Correctness Criteria for Database Replication: Theoretical and Practical Aspects

Vaidė Zuikevičiūtė and Fernando Pedone

Faculty of Informatics
University of Lugano
6900 Lugano, Switzerland

University of Lugano
Faculty of Informatics
Technical Report No. 2008/03
August 2008

Abstract. In this paper we investigate correctness criteria for replicated databases from the client's perspective and present their uniform characterization. We further study the effects of different consistency degrees in the context of three middleware-based replication protocols: primary-backup, optimistic update-everywhere and *BaseCON*, a simple yet fault-tolerant middleware-based replication protocol that takes advantage of workload characterization techniques to increase the parallelism in the system. We present three variants of BaseCON, one for each correctness criterion discussed, and analyze its behavior in case of failures and false suspicions.

We have implemented the correctness properties in all three protocols considered and show experimentally that stronger consistency does not necessarily imply worse performance. On the contrary, two of the three replication protocols evaluated show no significant performance divergence while ensuring different consistency criterion.

1 Introduction

Middleware-based database replication protocols have recently received a lot of attention [1–7]. The main reason for this stems from the fact that middleware protocols require few or no changes in the database engine. As a consequence they can be maintained independently of the database engine, and can potentially be used in heterogeneous settings. Just like kernel-based protocols, middleware-based database replication protocols can implement a large variety of consistency criteria, ensuring different degrees of guarantees to the system's clients. In this paper we investigate the performance cost of implementing different consistency degrees in middleware protocols.

A typical correctness criterion for replicated database systems is *one-copy serializability* (1SR) [8]. Informally, 1SR requires the execution of concurrent transactions on different replicas to appear as if transactions were executed in some sequential order on a single replica. 1SR does not necessarily preserve the order in which transactions are submitted to the system. It allows the situation where transaction T_j may not see the effects of T_i, even though T_i commits before T_j started executing. In some cases such reordering of transactions may be unacceptable. For instance, if both transactions are sumbitted by the same client, intuitively the client expects to read what it has written before. A stronger correctness criterion, named *strong serializability* [9], requires transactions to be serialized according to their real-time ordering. However, strong serializability may be too restrictive, since it requires the enforcement of transaction ordering constraints among all transactions, which may be unnecessary and costly to ensure. *Session consistency* [10] is stronger than one-copy serializability, but weaker than strong serializability: it preserves the real-time ordering of transactions submitted by the same client only; as a consequence, clients always read their own updates.

In this paper we focus on two broad categories of replication protocols [11]: primary-backup and update everywhere replication. Two of the protocols we consider belong to the update-everywhere replication category, however they differ in the execution mode: one executes transactions optimistically, the other, conservatively. The conservative protocol, named *BaseCON*, is introduced in this paper. BaseCON is a simple middleware-based protocol that makes use of atomic broadcast primitives to provide strong consistency and fault-tolerance. False suspicions are tolerated and never lead to incorrect behavior. BaseCON takes advantage of workload characterization techniques to increase the parallelism in the system. A lightweight scheduler interposed between clients and the database servers allows the system to adapt easily to the correctness criterion required and serves as a load-balancer for read-only transactions. If a scheduler fails or is suspected to have failed, a backup scheduler takes over.

It has been generally believed that additional constraints on correctness degrades the performance of a replicated system. To verify this statement we present a thorough experimental evaluation of the three classes of replication protocols: optimistic and conservative update everywhere (respectively, DBSM [13] and BaseCON), and primary-backup (Pronto [12]). Contrary to the general believe, both BaseCON and DBSM show no significant performance difference even if the strongest correctness criterion is ensured. Even though the implementation of strong serializability requires ordering also for read-only transactions in all protocols studied, the overhead introduced by total order primitives is negligible in middleware-based replication. This may have surprising consequences. For example, our implementation of session consistency in primary-backup replication exhibits worse performance than strong serializability.

