2010 16th International Conference on Parallel and Distributed Systems

Multi-Consistency Data Replication

Raihan Al-Ekram and Ric Holt
David R. Cheriton School of Computer Science
University of Waterloo, Canada
{aralekra, holt}@uwaterloo.ca

***Abstract*—Replication is a technique widely used in parallel and distributed systems to provide qualities such as performance, scalability, reliability and availability to their clients. These qualities comprise the non-functional requirements of the system. But the functional requirement consistency may also get affected as a side-effect of replication. Different replica control protocols provide different levels of consistency from the system. In this paper we present the middleware based McRep replication protocol that supports multiple consistency model in a distributed system with replicated data. Both correctness criteria and divergence aspects of a consistency model can be specified in the McRep configuration. Supported correctness criteria include linearizability, sequential consistency, serializability, snapshot isolation and causal consistency. Bounds on divergence can be specified in either version metric or delay metric. Our approach allows the same middleware to be used for applications requiring different consistency guarantees, eliminating the need for mastering a new replication middleware or framework for every application. We carried out experiments to compare the performance of various consistency requirements in terms of response time, concurrency conflict and bandwidth overhead. We demonstrate that in McRep workloads only pay for the consistency guarantees they actually need.**

Keywords- replication; consistency; corectness criteria; divergence; linearizability; sequencial consistency; serializability; snapshot isolation; causal consistency; session guarantee.

I. INTRODUCTION

Replication is a parallel and distributed computing technique widely used to achieve high performance, scalability, fault-tolerance and high availability to computer systems. These qualities comprise the non-functional requirements of the system. But the functional requirement consistency may also get affected as a side-effect of replication. Different replication protocols provide different levels of consistency in the system. Also different application requires different levels of consistency from the system. When developing a replicated system, the current practice is either to use different replication techniques for different applications according to their consistency requirements or to use a replication mechanism with a consistency guarantee strong enough for all the applications. The first case requires mastering a new protocol and framework each time implementing a new replicated application with new consistency requirement. Whereas, in the second case an application may have to pay for the higher overheads of a consistency guarantee stronger than required.

This paper presents McRep (**M**ulti-**c**onsistency **Rep**lication), a middleware based replication protocol that supports multiple consistency models in a replicated data system. The McRep consistency model allows the system to configure both the correctness criteria and the divergence aspects of consistency. Supported correctness criteria include linearizability, sequential consistency, serializability, snapshot isolation and causal consistency. Bounds on divergence can be specified in either version metric or delay metric.

We carried out experiments to compare the performance of the McRep for different consistency settings. We compared the response time for requests, number of concurrency conflicts, number of update propagation messages and overhead network traffic for different correctness criteria and different values for divergence bound. The results of the experiments confirm that in McRep stronger consistency guarantee costs higher performance penalty and workloads only pay for the consistency guarantees they actually need.

II. SYSTEM MODEL

For the purpose of this paper we consider a *computer system* to be comprised of an *application* and a set of *data* items, see Figure 1. The application provides services to the *clients* of the computer system and the data gives the system its state. The clients send service *requests* to the system and expect a *response* for each request. For each request the application initiates a *transaction* and computes the response. A transaction is a group of operations to be performed on the data as a unit. Transactions are atomic, either all of the operations in a transaction succeed or all of them fails. There can be two types of transactions: *read-only transactions* and *update transactions*. Read-only transactions read the values from one or more data items, perform some computations on the values and produce one or more results to be sent to the clients as responses. Read-only transactions do not modify any data item, leaving the system state unchanged. Update transactions may read the value of data items, perform some computations and then update (i.e. add, delete and/or modify) one or more data items while producing the results. Each successful update transaction produces a new *version* of the system state. The system versions are numbered and each successful update increments the version number by one. The set of data items a transaction reads comprises the *readset* of the transaction, while the set of data items added, deleted and modified comprises its *writeset*. For read-only transactions the write-set is always empty. A pair of transactions is *non-concurrent* if one of the transactions runs into completion before the other transaction begins. The transactions are *concurrent* if one of them begins before the completion of the other. A client request that requires a read-only transaction to be performed is a *read request*, whereas a client request requiring an update transaction is an *update request*.

A *replicated computer system* consists of a set of copies of the original system, each called a *replica*, connected to each other with a communication *network*. The replicas concurrently

1521-9097/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPADS.2010.67

process client requests. Concurrent update transactions in the replicas can cause the replica states to diverge and become inconsistent with each other. A replication control *protocol* runs in the replicas to co-ordinate among themselves. The protocol performs periodic or on-demand *update propagations* among the replicas in order to keep them in sync and consistent with each other. Each replica has its own *replica version* given by the version of the last update propagation received and applied in it. At any particular time a replica version is either up-to-date or lagging behind the *system version,* which would be the correct version of the system if there was no replication.

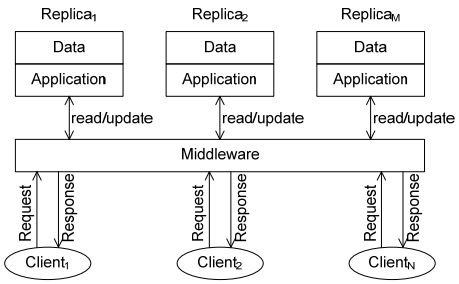

Figure 1. Middleware Based Replication Model

A *middleware based replicated system* has a middleware component that lies in between the clients and the replicas. The middleware intercepts client requests, routes them to and from the replicas and co-ordinates the operations in the replicas. The middleware hides the complexity of replication and provides an illusion of a single service to the clients

III. CONSISTENCY MODEL

In a replicated system, each replica contains a copy of the data that is concurrently being read and updated by transactions to serve the client requests. Hence the notion of consistency is central to replicated systems. There are two aspects of consistency of the data among the replicas to consider: (1) correctness criteria and (2) divergence bound.

(1) *Correctness Criteria* specify requirements on legal sequence of transaction execution that will leave the replica states (i.e. the data in the replicas) equivalent to each other according to some criteria in presence of concurrent access [20].

(2) *Divergence Bound* specifies how far the replica states may diverge from the correct system state at any time [4] before they are brought up-to-date by propagating updates.

A. Correctness Criteria

We classify the various correctness criteria of replicated data into following three classes.

