MOGs only use a database for storing data related to the client's performance information such as password and credit. They manage the game state information in main memory instead of the database. In our work, we take advantage of databases in different ways. We not only store static data related to clients but also dynamic data related to game states. We also take advantage of the concurrency control mechanisms provided by the database to control and serialize the execution of the game application.

4.5 Lessons learned from the design

In Section 3, we have looked in detail at different replication protocols. We must be aware that some replication protocols have certain limitations. These limitations might lead to different implementations or even different game semantics. For example, a lazy primary approach requires each transaction to be marked as read-only or not. Thus, we have to call JDBC setReadOnly() in the implementation of the game using the lazy primary copy approach. Symmetric requires all operations to be known at begin of transaction. Thus, in MiddleSIR, Symmetric requires an application to submit all SQL statements in one statement. As a consequence for the typing game, the player has to first type the full word, then press enter, and only then all letters are sent in a single SQL statement to MiddleSIR. In contrast, for the other protocols, each character typed will be sent immediately to MiddleSIR, Thus, type checking can occur immediately. In summary, the lazy primary copy approach and Symmetric put some restrictions on the game implementation, while SRCA-REP and SEQ did not have any restrictions.

By taking advantage of transaction properties, we do not need to be concerned with concurrency issues when designing the game. However, we need to carefully design our database tables in a reasonable way. In fact, we might

Table	primary key	attr1	attr2	attr3	attr4	attr5
player	pid	pname	pscore	pcid(refer cid)	psid(refer sid)	ptyping
string	sid	sx	sy	stxt	sstatus	
bullet	bid	bx	by	bsid(refer sid)		
color	cid	cr	cg	cb		

Table 2: Database schema for the typing game

consider different table schemes depending on which replication protocol is used. For example, two transactions of type (1) and type (5) do not conflict and should be able to run concurrently. The transaction of type (1) changes a string's y coordinate value. The other appends a character to *ptyping*. According to the schema shown in Table 2, these two transactions do not have write/write conflicts. However, if our database schema defined *ptyping* as an attribute of the string table, the two transactions would have write/write conflicts. Recall that the lazy primary copy approach, SRCA-REP, and SEQ will commit only one of several concurrent transactions with write/write conflicts. Thus, either the transaction of type (1) or type (5) would be aborted. If the type (1) transaction, which updates the positions of strings, aborted, the player would see as effect that strings do not move. If the type (5) transaction, which updates *ptyping*, aborted, the color of a fully typed string would not change. In Symmetric, concurrent conflicting transactions do not lead to an abort but they are executed serially. Thus, *ptyping* and the string position can be in the same table without possibility of abort. Thus, one can see how different replication protocols might lead to different game experience.

Clearly, one also has to be aware of the different fault-tolerance properties provided by the different protocols and as discussed in Section 3.5.

5. EXPERIMENTS

We now show experiments that show how well MiddleSIR provides scalability and fast response time to the typing game. [9] has compared different replication protocols and concluded that SEQ has the best response time in both LAN and WAN. Hence, we only show the results for SEQ.

5.1 Experiment setup

We conduct experiments in both LAN and WAN. Each machine runs a MiddleSIR replica (including middleware replica and database replica). We just refer to these machines as servers. In the LAN setup, all servers are located in one LAN. We use PCs with Pentium(R)4 2.6GHz cpu, 512MB memory, Linux 2.6.16-gentoo-r6, and 100Mbps network connection. In the WAN setup, we use Emulab [2] to simulate the network topology, in which the round trip time between any two servers is 50 to 100 milliseconds and the bandwidth is 5Mbps. The power of PCs is Pentium4 3.0GHz CPU, 1GB memory. The operation system is Linux 2.4.20-31.9.

Since we cannot manually run many game clients at the same time, we implemented a simulator to simulate the work load of game clients. The simulator starts a number of virtual clients. The difference between virtual clients and real clients is that virtual clients do not perform any graphics processing tasks. Each virtual client performs a read-only transaction each 1000 ms to the read game state (i.e., 30 strings, bullets, their positions and colors, and player scores). Each client also submits write transactions every 2000 ms. A write transaction is composed of 3 random characters on average. The write transaction simulates that a player types a string randomly. Note that in all experiments the virtual clients are connected to their local server and each server has the same number of virtual players.

It is widely accepted that response time of player's action is very important in experiencing MOGs. Hence, we measure the performance in terms of the response time of transactions at the game clients (i.e., how fast a player can read game state from the server, and how fast the effects of typing a word are received by the client). We do not take into account the time spent in displaying the strings, bullets and scores, because it depends on the power of the players' machines.