To sum up, this paper makes the following contributions: (1) it revisits the correctness criteria for replicated databases and uniformly characterizes them within the same model; (2) it proposes a simple although fault-tolerant replication protocol that takes advantage of workload characterization techniques to increase the parallelism in the system; (3) it shows how different correctness criteria could be implemented in three distinct classes of replication protocols; and (4) it evaluates experimentally the effects of different consistency criteria on middleware-based replicated system's performance using the three classes of protocols.

2 System Model and Definitions

2.1 Servers, Communication and Failures

We consider an asynchronous distributed system composed of database clients, $\mathcal{C} = \{c_1, c_2, ..., c_m\}$, and servers, $\mathcal{S} = \{S_1, S_2, ..., S_n\}$. Communication is by message passing. Servers can also interact by means of a total-order broadcast, described below. Servers can fail by crashing and subsequently recover.

Total-order broadcast is defined by the primitives broadcast(m) and deliver(m), and guarantees that (a) if a server delivers a message m then every server delivers m; (b) no two servers deliver any two messages in different orders; and (c) if a server broadcasts message m and does not fail, then every server eventually delivers m. The notion of uniform delivery captures the concept of durability, that is, a server must not forget that it has delivered a message after it recovers from a crash. After recovery the server delivers first all the messages it missed during the crashed period.

We do not make any assumptions about the time it takes for clients and severs to execute and messages to be transmitted. We do assume however that the system is augmented with unreliable failure detectors [14]. Such oracles can make an unbounded number of mistakes: servers may be incorrectly suspected to have failed even if they are just slow to respond. In order to ensure progress unreliable failure detectors satisfy the following two properties: there is a time after which some operational server is never suspected by any operational server (*accuracy*); and eventually every server that fails is suspected to have failed (*completeness*).

2.2 Database and Transactions

Each server has a full copy of the database. Servers execute *transactions* according to strict two-phase locking (2PL)[8]. Transactions are sequences of read and write operations followed by a commit or an abort operation. A transaction is called *read-only* if it does not contain any write operation; otherwise it is called an *update* transaction. Transactions within the same client session are submitted sequentially. The *workload* submitted to the database is composed of a set of predefined, parameterized transactions $\mathcal{T} = \{T_1, T_2, T_3, ...\}$. Each such transaction is identified by its type and parameters, provided by the application program when the transaction is instantiated. From its type and parameters, conflicts between transactions can be detected, even if conservatively, before the transaction is executed. Transaction T_i conflicts with T_j if they have *conflicting* operations. Two operations conflict if they are issued by different transactions, access the same data item and at least one of the operations is a write.

3 Correctness Criteria for Replicated Databases

In order to discuss and uniformly define the correctness criteria considered throughout this article, we recall some basic definitions introduced in [8].

A history H over a set of transactions $\mathcal{T}$ is a partial order with ordering relation $<_H$, where (a) H contains all the operations op of each transaction $T_i \in \mathcal{T}$; (b) $\forall T_i \in \mathcal{T}$ and $\forall op_i \in T_i$: if op_i precedes $op_i{}'$ in T_i, then $op_i <_H op_i{}'$; and (c) if T_i reads data item x from T_j, then $w_j[x] <_H r_i[x]$. A history H is *serial* if, for every two transactions T_i and T_j in H, either all operations of T_i happen before all operations of T_j or vice versa. Two histories H, H' are *view equivalent* ($\equiv$) if (a) they are over the same set of transactions; (b) for any T_i, T_j and data item x: if $w_j[x] <_H r_i[x]$, then $w_j[x] <_{H'} r_i[x]$; and (c) for each x, if $w_i[x]$ is the final write on x in H, then it is also the final write of x in H'. Transaction begin and commit events are defined from the clients perspective.

Typical correctness criterion for replicated systems is *one-copy serializability.*

Definition 1. History H is *one-copy serializable* iff there is some serial history H_s such that $H \equiv H_s$.