Correctness Class: Total

According to the most popular class of correctness criteria, the execution of a set of transactions and their corresponding update propagations in the replicas is correct if it is equivalent to execution of those transactions in some sequential order on the same system without replication. Every replica observes the same total ordering on the transaction (hence the name *Total* correctness class), but different specific correctness criteria impose different total ordering on the transactions. Enforcing any specific correctness criteria from this class leaves the data in replicas exactly in the same state after the execution of the transactions. But the resulting state might be different for different specific correctness criteria. Following are some specific correctness criteria of this class of replicated data consistency proposed in the literature:

- *Linearizability* [16][27], also known as *Strict Serializability* [17], imposes a total ordering on the transactions among the replicas even if the transactions access disjoint data items and would not otherwise need to be ordered. In linearizability, concurrent transactions can be executed in any total order between them but the real-time ordering for non-concurrent transactions should be preserved. All transactions observe the most recent system version and the latest value of the data items.
- *1SR* or *One-Copy Serializability* [7], *Serializability* [22] for multi-versioned replicated data, allows any total ordering on the transactions among the replicas. No real time ordering is necessary. Transactions may read stale data by getting ordered on a staler system version than the ones created by recent update transactions. Clients may observe inconsistent responses from their consecutive requests due to the non-real-time ordering of the corresponding transactions.
- *Sequential Consistency* [19], also known as *Strong Session Serializability* [11], requires a total ordering on the transactions among the replicas with a real-time ordering only on the non-concurrent transactions from the same client session. With sequential consistency transactions from the same client always observe a linearizable view of the system, whereas transactions from different clients observe a serializable view. The clients observe a consistent view of the system between consecutive requests but transactions may still read stale data.
- *Generalized Snapshot Isolation* [14] or *One-Copy Snapshot Isolation* [21], *Snapshot Isolation* [6] for replicated data, imposes a total ordering only on the write operations in the update transactions among the replicas. The read operations in the transactions may get arbitrarily ordered on a staler version (or snapshot) of the system. Since there is no total ordering among the whole transactions, concurrent transactions may conflict with each other. In order to avoid write-write conflicts, concurrent transactions are not allowed to update any common data item. Snapshot isolation may still suffer from a concurrency anomaly that violates constraints over multiple data items known as *write skew* [6], caused by read-write conflicts. Also like serializability transactions may read stale data and clients may observe inconsistent responses.
- *Strong Session Snapshot Isolation* [12] constrains a real-time ordering on the non-concurrent transactions (as a whole, not just read or write operations) from the same client session on top of snapshot isolation. This eliminates the inconsistency from clients' perspective.

The difference among the various correctness criteria of this class can be explained with the transaction execution scenario illustrated in Figure 2. In this scenario the replicated system needs to process four transactions originated from the requests of two clients. Transactions *T1* and *T2* are originated

from *Client 1*, while transactions *T3* and *T4* are from *Client 2*. *T1* and *T4* are read-only transactions, while *T2* and *T3* are update transactions. *T1* reads data items *a* and *d*. *T2* reads data items *a*, *c* and *d* and updates data item *b*. *T3* reads *a*, *b* and *c* and updates *a*. T4 reads *a* and *b*. Transactions *T2* and *T3* are concurrent. *T1* precedes them in real-time, while *T4* follows.

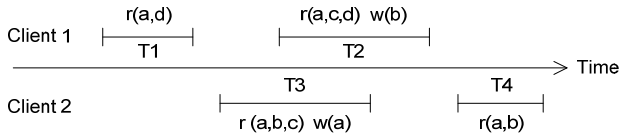

Figure 2. Transaction Execution Scenario 1

- Valid execution sequence of the transactions in this scenario in order to fulfill linearizability should be (*T1*, *T2*, *T3*, *T4*) or (*T1*, *T3*, *T2*, *T4*). Concurrent transactions *T2* and *T3* can have any ordering but the real time ordering of *T1* and *T4* are preserved.
- Execution sequence (*T1*, *T3*, *T4*, *T2*) is not linearizable since the real time ordering of non-concurrent transactions *T2* and *T4* are not preserved. But it fulfills sequential consistency because the real time ordering between transactions *T1*, *T2* and *T3*, *T4* are maintained individually. Execution sequence (*T3*, *T1*, *T2*, *T4*) can be another sequentially consistent ordering.
- Execution sequence (*T1*, *T4*, *T3*, *T2*) or (*T1*, *T4*, *T2*, *T3*) are one-copy serializable but does not obey sequential consistency, because the real time ordering between transactions *T3* and *T4* are not preserved.
- (*T1*, *T2-r*, *T3-r*, *T2-w*, *T3-w*, *T4*) is not a one-copy serializable ordering. The read and write operations of transaction *T3* are performed on two different system versions, transaction *T2* changes the system version in between them. But the ordering obeys strong session snapshot isolation since the write operations in the transactions are ordered and the real time ordering of the read and write operations between *T1*, *T2* and *T3*, *T4* are maintained individually. (*T1*, *T2-r*, *T3-r*, *T3-w*, *T4*, *T2-w*) is another ordering that obeys strong session snapshot isolation.
- (*T1*, *T2-r*, *T3-r*, *T4*, *T3-w*, *T2-w*) is an execution sequence for generalized snapshot isolation. It does not obey strong session snapshot isolation since the real time ordering between *T3* and *T4* are not maintained.

Correctness Class: Causal

The next class of correctness criteria *Causal Consistency* [2] permits each replica to have its own sequential view of the execution as long as the transactions preserve the causal ordering relationship in the replicas. A pair of transactions is causally related if there is a *client-order* dependency or a *read-from* dependency between them [26]. Client-order dependency, also known as *Session Guarantee* [29], requires a real-time ordering between non-concurrent transactions from the same client. Read-from dependency gives the ordering between transactions where one transaction reads a data item written by the other before it gets overwritten. With causal consistency a set of transactions and their corresponding update propagation can be executed in different orders in different replicas, resulting the replica states to be different from each other.

Correctness criteria of this class can be explained with the scenario illustrated in Figure 3. *Client 1* submits transactions *T1* and *T2* to the system and *Client 2* submits *T3*, *T4* and *T5*. The transactions from *Client 1* are served by *Replica 1*, while *Replica 2* serves the transactions from *Client 2*.

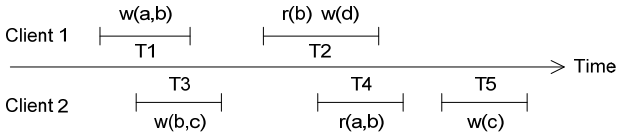

Figure 3. Transaction Execution Scenario 2

An execution sequence of (*T1*, *T3*, *T2*, *T5*) in *Replica 1* and (*T3*, *T1*, *T4*, *T5*, *T2*) in *Replica 2* do not fulfill any correctness criteria from the *Total* correctness class, since the sequential views of the transactions in the replicas are not the same. But they follow the causal consistency criteria because the order of *T1* and *T2* in *Replica 1* and *T3*, *T4* and *T5* in *Replica 2* is maintained.