5.2 Experiments in LAN

Scalability

Figure 8 shows how the system scales from 1 server to 6 servers in a LAN. The system scales quite well from 1 sever to 4 servers. However, it scales badly when there are more than 5 servers. This is due to the ROWA property of SEQ. Recall that in a ROWA protocol a read request only needs to be executed in one site while a write request must be executed in all sites. Hence, as long as there is an increasing number of reads but a relatively stable number of writes in the system, adding new servers will help handling the increasing load. However, since writes have to be executed at all replicas, adding more servers will not enable the system to handle more write requests. In our game environment, each player has both read and write transactions, thus, scalability is limited. If the game had more read tasks per player, the system could handle more players by adding more servers.

Response time in a system with 4 servers

We measured the average response time of read and write transactions of players in a system with 4 servers. We separate the results for SEQ sequencer and SEQ non-sequencers. It is because SEQ sequencer has more validation overhead than SEQ non-sequencers.

Figure 9 and 10 show the average response time of read and write transactions respectively. Both read and write transactions have fast response time (i.e., 10 to 20 ms for read and 10 to 40 ms for write). Note that write transactions submitted to the sequencer are slightly faster than those submitted to non-sequencers. It is because the write transactions submitted to non-sequencers need to communicate with the sequencer for validation. Thus, one round trip time between a non-sequencer and the sequencer is included in the response time.

Figure 11 shows the CPU usage in the server. The sequencer has slightly higher CPU usage than non-sequencers.

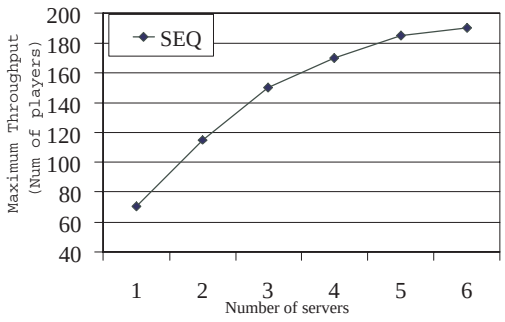


Figure 8: scalability in LAN

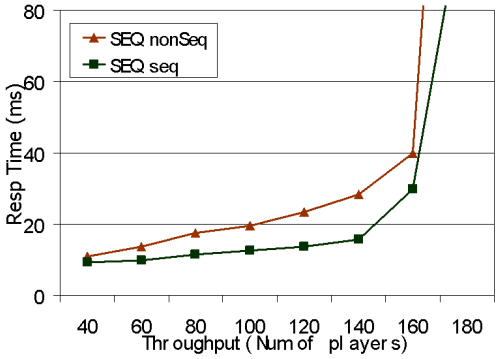


Figure 10: write transactions, 4 servers in LAN

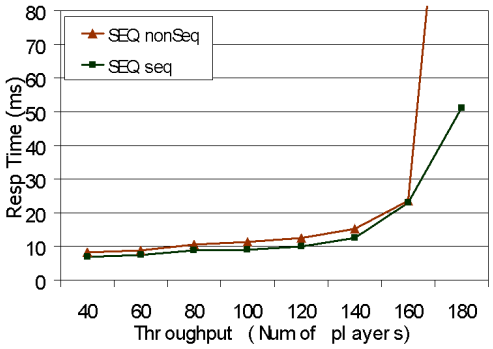


Figure 9: read transactions, 4 servers in LAN

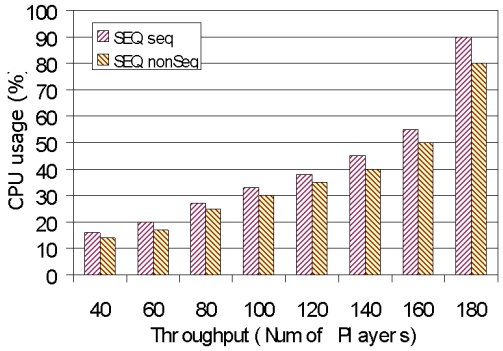


Figure 11: cpu usage, 4 servers in LAN

It is because the sequencer needs to perform validation for all write transactions, while non-sequencers do not. The figure shows that the validation overhead is not significant. The system saturates at the throughput of 180 players.

5.3 Experiments in WAN

Scalability

Figure 12 shows the scalability of the system with SEQ. As in LAN, the system scales well with less than 5 servers. With more than 5 servers, the system does not scale.

Response time in a system with 4 servers

Figure 13 shows the response time of read transactions in a system with 4 servers in the WAN. We can see that the response time of a read transaction is as fast as in LAN experiments. This is because in SEQ read transactions are executed in their local servers without any synchronization with other remote servers. As long as the server is near the client (e.g., in the same LAN), the response time will be fast.