1SR permits the situation where transaction T_j may not see the updates of T_i even if T_i commits before T_j starts executing. Although some applications can accept this for performance, in most cases transactions expect to read the updates of specific preceded transactions. For example, the effect of an update transaction should be seen by a successive read-only transaction issued by the same client. In practice, transactions of the same client are executed within a session. Thus, at least transactions issued in one session should see the effects of each other.

In order to capture this additional requirement *session consistency* (SC) was introduced [10].

Definition 2. History H is *session consistent* iff there is some serial history H_s such that (a) $H \equiv H_s$ (i.e., H is 1SR) and (b) for any two transactions T_i and T_j that belong to the same session, if the commit of T_i precedes the submission of T_j in real time, then T_i commits before T_j is started in H_s.

Even stronger properties must be defined if we require that all transactions in the workload preserve the real-time order, i.e., any transaction reads updates of previously committed transactions. Such real-time ordering of transactions is captured by *strong serializability* (strong 1SR) introduced in [9].

Definition 3. History H is *strongly serializable* iff there is some serial history H_s such that (a) $H \equiv H_s$ and (b) for any two transactions T_i and T_j, if the commit of T_i precedes the submission of T_j in real time, then T_i commits before T_j is started in H_s.

Both session consistency and strong serializability strengthen the original correctness criterion by restricting what transactions are allowed to read. Thus, the notion of these stronger properties is valid regardless of how the original correctness criterion handles the execution of transactions. SC and strong 1SR are not fundamentally related to 1SR and they could be applied to other correctness criterion such as *snapshot isolation* (SI) [15].

4 The BaseCON Protocol

BaseCON is a conservative replication protocol which takes advantage of total-order broadcast primitives to provide strong consistency and fault-tolerance. False suspicions are tolerated and never lead to incorrect behavior. In this section we first describe the behavior of the algorithm that guarantees 1SR in a failure-free scenario and in case of failures; then we discuss two BaseCON variants, each of which ensures different consistency guarantees: SC and strong 1SR. Correctness proofs of each variant of BaseCON can be found in the Appendix.

4.1 One-Copy Serializability

BaseCON assumes a lightweight scheduler interposed between the clients and the cluster of database servers. Such scheduler serves as a load-balancer for read-only transactions and implements different consistency properties. The main challenge is to guarantee that despite failures and false suspicions of the scheduler, required consistency degree is still provided.

In a nutshell, BaseCON handles transactions as follows. Update transactions are atomically broadcast to all replicas and the scheduler. Every database server executes every update transaction. Although non-conflicting transactions can be executed concurrently at a server, their commit order should be the same at all replicas. As we show next, allowing inversions of transactions may violate serializability. The response to the client is given by the scheduler once the first reply is received from any of the replicas. Read-only transactions are sent to the lightweight scheduler, which redirects them to the selected replica for execution.

Algorithm 1 presents the complete BaseCON when no failures or false suspicions occur. The algorithm is composed of five concurrent tasks and several instances of $executeTask$. A new instance of $executeTask$ is created for each update or read-only transaction. Each line of the algorithm is executed atomically. Access to all shared variables is mutually exclusive.