Correctness Class: Eventual

The final class of correctness criteria does not guarantee equivalence among the replica states at all times. *Eventual Consistency* [30] allows the replicas to arbitrarily diverge from each other. The correctness guarantee is that if there is no new update request coming in the system, eventually all update transactions will propagate to all the replicas and after resolving any conflicts the replicas will eventually converge to equivalent states. The equivalence can be either a total or a causal ordering of the write operations of all the update transactions [32]. With eventual consistency, an apparently successful update transaction may get aborted after resolving all its conflicts. Also a transaction may read data from another transaction whose conflicts have not yet been resolved and may get aborted.

Refer to the scenario in Figure 3 in order to explain eventual consistency. Initially *Replica 1* executes the transaction sequence (*T1*, *T2*) and *Replica 2* executes (*T3*, *T4*, *T5*). Transaction sequence (*T1*, *T2*) in *Replica 1* and (*T3*, *T4*, *T5*) in *Replica 2* are concurrent since they are executed concurrently and independently in the replicas without enforcing any correctness criteria. Eventually *Replica 1* receives (*T3-w*, *T4-w*) from *Replica 2*. *Replica 1* determines a write-write conflict between *T1* and *T3-w* and decides to abort *T3-w*. The final state of *Replica 1* is given by the write sequence (*T1-w*, *T2-w*, *T5-w*). Similarly *Replica 2* eventually receives (*T1-w*, *T2-w*) from *Replica 1*. *Replica 2* also must decide to abort *T3* after detecting the conflict. The final state of *Replica 2* is given by (*T5-w*, *T1-w*, *T2-w*), which is same as the final state of *Replica 1* since they perform the same set of modifications on the data.

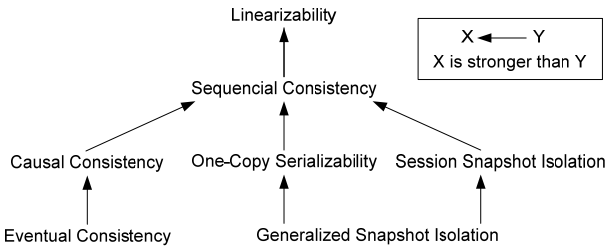

Figure 4. Relationship Among the Various Correctness Criteria

Figure 4 shows a partial order relationship among the various correctness criteria ranked according to their strength. A correctness criteria *X* is stronger than another criteria *Y*, if *X* is more restrictive than *Y*. There is also a compositional relationship among some of the criteria.

- Sequential Consistency = One-Copy Serializability + Session Guarantee
- Session Snapshot Isolation = Generalized Snapshot Isolation + Session Guarantee.

There are other related but less commonly used correctness criteria that are not discussed in this paper. Z-linearizability [27], epsilon serializability [25], quasi serializability [13] and semantic consistency [15] are some of them.

B. Divergence Bound

According to the replication model used in this paper, a replica is either up-to-date with the current system state or lagging behind it. Divergence is the measure of how far behind a replica is from the current system state. The divergence of a replica at any particular time can be quantified by three metrics: (1) version metric, (2) delay metric and (3) arithmetic metric [4][31].

- The *version metric* gives how many versions or update transactions the replica is behind from the most recent system version.
- The *delay metric* gives the time difference in real clock time between the creation time of the system version that corresponds to the current replica version and the creation time of the most recent system version.
- If the data items are numbers, the *arithmetic metric* gives the numeric difference between the value of a data item in the replica and the value of the same data item from the most recent system version.

To illustrate the three divergence metrics, consider at some particular time *t0+t1* all the replicas of the system are up-to-date with the most recent system version *V*. *V* was created by an update transaction at time *t0* but was propagated and applied in a particular replica *R* at time *t0+t1*. At this time the value of a particular numeric data item *D* in the replicas is *d1*. Within next *t2+t3* units of time the system executes *n* update transactions, the last one being executed at time *t0+t1+t2*. The system version at *t0+t1+t2+t3* is hence *V+n*, the creation time of this version being *t0+t1+t2*. Some of these transactions write *D* changing its value to *d2*. Consider replica *R* did not receive any update propagation during the time *t0+t1* to *t0+t1+t2+t3*, having its replica version still at *V*. At time *t0+t1+t2+t3*, the divergence of *R* is:

- In version metric, $(V+n) - V = n$
- In delay metric, $(t0+t1+t2) - t0 = t1+t2$
- In numeric metric for data item *D*, $|d1-d2|$

The divergence of a replica can be bounded by bringing it up-to-date with the latest system state by sending the pending updates to it before it crosses the allowed bound.

Divergence is an issue orthogonal to correctness criteria, except for linearizability. It is possible to provide most of the correctness criteria with replicas diverging from each other. But linearizability requires a replica to be up-to-date before executing any transaction.

- Linearizability = One-copy Serializability + Zero Divergence Bound

IV. Implementing Correctness

Suppose a transaction *T* is initiated by a client request received in the system at time *request*(*T*). *T* is being executed in a replica with a replica version *V*, which can be older than the most recent system version at *request*(*T*). The effective starting time *start*(*T*) of transaction *T* will be right after the creation time *create*(*V*) of the system version *V*. Thus *start*(*T*) can be much earlier than *request*(*T*). But the completion time *end*(*T*) of the transaction would be the same as *response*(*T*), the time the response for the request is sent back to the client. If *T* is an update transaction, upon successful completion it will create a new system version *V+n*+1 at *end*(*T*), assuming *V+n* was the latest system version just before the completion of *T*. Note that the latest system version *V+n* just before *end*(*T*) and hence *response*(*T*) might be higher than the latest system version at *request*(*T*), caused by concurrent update transactions completed between *request*(*T*) and *response*(*T*). The relationship among the various timelines of the transaction *T* is:

(i) $create(V) < start(T) < create(V+1)$
(ii) $create(V+n) < end(T) = create(V+n+1)$
(iii) $start(T) \leq request(T) < response(T) = end(T)$

When determining concurrency of transactions, this definition of *start*(*T*) and *end*(*T*) should be considered.

A. Generalized Snapshot Isolation

Implementation of generalized snapshot isolation is described in [14] and [21]. A read-only transaction *Ti* can be executed at any replica with a replica version is *V*. Ordering of *Ti* in the transaction sequence will be at *start*(*Ti*), right after the transaction that created the system version *V*. An update transaction *Ti* can also begin at any replica. The read operations of *Ti* are then ordered at *start*(*Ti*). The write operations create a new system version at *end*(*Ti*) and are ordered here; after the existing completed transactions up to *end*(*Ti*). But in order to prevent the write-write conflicts if two concurrent update transactions write on a common data item, one of them must be aborted. Transactions *Ti* has a write-write conflict with transaction *Tj* if:

(i) $start(Ti) < end(Tj) < end(Ti)$
(ii) $writeset(Ti) \cap writeset(Tj) \neq \emptyset$

The first inequality tests the concurrency of *Ti* and *Tj* and the second inequality tests for conflict. In case of a conflict *Tj*, the transaction completed first, wins and *Ti* must be aborted. If there is no conflict, concurrent transactions are ordered by their completion time. The inequalities are tested at the end of each update transaction to determine if it can complete successfully and create a new system version or it must be aborted.