Figure 14 shows the response time of write transactions in a system with 4 servers in the WAN. The SEQ sequencer responds to its clients very fast. It is because the response times of write transactions submitted to the sequencer do not include any WAN communications. The scheduling is done locally in the sequencer. However, for clients of non-sequencer servers, the response times of write transactions are between 60 and 80 ms. This is because the response time includes one round trip WAN communication. The response time difference in LAN was negligible because of the small network latency in LAN. In a WAN environment, however, the large network latency has a large impact.

5.4 Discussion

The experiments above show that the game has fast response time in both LAN and WAN environments. Read-only transactions are always very fast (i.e., no more than 20 ms). This is true because read-only transactions are always executed locally without network communication. For write transactions, the response times of different protocols vary. Some research [15, 12] investigated the acceptable response time of interactions in MOGs. They concluded that 100 to 300 ms will be a tolerable bound for MOGs players to continue playing. According to our experimental results, SEQ produces response times less than 100 ms for the typing game, even in a WAN with 50 to 100 ms round trip time delay between servers.

One observation is that database replication can only provide a certain degree of scalability to the system. This is because writes must be applied at all sites. If the game had only write requests, the system would not be scalable. The typing game has a considerably higher write than read load. Hence, database replication does not provide very good scalability. In future work, we are planning to analyze what the typical ratio of reads and writes is in common MOGs to better understand the potential for scalability.

Another solution to improve the scalability is using partial replication [18]. Partial replication is different from ROWA in that each replica has only part of the data. Thus not all

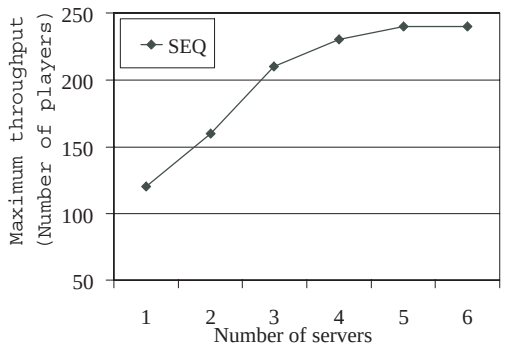


Figure 12: scalability in WAN

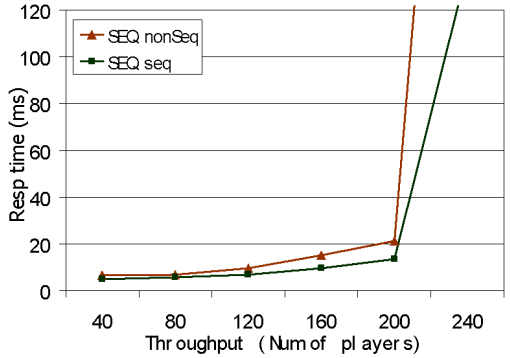


Figure 13: read transactions, 4 sites in WAN

write requests need to be applied at all sites. However, this will introduce complexity for read requests since some sites might not have the data to be read. In regard to research in database replication, partial replication is still a challenging and open topic. However, we believe it is possible to design a practical partial replication protocol for MOGs based on the characteristics of MOGs. Several game research proposals suggest to divide a MOG into different game zones whose data are mainly disjoint from each other except that players move from one zone to others [6, 1]. Zone partitioning might be an interesting starting point to find a good partial replication protocol. We are particularly interested in extending our work in this direction.

The current implementation of the typing game is actually in a pulling mode. All players read different data actively from the database. This is true because they subscribe to different subsets of data. But these data overlap a lot. We can reduce the read load at the database by implementing the typing game in a pushing mode. For example, we can implement a game server which reads all data on behalf of all clients and sends appropriate data to each individual client. This will reduce the overhead at the database. Note that the system can still be scalable. If the bandwidth of a game server is exhausted while the load of the database is still low, we can add more game servers to handle more clients. Adding one game server to the replicated database will have much less load than adding several clients. The trade-off is that we need to implement a game server application properly according to the game semantics. We expect that the typing game system will scale better in pushing mode than

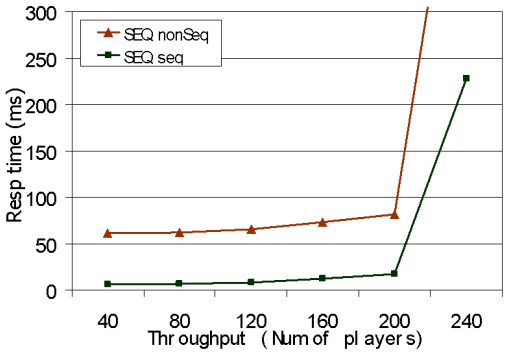


Figure 14: write transactions, 4 sites in WAN

in pulling mode.