Algorithm 1 The BaseCON Algorithm: 1SR

```
1:  Client $c_k$:
2:    if $T.isReadOnly$
3:      send(ReadOnly, $T$) to $D_k$
4:    else
5:      to_broadcast($T$)
6:  Scheduler $D_k$:
7:    $\forall S_k \in \mathcal{S} : Load[S_k] \leftarrow 0$
8:    $p \leftarrow 0$
9:    upon receive(ReadOnly, $T$)    {$T1$}
10:     if $p = D_k$
11:       $S_k^{min} \leftarrow min(Load[S_k], S_k \in \mathcal{S})$
12:       $Load[S_k^{min}] \leftarrow Load[S_k^{min}] + T.weight$
13:       $T.repId \leftarrow S_k^{min}$
14:       send(ReadOnly, $T$) to $S_k^{min}$
15:     else
16:       send(ReadOnly, $T$) to $p$
17:   upon to_deliver($T$)    {$T2$}
18:     if $p = D_k$
19:       $\forall S_k \in \mathcal{S} : Load[S_k] \leftarrow Load[S_k] + T.weight$
20:   upon receive(Result, $T$)    {$T3$}
21:     if $p = D_k$
22:       $Load[T.repId] \leftarrow Load[T.repId] - T.weight$
23:       if $T.repId$ is the first replica to execute $T$
24:         send(Result, $T.result$) to $c_k$
25: Replica $S_k$:
26:   $txnQ \leftarrow \epsilon$
27:   $p \leftarrow 0$
28:   function conflict($T$, $T'$)
29:     return $T.rs \cap T'.ws \neq \emptyset$
30:   upon to_deliver($T$)    {$T4$}
31:     $T.repId \leftarrow S_k$
32:     $prTxn \leftarrow txnQ$
33:     $enqueue(txnQ, T)$
34:     fork task executeTask($T, prTxn$)
35:   upon receive(ReadOnly, $T$)    {$T5$}
36:     fork task executeTask($T, \emptyset$)
37:   task executeTask($T, txnSet$)
38:     if $T.isReadOnly$
39:       submit($T$)
40:       $T.result \leftarrow$ commit($T$)
41:     else
42:       wait until $\nexists T' \in txnSet$ :
              conflict$(T, T') \wedge T' \in txnQ$
43:       submit($T$)
44:       wait until $T = head(txnQ)$
45:       $T.result \leftarrow$ commit($T$)
46:       $dequeue(txnQ, T)$
47:     send(Result, $T$) to $p$
```

Clients. Transactions are issued by one or more concurrent clients. Update transactions are atomically broadcast to all database replicas and the scheduler (line 5); read-only transactions are just sent to the scheduler D_k(line 3).

Replicas. The algorithm uses two global variables shared among all the tasks at each replica: a queue of update transactions to be executed, $txnQ$, and an identifier of the scheduler, p. Upon delivery of an update transaction the database server enqueues the transaction (line 33) and creates a new instance of $executeTask$ to process the transaction (line 34). Similarly, a new $executeTask$ is created once a read-only transaction is received (lines 35-36). Different instances of $executeTask$ can execute concurrently as long as the $txnQ$ data structure is thread-safe. If a transaction is read-only, it can be submitted to the database for execution and committed straightaway (lines 39-40). If a transaction is an update, the server checks whether there are any conflicts with previously received but not yet completed transactions (stored in $txnSet$). If there are no conflicts, the transaction is submitted to the database (line 43); if there are some conflicts, the transaction has to wait until conflicting transactions commit (lines 42). To ensure that all replicas converge to the same database state, conflicting update transactions must commit in the order they were delivered. However, if non-conflicting update transactions can commit in different orders at the replicas and read-only transactions are allowed to execute at any database server, serializability guarantees may be violated. Consider four transactions: T_1:$w_1[x]$, T_2:$w_2[y]$, T_3:$r_3[x], r_3[y]$ and T_4:$r_4[y], r_4[x]$. Since T_1 and T_2 do not conflict they can execute and commit at database servers in different orders. Let's assume that T_1 commits at S_1 and T_2 commits at S_2 first. Transaction T_3 is scheduled for execution at S_1 and T_4 at S_2; then S_1 commits T_2 and S_2 commits T_1. As a consequence, transaction T_3 sees the updates of T_1, but not those of T_2, while T_4 sees the updates performed by T_2 but not by T_1, and thus, violates serializability. To avoid situations like this, we require all update transactions to commit in their delivery order. Therefore, a commit is sent to the database server only after the update transaction has reached the head of $txnQ$, i.e., all previously delivered update transactions have completed already (lines 44-45). As soon as the transaction commits, the result is communicated to the scheduler (line 47).