B. One-Copy Serializability

One-copy serializability can be achieved on a system providing generalized snapshot isolation with an additional condition to prevent the read-write conflicts between concurrent update transactions [14][23]. *Ti* has a read-write conflict with *Tj* if:

(i) $start(Ti) < end(Tj) < end(Ti)$
(ii) $writeset(Ti) \cap readset(Tj) \neq \emptyset$

In case of a conflict *Tj*, the transaction to end first, wins and *Ti* is aborted. A read-only transaction *Ti* executed at a replica with replica version *V* is ordered at *start*(*Ti*). The inequalities are tested at the end of each update transaction. A non-concurrent or a concurrent but non-conflicting or a conflicting but winning update transaction *Ti* creates a new system version at *end*(*Ti*) and is ordered here.

C. Sequential Consistency and Session Snapshot Isolation

Sequential consistency can be obtained on a replicated system providing one-copy serializability by ensuring transactions from a client are executed on replicas with replica versions no less than the highest system version observed by the completed transactions from the same client so far. The system version observed by a transaction *T* is the system version where *T* is ordered. If *T* is read-only it is the replica version *V* of the replica executing *T* at *start*(*T*). If *T* is an update transaction it is the system version *V*+*n*+1 created by *T* upon successful completion at *end*(*T*).

Suppose at *request*(*Ti*) of a transaction *Ti* from a client *C* the highest system version observed by the completed transactions from *C* is *U*. Executing *Ti* in a replica with a replica version at least *U* guarantees that *create*(*U*) < *start*(*Ti*) and thus *Ti* will be ordered after the transaction that created the system version *U*.

The same replica selection strategy can be used to achieve session snapshot isolation from generalized snapshot isolation.

D. Linearizability

Linearizability can be implemented on a system providing one-copy serializability by selecting a replica with zero divergence, i.e. it is up-to-date with the most recent system version, for the execution of a transaction.

Suppose at *request*(*Ti*) of a transaction *Ti* the latest system version is *U*. *Ti* must be executed in a replica with a replica version equal to at least *U*. To make sure for every transaction there are up-to-date replicas available, whenever a new system version is created it should be propagated to all the replicas.

E. Causal Consistency

To guarantee casual consistency, a transaction *Ti* can be executed independently in any replica *R* as long as *R* has observed all transactions with client-order and read-from dependency with *Ti* [9][26]. Transaction *Ti* is client-order dependent on transaction *Tj*, if *Tj* is a previously completed transaction from the same client that originated *Ti*. And transaction *Tj* is read-from dependent on transaction *Tk*, if *Tj* reads a data item written by *Tk*. At *request*(*Ti*) an *R* can be chosen for the transaction *Ti* such that *R* has observed the immediate *Tj* for *Ti* and all *Tk* for *Tj*.

V. The McRep Protocol

This section presents the McRep protocol that can provide a range consistency guarantee to a replicated computer system. The choices for correctness criteria are linearizability, sequential consistency, one-copy serializability, session snapshot isolation, generalized snapshot isolation and causal consistency. The divergence bound can be specified in version and/or delay metric.

The basic structure of McRep protocol is based on the middleware based snapshot isolation protocol SRCA by Lin et al [21]. In McRep a centralized middleware is assumed to be connected to the clients and the replicas with a network that provides reliable and in order delivery of messages. McRep uses an asynchronous transaction processing mechanism for achieving high performance and scalability.

A. Basic Operation

The middleware intercepts each client request and selects a replica to process the request. A read request is sent to the selected replica for performing a read-only transaction. After executing the transaction the replica sends the result to the middleware, which in turn sends a response back to the client.

An update request can be processed in the replicas in one of two modes: (1) the *update-apply* mode and (2) the *simulate-apply* mode, by combining an *update* or a *simulate* transaction with many *apply* transactions. During the update transaction the application in a replica reads one or more data items, perform some computations and then updates (i.e. add, delete and/or modify) one or more data items required by the transaction. It also computes a *writeset* for the transaction, which contains a list of data items being updated and their modified values. During the simulate transaction the application performs all the data reading and computation required by the transaction but does not actually modify any replica data. Instead it computes the *writeset* and optionally a *readset*. The readset contains only the list of data items that are read by the transaction. During the apply transaction a set of replica data items is modified as given by a *writeset*.

In the simulate-apply mode an update request intercepted by the middleware is sent to the selected replica for performing the simulate transaction. The replica performs the simulate transaction and sends the middleware the result with the corresponding writeset and optionally the readset. The middleware do some housekeeping and sends a response back to the client. The housekeeping includes detecting concurrency conflicts, ordering the writesets and accumulating the writesets in a propagation queue. The middleware performs periodic or on-demand update propagation, when the accumulated writesets are sent to all replicas, including the one that performed the simulate transaction to perform the *apply* transaction.

In the update-apply mode the selected replica performs the update transaction and sends the middleware the result with the corresponding writeset. The middleware do the housekeeping and sends a response back to the client. During update propagation the accumulated writesets are sent to all replicas, except the one that performed the update transaction, to perform the *apply* transaction.

The mechanism for performing simulate transactions is similar to the multi-version concurrency control [8] mechanism in database systems. New versions are created for data items being modified by the simulate transaction. But these new versions are private to the transaction and only exist until the writeset/readset is calculated. There is no locking required during simulate transactions. Simulate transactions are totally non-blocking like read-only transactions.

B. Replica Operation

Each replica maintains a *Replica Version Number* (*RVN)* variable, which gives the system version currently installed at

the replica. Each incoming transaction (read-only, update, simulate or apply) in a replica contains a *reQuired Version Number* (*QVN*) tag. There is priority queue in each replica to store the incoming transactions ordered by increasing values of their *QVN*. A transaction in the queue is ready to be executed as soon as the *RVN* of the replica is ≥ the *QVN* of the transaction. The replica may execute the ready transactions in parallel provided the local concurrency control maintains linearizability.

After executing a read-only, update or simulate transaction, the result is tagged with the *RVN* of the replica the transaction has observed and is sent to the middleware as response. In the case of an update or simulate transaction the writeset and optionally the readset is also included in the response.

The value of the *RVN* in the replica is incremented by one after executing each apply transaction to reflect the latest system version it has observed from the update propagation. The newly created *RVN* is sent to the middleware as the result of the apply transaction.

C. Middleware operation

The middleware maintains a set of data structures to facilitate the protocol operation. The *Global Version Number* (*GVN*) variable stores the current version of the system state. Any successful update or simulate transaction produces a new version of the system state. This is reflected by incrementing the *GVN*.