Note that by using database replication, the game can also achieve transparent fault tolerance and data consistency, as have been discussed in Section 3.

6. RELATED WORK

[4] proposed to replicate states in MOG. But it does not use database replication. [10] proposed to use transactions in MOG to enhance parallel execution. It considers simple and complex transactions. A complex transaction will contain several simple atomic transactions. However, it is unclear in [10] how to apply transactions in the real design.

Zone partitioning is used in many MOGs studies such as [6, 1]. [6] studies MOGs in a peer-to-peer system. It divides the system into regions. The game states of all players in a region are kept consistent by multicast within the region. If a player moves from one zone to another, its personal data must be migrated. [1] studies the load balancing problem when different servers handle different zones.

[3] proposed the categorization of database replication into lazy/eager and primary copy/update. [19] did extensive analysis and experiments on update everywhere protocols based on multicast. [5, 11] have shown that a replicated database system is able to scale in LAN. [7] has shown that database replication can provide fast response time even in WANs.

[16] introduces *local perception filter* to compromise the inconsistent visual effect different players experience due to network delays. The method is extended by [17] to realize the bullet time effect which was firstly introduced in the file *The Matrix*. The problem of inconsistent visual effects seen by different players is also avoided in our approach. Our approach is different from local perception filter. It uses database replication instead of adjusting each user's perception of the environment depending on network delay.

7. CONCLUSION

Recent advances in database replication have shown that database replication can provide good fault tolerance, scalability, and fast response time while preserving data consistency. These are exactly the features that are needed in MOGs. However, we are not aware of any study that analyzes whether and how database replication can be applied to MOGs. In this paper, we describe how to do so by implementing a small multi-player typing game on top

of MiddleSIR, a framework which provides database replication support. We store the game state in a relational database. Different player actions are modelled as either read-only or write transactions on the game state. We take advantage of transaction semantics to simplify the concurrency control overhead in designing the game. Moreover, the replicated database system guarantees data consistency across all replicas so that all players read the same game state even if connected to different servers.

We also discuss different replication protocols implemented in MiddleSIR and analyze how these protocols may influence the game semantics and the design of our typing game. We test the typing game in LAN and WAN environments using one of the protocols. The results show that database replication can provide good response time to the typing game. For scalability, the system scales well up to 5 sites. We also discuss how scalability can be achieved beyond 5 sites by using various techniques.

8. REFERENCES

[1] Jin Chen, Baohua Wu, Margaret Delap, Bjorn Knutsson, Honghui Lu, and Cristiana Amza. Locality Aware Dynamic Load Management for Massive Multiplayer Games. In *PPoPP*, 2005.

[2] Emulab. http://www.emulab.net.

[3] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD'96*.

[4] Carsten Griwodz. State replication for multiplayer games. In *Netgames*, Portland, Oregon, USA, August 2004.

[5] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving Scalability of Fault Tolerant Database Clusters. In *ICDCS'02*.

[6] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massy multiplayer games. In *INFOCOM*, Hong Kong, China, 2004.

[7] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Consistent Data Replication: Is it feasible in WANs? In *Euro-Par*, Sep 2005.

[8] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Middleware based data replication providing snapshot isolation. In *SIGMOD*, June 2005.

[9] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Practical Database Replication in WANS. In *Unpublished manuscript*, Nov 2005.

[10] Ian Lintault. A transaction execution engine architecture for multiplayer online games. In *Netgames*, Portland, Oregon, USA, August 2004.

[11] M. Patiño-Martínez and R. Jiménez-Peris and B. Kemme and G. Alonso. Consistent database replication at the middleware level. In *ACM TOCS*, volume vol.23(4), 2005.

[12] I. S. MacKenzie and C. Ware. Lag as a determinant of human performance in interactive systems. In *Proceedings of the CHI'93 Conference on Human factors in computing systems*, New York, NY, 1996. IEEE Computer Society Press.

[13] News. http://pc.gamespy.com/pc/world-of-warcraft/582134p1.html.

[14] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, 2004.

[15] S.Cheshire. Latency and the quest for interactivity, 1996.

[16] P.M. Sharkey, M.D. Ryan, and D.J. Roberts. A local perception filter for distributed virtual environments. In *Virtual Reality Annual International Symposium*, 1998.

[17] J. Smed, H. Niinisalo, and H. Hakonen. Realizing bullet time effect in multiplayer games with local perception filters. *Computer Networks*, 49(1).

[18] A. Sousa, R. Oliveira, F. Moura, and F. Pedone. Partial replication in the database state machine. In *IEEE International Symposium on Network Computing and Applications*, 2001.

[19] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE transactions on knowledge and data engrineering*, 17(4):551–566, 2005.