Scheduler. We consider a primary-backup model to tolerate scheduler failures. There is only one scheduler at a time serving transactions, the *primary*. If the primary scheduler fails or is suspected to have failed, a *backup* scheduler takes over. Since our scheduler is lightweight, any replica can play the role of a backup scheduler.

To ensure 1SR the scheduler can forward read-only transaction to any replica, however, in order to balance the load we send the transaction to the least-loaded replica. Any load balancing strategy can be applied. The scheduler maintains current load information about each database server in *Load*[]. Once a read-only transaction is received by the primary scheduler, it is redirected to the replica with the lowest aggregated weight (lines 9-14). Upon delivery of an update transaction at the primary scheduler the load over all servers is increased by the weight of the transaction (line 19). The result of the transaction is communicated to the client as soon as the corresponding response for the read-only transaction is received; or the first response of the update transaction is received from any server (lines 23-24). Load information is updated with every reply from the database servers (line 22).

Dealing with failures. If a replica suspects the scheduler has failed (see Algorithm 2), a special NewScheduler message is atomically broadcast (lines 8-9). Upon delivery of this message, a new primary scheduler is selected from the backups (lines 11). If the scheduler was suspected incorrectly, it will also deliver the NewScheduler message and will update its state to a backup scheduler, thus, will stop serving read-only transactions (line 3). If a read-only transaction is received, it is immediately forwarded to the primary scheduler. A failover scheduler does not have any load information on the database servers. Therefore, replicas respond to the new scheduler with their load estimates required to handle read-only transactions (lines 12). Rarely, but it may happen that transaction results are lost during scheduler failover. Hence client application should be ready to resubmit transactions and ensure exactly-once semantics.

Since we do not make any assumptions on how long it takes for messages to be transmitted and failure detectors can make mistakes, it is possible that

Algorithm 2 The BaseCON Algorithm: Scheduler failover

```
Scheduler $D_k$:
 upon to_deliver(NewScheduler)
   $p \leftarrow (p+1) \bmod |\mathcal{D}|$
 upon receive(State, $load, S_k$)
   if $p = D_k$
     $Load[S_k] \leftarrow load$

Replica $S_k$:
 upon scheduler suspected
   to_broadcast(NewScheduler)
 upon to_deliver(NewScheduler)
   $p \leftarrow (p+1) \bmod |\mathcal{D}|$
   send(State,$|txnQ|, S_k$) to $p$
```

for a certain time period two schedulers may be able to process transactions simultaneously. Such scenario is depicted in Fig. 1. Client c_1 submits an update transaction T_1 which is atomically broadcast to all members of the system. Database server s_1 executes transaction T_1 and forwards the result to the scheduler p_1. Since it is the first reply for this transaction, the result is communicated to the client. Shortly after that, database server s_2 suspects the primary scheduler p_1 to have failed and broadcasts a NewScheduler message. Upon delivery of this message both database servers send their load information to the newly selected primary scheduler p_2. Database server s_2 executes transaction T_1 and since it has already delivered NewScheduler message, the result of T_1 is forwarded to the scheduler p_2. The old scheduler was suspected incorrectly: it is still fully functional, but since it hasn't yet delivered a scheduler change message it is unaware of the presence of the new scheduler. Consequently, it is unaware of transaction's T_1 commitment at replica s_2. If client c_2 submits read-only transaction T_2 to the old scheduler p_1, the transaction will be scheduled based on erroneous load information at the replicas. However, to ensure 1SR read-only transactions can execute at any replica, thus even if a scheduler makes decisions based on incomplete information, the consistency is still guaranteed, and only load-balancing may be affected. We further discuss the implications of failures on consistency in the context of SC and strong 1SR.