Each ongoing client session has a *Session Version Number* (*SVN*). It stores the highest system version the client has observed from its completed transactions. This is either the highest *RVN* observed by the read-only transactions or the highest *GVN* created by the update or simulate transactions from the session, whichever is higher.

The middleware and the replicas communicate with each other asynchronously and the middleware does not know the exact value of the *RVN*s at the replicas. The middleware maintains a lower bound *RVNlo* and an upper bound *RVNhi* for the *RVN* at each replica. *RVNlo* is the version number of the replica confirmed most recently to the middleware that the replica has applied in it. *RVNhi* is the highest version number propagated to the replica for performing apply transaction. At any particular time for a transaction sent to a replica, the *RVN* of the replica when processing the transaction will be in between *RVNlo* and *RVNhi* inclusive.

The middleware also maintains the *propagation queue*, which stores the writesets from successful update and simulate transactions that create new *GVN*s. The indexes of the queue correspond to the past and present *system versions* and each index location stores a 3-tuple: identification of the replica executing the transaction, the writeset creating the version and the creation time of the version in real clock time. The *GVN*, the *SVN*s for the sessions and the *RVNlo* and *RVNhi* for the replicas are represented as pointers to this queue. Figure 5 illustrates the relationship among the propagation queue and the state variables for a system with 3 replicas and 5 client sessions. The (*X*+*5*)-th version is the most recent system version and hence is the *GVN* of the system that was created by a transaction executed in Replica 2 with the writeset WS_{X+5} at time RT_{X+5}. Replica 1 has received the writesets of up to the (*X*+*4*)-th system version and acknowledged applying writesets up to *X*-th system version. Session 1 has observed up to the (*X*+*4*)-th system version.

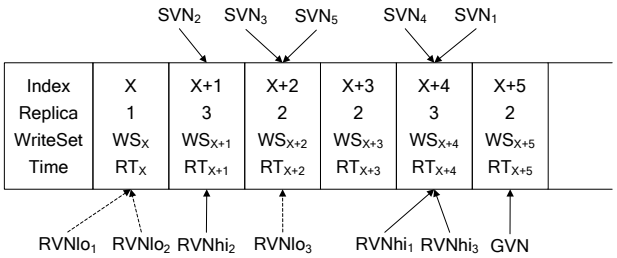

Figure 5. Propagation Queue

The middleware performs three main tasks: (1) handling requests from the clients, (2) handling responses from the replicas and (3) performing update propagation.

Handling Requests

Upon receiving a client request the middleware selects a replica to process the transaction corresponding to the request. The replica selection criteria for different correctness criteria is summarized in TABLE I. For guarantee the linearizability the selected replica should have an *RVNhi* = the *GVN* of the system. Sequential consistency, session snapshot isolation and causal consistency requires a replica with *RVNhi* ≥ the *SVN* of the client originating the request. If there is no replica that fulfills the selection criteria, a replica from the set of all replicas is brought up-to-date by performing update propagation to it and is selected for the transaction. For one-copy serializability and generalized snapshot isolation any of the replicas can be selected regardless of its *RVNhi*. When selecting a replica from a set of replicas, a load balancing strategy can be followed for better system performance.

The transaction is then tagged with a *QVN*, the replica version the transaction was expected to observe from replica selection, and sent to the selected replica for execution. For a read request the transaction type is read-only. For an update request the transaction type is update for causal consistency and simulate for the rest of the correctness criteria. For linearizability, sequential consistency and one-copy serializability it is required that the responses for the simulate/update transactions include both the readset and the writeset. Session snapshot isolation, generalized snapshot isolation and causal consistency requires only the writeset.

TABLE I. HANDLING REQUESTS

Correctness Criteria	Replica Selection	QVN	Request	Response
Linearizability	*RVNhi = GVN*	*GVN*	Simulate	Readset Writeset
Sequential Consistency	*RVNhi ≥ SVN*	*SVN*	Simulate	Readset Writeset
One-Copy Serializability	Any	0	Simulate	Readset Writeset
Session Snapshot Isolation	*RVNhi ≥ SVN*	*SVN*	Simulate	Writeset
Generalized Snapshot Isolation	Any	0	Simulate	Writeset
Causal Consistency	*RVNhi ≥ SVN*	*SVN*	Update	Writeset

TABLE II. HANDLING RESPONSES AND UPDATE PROPAGATION

Transaction	GVN	RVNhi	RVNlo	SVN
Read-Only	-	-	-	max(*SVN*, *RVN*)
Simulate	++	-	-	*GVN*
Update	++	-	-	*GVN*
Apply	-	-	*RVN*	-
Update Propagation	-	*GVN*	-	-

Handling Responses

How the middleware handles an incoming response from the replicas depends on the type of the transactions the replicas have performed. The result of handling responses is summarized in TABLE II.

For a read-only transaction the middleware updates the *SVN* pointer for the session originating the transaction. The session now has observed the system state given by the *RVN* included in the response. The middleware then responds the client with the result of the transaction from the replica.

For a simulate transaction the writeset and the readset (if present) accompanying the response is tested for possible conflicts with any concurrent simulate transactions in other replicas. If there is a conflict, the simulate operation can be resent to a fresher replica for retry for a number of times. If there is still conflict after the retries, the response is discarded and the client is notified of the conflict. The client can resubmit the request later. If there is no conflict the writeset is accepted as a change of system state, thus incrementing the *GVN*. The replica id, the writeset and the current time in the middleware are placed at the location in the propagation queue corresponding to the newly created *GVN*. The *SVN* pointer of the session originating the transaction is set to this *GVN* and result of the transaction from the replica is sent to the client.

We now determine if a readset or a writeset XS_P from a simulate operation is in conflict with any writeset WS_Q in the propagation queue. Let Ver(XS_P) be the system version XS_P is obtained on. Ver(XS_P) is given by the *RVN* accompanying the response corresponding to XS_P. Also let Ver(WS_Q) be the system version created by WS_Q. Ver(WS_Q) is given by the index of propagation queue WS_Q is located in. Then XS_P is in conflict with WS_Q, if Ver(XS_P) < Ver(WS_Q) and $XS_P \cap WS_Q \neq \varnothing$. Which means XS_P is in conflict with any writeset which has already created a system version newer than the version where XS_P was obtained and both the writesets modify at least one common data item.

For an update transaction the writeset accompanying the response is accepted as a change of system state and the *GVN* is incremented. The replica id, the writeset and the current time in the middleware are placed at the location in the propagation queue corresponding to the newly created *GVN*. The *SVN* pointer of the session originating the transaction is set to this *GVN* and the result of the transaction is sent to the client.

When an apply transaction returns from a replica the *RVNlo* pointer of the replica is updated to point the location in the propagation queue corresponding to the *RVN* in response.

Update Propagation

Update propagation from the middleware to the replicas is done optimistically and periodically as dictated by the divergence bound. The middleware maintains a *Version Threshold* and a *Delay Threshold* variable, which can be set to initiate update propagation when a replica reaches the divergence bound given in version metric or delay metric respectively. When the *RVNhi* of a replica falls behind the *GVN* by the *Version Threshold* or when the creation time of the *RVNhi* falls behind the system time in the middleware by the *Delay Threshold* the writesets from the propagation queue, starting from the index right after the *RVNhi* pointer up to the current *GVN*, are sent to the replica for performing apply transactions. Only exception is for causal consistency – if a writeset was originated from the replica the update propagation is destined to, an empty writeset is sent instead of the one in the propagation queue. This is because the destination replica has already performed an update transaction, not a simulate transaction, which generated the writeset. The *RVNhi* pointer of the replica is then advanced to the current *GVN*. Update propagation is also initiated when there is no replica fresh enough to fulfill the replica selection criteria. During update propagation, the *QVN* of an apply transaction containing a writeset that created the system version *V* is set to *V*-1. This is to ensure that writesets are applied in the replicas in the same order given by their order in the propagation queue.

The propagation queue in the middleware keeps growing with update requests from the clients. But a writeset is only needed to be kept in the queue until all the replicas confirm that they have performed the apply transaction for it. These are the writesets before the leftmost *RVNlo* pointer in the propagation queue, which can be purged to reduce the queue size. But the propagation queue can still be growing if the rate of arrival of the update requests is more than the replicas can handle. A *propagation queue size threshold* can be used as a mechanism for access control or as an indication when more replicas should be added to the system.

D. Fault Tolerance

The failure model assumed in McRep is node failure – the replicas and the middleware, link failure is not considered. All failures are assumed to be fail-stop.

Replica Failure

The replicas should be equipped with persistent storage. So that when a replica fails, the data in it along with the *RVN* can be recovered to a stable state before failure. The transactions being executed and transactions in the replica queue are lost. The clients will wait for the responses for the requests and eventually timeout. They can resend the requests and a different replica can process them next time. The middleware can detect a replica failure if does not accept any transaction for a certain amount of time. The middleware stops sending subsequent transactions to the replica. After the replica is repaired it sends a recovery message to the middleware along with its *RVN*. The middleware then initiates an update propagation starting from the provided *RVN*. The *RVNlo* pointer in the middleware ensures that there is enough writesets in the propagation queue required to recover the replica. The replica starts its regular operation from there.

Middleware Failure

The middleware is the most important component in this architecture. It is also a source of single point failure. But the middleware has a simple design and it performs simple tasks. The data structure the middleware maintains is a queue and

some pointers to the queue. The tasks performed by the middleware are routing client requests, resolving conflicts and update propagation. Traditional high availability techniques like active [28] or passive [10] replication can be used to eliminate the single point failure of the middleware without sacrificing any performance because of its simplicity.

VI. Performance Analysis

We have developed a simulation model for the McRep protocol in order to analyze its performance. The model is implemented using Tortuga [34], a framework for discrete-event simulation in Java. We also used the statistics package of Java Simulation Library [35] for generating timings of events according to different random distributions.

A. Simulation Model

The simulation model represents entities for the clients, the middleware and the replicas. TABLE III lists the parameters used in the simulation and the default values used for them.

TABLE III. Simulation Model Parameters

Parameter	Default Value	Parameter	Default Value
numReplicas	5	*sizeDataItem*	1,000 byte
numClients	Varies	*sizeDataID*	8 byte
updateProb	20%	*minDataReads*	2
meanThinkTime	5 sec	*maxDataReads*	10
conflictTime	0.03 msec	*minDataWrites*	1
corrCriteria	Varies	*maxDataWrites*	5
divThreshold	25	*dataReadTime*	5 msec
numDataItems	1,000,000,000	*dataWriteTime*	8 msec

There are *numClients* clients and *numReplicas* replicas. Each simulated client repeatedly generates requests following a Poisson distribution with a mean of *meanThinkTime* time unit between subsequent requests. Each request is an update request with a probability of *updateProb*, otherwise it is a read request. The simulated middleware routes requests from clients to replicas and responses from replicas to clients. It also maintains the simulated propagation queue and related data structures. Conflict detection between a pair of readset or writeset in the middleware requires *conflictTime* time unit. The version threshold of the replicas for initiating update propagation is *divThreshold* (delay threshold is not used). Each replica contains a simulated data store of *numDataItems* items with each item having a size of *sizeDataItem* and requiring an identifier size of *sizeDataID* to uniquely identify each of them. Each read or update request requires reading a number of data items between *minDataReads* and *maxDataReads* with a uniform distribution. Each update request requires updating a number of data items in between *minDataWrites* and *maxDataWrites* with a uniform distribution. Time required to read and update a single data item is exponentially distributed with a mean of *dataReadTime* and *dataWriteTime* respectively. There is no concurrency within a client or a replica, but there are three simulated threads in the middleware to execute its three tasks concurrently.

For each experiment the simulation run lasts for 15 simulated minutes. First 5 minutes of the simulation time are allowed to warm up the system and stabilize from initial jitters. Measurements used in the experiments are collected from the remaining 10 minutes of the simulation.

B. Experimental Results

In the first experiment we compare the performance of the protocol for six different correctness criteria in terms of average response time, see Figure 6. The experimental setting consists of 5 replicas subjected to workloads from an increasing number of clients (500, 550, 600, 650 and 700). The experiment is repeated for the following correctness criteria configured in the middleware – **cc1**: linearizability, **cc2**: sequential consistency, **cc3**: one-copy serializability, **cc4**: session snapshot isolation, **cc5**: generalized snapshot isolation and **cc6**: causal consistency. The average response time for the client requests is computed for each simulation run. The result of the experiment shows at low workload the protocol performs similarly for all the correctness criteria. But as load increases cc1, cc2 and cc3 become saturated faster than the rest; cc4 and cc5 holds on a bit longer; and cc6 maintains the best response time. This is due to the fact that cc1, cc2 and cc3 use both the readset and the writeset from simulate transactions in conflict detection, making their conflict detection time longer than cc4 and cc5 that only use the writeset. Also there is no conflict detection in cc6 since causal consistency is a conflict free correctness criteria.

In the second experiment we compare the number of concurrency conflicts occurring for different correctness criteria as a percentage of the number of update transactions, see Figure 7. In this experiment the replicas are subjected to workloads from an increasing number of clients (500, 600 and 700). In order to cause measurable number of conflicts the workload is composed of 50% update requests and the number of data items in the replicas is reduced to 5,000,000. We observe that cc1, cc2 and cc3 results in higher number of concurrency conflicts than cc4 and cc5 and there is no conflict in cc6. The explanation is same as the previous experiment.