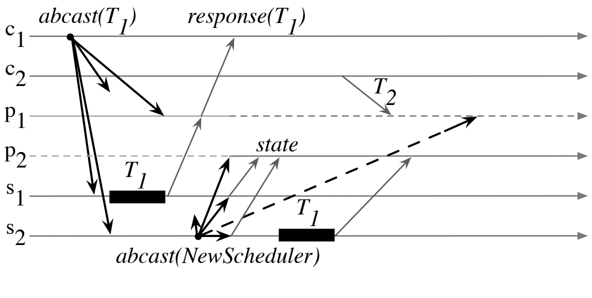


Fig. 1. Dealing with failures and false suspicions

We use atomic broadcast primitives just to simplify the replacement of a suspected scheduler. It is not required to ensure consistency. Since update transactions are also totally ordered, if a scheduler change occurs, all consequent update transaction will be processed by the new scheduler, i.e., all replicas will directly contact the new scheduler with results of processed transactions. If a scheduler change was not totally ordered with respect to update transactions, different schedulers would be contacted by different replicas after processing the same update transactions, and thus, more communication messages might be wasted for the client to receive the result of his transaction.

4.2 Session Consistency

To achieve session consistency the scheduler must forward read-only transactions to the replica which has committed previous update transactions of the same client.

We have modified BaseCON so that session consistency is ensured. The updated part of the algorithm is presented in Algorithm 3. In addition to the load information the scheduler also stores the identifier of the last update transaction committed per server in *Committed*[] (line 6). The identifier of an update transaction corresponds to its delivery and consequently commit order. Since some replicas might have fallen behind with the application of update transactions, for read-only transactions the scheduler first determines the set of replicas $\mathcal{I}$ where previous transactions of the same client have been completed. From this set the scheduler then selects the least-loaded replica as the replica to execute the read-only transaction (lines 13-17).

Algorithm 3 The BaseCON Algorithm: SC

```
1:  Client $c_k$:
2:    if $T.isReadOnly$
3:      send(ReadOnly, $T$, $id$, $rep$) to $D_k$

4:  Scheduler $D_k$:
5:    $p \leftarrow 0$
6:    $\forall S_k \in \mathcal{S} : Load[S_k] \leftarrow 0,$
                              $Committed[S_k] \leftarrow 0$
7:    upon receive(ReadOnly, $T$,$id$, $rep$)
8:      if $p = D_k$
9:        $\forall S_k \in \mathcal{S}$ :
10:         if $id \leq Committed[S_k]$
11:           $\mathcal{I} \leftarrow \mathcal{I} \cup S_k$
12:       if $\mathcal{I} \neq \emptyset$
13:         $S_k^{min} \leftarrow min(Load[S_k], S_k \in \mathcal{I})$
14:         $Load[S_k^{min}] \leftarrow Load[S_k^{min}]+$
                              $T.weight$
15:         $\mathcal{I} \leftarrow \emptyset$
16:         $T.repId \leftarrow S_k^{min}$
17:         send(ReadOnly, $T$) to $S_k^{min}$
18:       else
19:         send(ReadOnly, $T$) to $rep$
20:     else
21:       send(ReadOnly, $T$) to $p$
22:   upon receive(Result, $T$)
23:     if $p = D_k$
24:       $Load[T.repId] \leftarrow$
                    $Load[T.repId] - T.weight$
25:       if $\neg T.isReadOnly$
26:         $Committed[T.repId] \leftarrow T.id$
27:       if $T.repId$ is the first replica to
                                  execute $T$
28:         send(Result, $T.result$, $T.id$,
                          $T.repId$) to $c_k$
```

To guarantee session consistency in case of false suspicions, when submitting a read-only transaction T the client also sends information about its last committed update transaction T': the identifier *id* of T' and the replica *rep* which executed T' first (line 3). On receiving T the scheduler checks whether it has information about T''s commitment (line 10). If so, it is safe to schedule T based on data available at the scheduler; if not, the result of T' was given to the client by a newly introduced primary scheduler. Thus, the scheduler has no other choice than sending T where T' has executed (line 19) – no load-balancing is performed in this case.