We repeated the previous two experiments with different values of divergence bounds (version thresholds 25, 100, 500 and 1000) for a given correctness criteria (cc4). The results of the experiments demonstrate that unlike correctness criteria divergence bound has no implication on the response time or the number of conflicts in the system. Since update propagation is done asynchronously in all cases, there is no significant difference in doing it sooner or later.

In the next experiment we observe the effect of divergence bound on the number of update propagations for different correctness criteria. 5 replicas are subjected to workloads from 600 clients with 6 correctness criteria and different values for version thresholds (10, 25, 50 and 100). Figure 8 shows the result of the experiment in terms of number of update propagation messages in the system as a percentage of all the messages (read, update, simulate and apply transactions sent to the replicas, and their replies). We observe that the number of update propagation messages for cc1 is the same irrespective of the value of the divergence threshold and this number is highest among all the correctness criteria. This is because in linearizability every new writeset is propagated to all the replicas immediately instead of accumulating them. The number of update propagation messages remains almost the same for the rest of the correctness criteria for a given divergence threshold. But as the value of the divergence threshold increases the number of update propagation messages decreases.

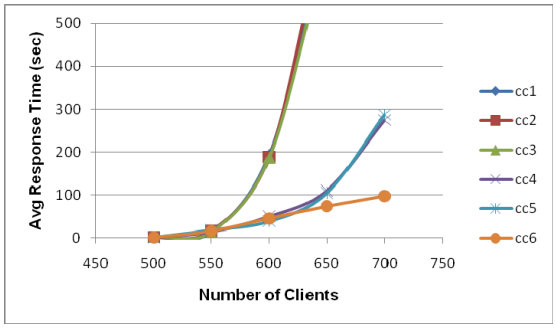

Figure 6. Performance in Avg Response Time

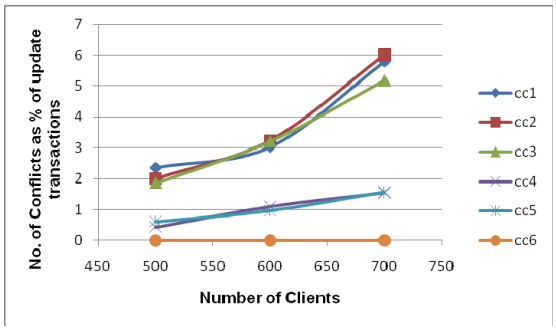

Figure 7. Number of Conflicts

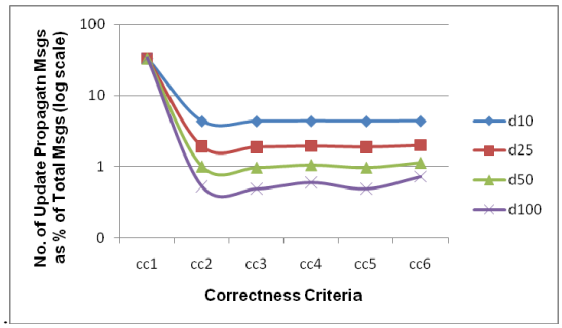

Figure 8. Number of Update Propagations

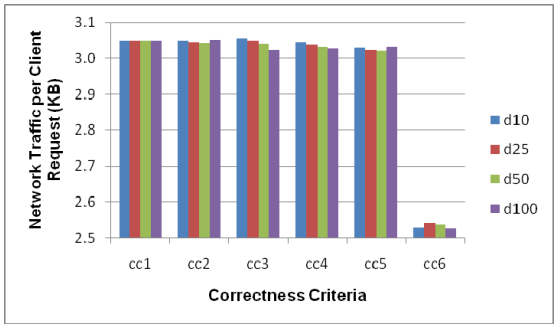

Figure 9. Network Traffic in the System

In the last experiment we examine the overhead of the protocol in terms of network traffic generated in the system, to and from the middleware and the replicas. The experimental setting is the same as the previous experiment. Figure 9 compares the overhead network traffic per client request. Although previous experiment showed that update propagation messages increase with decreasing divergence bound, this experiment demonstrates the resulting total network traffic remains the same irrespective of divergence bound. Divergence bound has no implication on the amount of generated network traffic. All the correctness criteria generate almost the same amount of traffic; except cc6 generates 15% less traffic than the rest. This is since causal consistency follows the update-apply transaction mode for the update requests, where a replica creating a writeset does not require the writeset to be propagated to it during update propagation.

We summarize the results of the experiments in TABLE IV. Correctness criteria cc6 performs the best in all four measures; cc4 and cc5 are the second best is performance; cc1, cc2 and cc3 cost more with cc1 being the worst in one performance measure. We also observe that divergence bound only affect the number of update propagation messages. We conclude, in McRep stronger consistency guarantee costs higher performance penalty.

TABLE IV. CORRECTNESS CRITERIA COMPARIOSON

Response Time	Conflicts	Update Prop. Msgs	Network Traffic	
cc1	cc1	cc1	cc1	Worse
cc2	cc2	cc2	cc2	
cc3	cc3	cc3	cc3	
cc4	cc4	cc4	cc4	
cc5	cc5	cc5	cc5	
cc6	cc6	cc6	cc6	Better

VII. RELATED WORK

The PRACTI [5] replication architecture provides a range of consistency and coherency guarantee along with partial replication and topology independence. The consistency guarantees provided by PRACTI are bounded divergence in terms of version metric and delay metric, application specific write-write conflict detection and two-phase update transaction. McRep provides more choices for correctness criteria than PRACTI. Also unlike PRACTI, McRep does not use any locking mechanism for providing a strong consistency. Transaction processing is always asynchronous in McRep.

Atul Adya in his work [1] redefines and further extends the consistency levels defined in the extended ANSI SQL isolation levels [6]. He provides implementation techniques to facilitate the optimistic implementation of the consistency levels in a distributed database, where a transaction may encompass more than one node, rather than a fully replicated database.

AQuA [18] is a group communication based middleware that provides a total or a FIFO ordering guarantee of the update operations among the replicas along with a divergence bound in terms of version metric. Ganymed [24] is a centralized replication middleware aimed to provide scalability and snapshot isolation guarantee. The conflict-aware scheduler in [3] also uses a centralized middleware to provide scalable one-copy serializable consistency. The middleware based snapshot isolation protocol of [21] can be implemented using a centralized or a decentralized group communication based middleware. In comparison to all these middleware, McRep provides a wider range of consistency guarantees.

BaseCON [33] is group communication based middleware that provides the first three of McRep's six correctness criteria. The performance evaluation of BaseCON confirms our result that there is no significant performance difference among these three criteria. BaseCON uses a pessimistic transaction processing approach and hence there is no divergence allowed.

VIII. CONCLUSION

We present the McRep middleware for developing replicated systems with various consistency requirements. Experiments demonstrate that in McRep divergence bound has very little consequence on the system performance. This suggests asynchronous transaction processing is good enough for performance and scalability, being lazier does not add anything extra.

The correctness criteria can be divided into four groups according to their performance: (1) linearizability, (2) sequential consistency and one-copy serializability, (3) session snapshot isolation and generalized snapshot isolation and (4) causal consistency. This suggests providing session guarantee has no additional cost, but other than that stronger consistency guarantee costs higher performance penalty. Concluding, workloads of replicated application developed using McRep middleware only pays for the consistency they actually need.

REFERENCES

[1] Atul Adya, Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. PhD Thesis, MIT Laboratory for Computer Science, Cambridge, MA, March 1999.

[2] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli and Phillip W. Hutto, Causal memory: definitions, implementati\on, and programming. Distributed Computing, Vol. 9 No. 1, March 1995.

[3] Cristiana Amza, Alan Cox and Willy Zwaenepoel, Conflict-aware scheduling for dynamic content applications. Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems, 2003.

[4] Daniel Barbara and Hector Garcia-Molina, The Case for Controlled Inconsistency in Replicated Data. Proceedings of the Workshop on the Management of Replicated Data, 1990.

[5] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula and Jiandan Zheng, PRACTI replication. Proceedings of the 3rd conference on Networked Systems Design & Implementation, May 2006.

[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil and Patrick O'Neil, A critique of ANSI SQL isolation levels. Proceedings of the ACM SIGMOD international conference on Management of data, June 1995.

[7] Philip Bernstein and Nathan Goodman, A Proof Technique for Concurrency Control and Recovery Algorithms for Replicated Databases. Distributed Computing, Vol. 2 No. 1, 1987.

[8] Philip Bernstein, Vassos Hadzilacos and Nathan Goodman, Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[9] Jerzy Brzezinski, Cezary Sobaniec and Dariusz Wawrzyniak, From Session Causality to Causal Consistency. Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004.

[10] N. Budhiraja, K. Marzullo, F. B. Schneider and S. Toueg, The primary-backup approach in Distributed Systems. Addison-Wesley, 1993.

[11] Khuzaima Daudjee and Kenneth Salem, Lazy Database Replication with Ordering Guarantees. Proceedings of the 20th IEEE International Conference on Data Engineering, 2004.

[12] Khuzaima Daudjee and Kenneth Salem, Lazy Database Replication with Snapshot Isolation. Proceedings of the 32nd international conference on Very large data bases, 2006.

[13] Weimin Du, Ahmed Elmagarmid and Won Kim, Maintaining Quasi Serializability in Multidatabase Systems. Proceedings of the 7th International Conference on Data Engineering, April 1991.

[14] Sameh Elnikety, Willy Zwaenepoel and Fernando Pedone, Database Replication Using Generalized Snapshot Isolation. Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems, October 2005.

[15] Hector Garcia-Molina, Using semantic knowledge for transaction processing in a distributed database. Transactions on Database Systems, Vol. 8 No. 2, June 1983.

[16] Maurice P. Herlihy and Jeannette M. Wing, Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, Vol. 12 No. 3, July 1990.

[17] Udo Kelter, Strictness and serializability. Proceedings of 3rd annual symposium on theoretical aspects of computer science, 1986.

[18] Sudha Krishnamurthy, William H. Sanders and Michel Cukier, An Adaptive Quality of Service Aware Middleware for Replicated Services. IEEE Transactions on Parallel and Distributed Systems, Vol. 14 No. 11, November 2003.

[19] Leslie Lamport, How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, Vol. 28 No. 9, 1979.

[20] David E. Langworthy, Evaluating correctness criteria for transactions. Proceedings of the ACM SIGPLAN workshop on Object-based concurrent programming, 1988.

[21] Yi Lin, Bettina Kemme, Marta Patiño-Martínez and Ricardo Jiménez-Peris, Middleware based data replication providing snapshot isolation. Proceedings of the ACM SIGMOD international conference on Management of data, June 2005.

[22] Christos H. Papadimitriou, The serializability of concurrent database updates. Journal of the ACM, Vol. 26 No. 4, October 1979.

[23] Fernando Pedone, Rachid Guerraoui and André Schiper, The Database State Machine Approach. Distributed and Parallel Databases, Vol. 14 No. 1, July 2003.

[24] Christian Plattner and Gustavo Alonso, Ganymed: scalable replication for transactional web applications. Proceedings of the 5th international conference on Middleware, October, 2004.

[25] Calton Pu, Wenwey Hseush, Gail Kaiser Kun-Lung Wu and Philip Yu, Distributed divergence control for epsilon serializability. Proceedings of the 13th International Conference on Distributed Computing Systems, 1993.

[26] Michel Raynal, Gérard Thia-Kime and Mustaque Ahamad, From serializable to causal transactions for collaborative applications. Proceedings of the 23rd EUROMICRO Conference, September 1997.

[27] Torval Riegel, Christof Fetzer, Heiko Sturzrehm and Pascal Felber, From causal to z-linearizable transactional memory. Proceedings of the 26th annual ACM symposium on Principles of distributed computing, August 2007.

[28] Fred B. Schneider, Replication management using the state-machine approach. Distributed systems (2nd Edtition), ACM Press/Addison-Wesley Publishing, May 1993.

[29] Douglas Terry, Alan Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer and Brent Welch, Session Guarantees for Weakly Consistent Replicated Data. Proceedings of 3rd International Conference on Parallel and Distributed Information Systems, September 1994.

[30] Douglas Terry, Marvin Theimer, Karin Petersen, Alan Demers, Mike Spreitzer and Carl Hauser, Managing update conflicts in Bayou, a weakly connected replicated storage system. Proceedings of the 15th ACM symposium on Operating systems principles, December 1995.

[31] Haifeng Yu and Amin Vahdat, Design and Evaluation of a Continuous Consistency Model for Replicated Services. 4th Symposium on Operating Systems Design and Implementation, October 2000

[32] Wanlei Zhou, Li Wang and Weijia Jia, An analysis of update ordering in distributed replication systems. Future Generation Computer Systems, Volume 20 Number 4, May 2004.

[33] V. Zuikeviciute and F. Pedone, Correctness Criteria for Database Replication: Theoretical and Practical Aspects. Proceedings of the 10th International Symposium on Distributed Objects, Middleware, and Applications, November, 2008.

[34] MITRE Corporation. Tortuga, discrete-event simulation framework in Java. 2004/2006. (http://code.google.com/p/tortugades/).

[35] Manuel D. Rossetti. JSL, a Java simulation library. 2008/2009. (http://bitbucket.org/rossetti/jsl/).