4.3 Strong serializability

To implement strong serializability the scheduler must ensure that read-only transactions see all updates performed by previously committed transactions. Let's see how that can be guaranteed in a failure-free environment. The client sends its read-only transaction to the scheduler. Upon reception of the transaction the scheduler first determines the set of replicas where preceding update transactions of any client have already been executed and committed. Then the scheduler chooses the least-loaded server from that set and forwards the transaction for execution. Unfortunately, that is not enough for consistency if failures and false suspicions are tolerated: due to mistakes made by failure detectors two schedulers may simultaneously schedule read-only transactions. Therefore, to avoid inconsistent scheduler decisions, read-only transactions need to be atomically broadcast to all replicas as well. The scheduler still serves as a load-balancer: read-only transactions are executed at a single database server.

The algorithm presented in Algorithm 4 works as in the case without failures: read-only transactions are sent to the scheduler (line 4) and "optimistically" dispatched to the selected replica for execution (lines 10-15). In addition to this, the client also atomically broadcasts its read-only transaction (line 2). Further, in order to optimize the response time of read-only transactions in the absence of failures, we overlap the scheduling and the actual execution of the transaction with the time it takes for a replica to deliver it. Assuming that it takes less time to transmit a simple TCP message than an atomic broadcast message, when the chosen replica delivers the transaction, its execution has started already. A transaction can commit only if it passes the *validation* test at the time of the delivery. The validation test checks whether the transaction was executed and delivered during the operation of the same scheduler (line 31). If there were scheduler changes, the transaction is re-executed (lines 32-33), otherwise the transaction commits and its result is forwarded to the client (line 41).

4.4 Conflict detection

Conflicts between predefined transactions can be detected automatically before their actual execution by partially parsing SQL statements. It is relatively easy to determine tables accessed by the transaction, but table level granularity inevitably introduces spurious conflicts. Obviously, record level granularity is preferred, but more difficult to achieve. To avoid being too conservative, we use

Algorithm 4 The BaseCON Algorithm: strong 1SR

```
Client $c_k$:
  to_broadcast($T$)
  if $T.isReadOnly$
    send(READONLY, $T$) to $D_k$
Scheduler $D_k$:
  $p \leftarrow 0$
  $\forall S_k \in \mathcal{S} : Load[S_k] \leftarrow 0,$
                        $Committed[S_k] \leftarrow 0$
upon receive(READONLY, $T$)
  if $p = D_k$
    $\mathcal{I} \leftarrow max(Committed[S_k], S_k \in \mathcal{S})$
    $S_k^{min} \leftarrow min(Load[S_k], S_k \in \mathcal{I})$
    $Load[S_k^{min}] \leftarrow Load[S_k^{min}] +$
                        $T.weight$
    $T.repId \leftarrow S_k^{min}$
    $T.scheduler \leftarrow p$
    send(READONLY, $T$) to $S_k^{min}$
  else
    send(READONLY, $T$) to $p$
Replica $S_k$:
  $txnQ \leftarrow \epsilon$
  $p \leftarrow 0$
upon to_deliver($T$)
  $T.repId \leftarrow S_k$
  $prTxn \leftarrow txnQ$
  $enqueue(txnQ, T)$
  if $\neg T.isReadOnly$
    fork task executeTask($T, prTxn$)

task executeTask($T, txnSet$)
  if $T.isReadOnly$
    submit($T$)
    wait until $T \in txnQ$ :
    if $T.scheduler \neq p$
      rollback($T$)
      submit($T$)
    $T.result \leftarrow$ commit($T$)
  else
    wait until $\nexists T' \in txnSet$ :
            conflict($T, T'$) $\wedge T' \in txnQ$
    submit($T$)
    wait until $T = head(txnQ)$
    $T.result \leftarrow$ commit($T$)
  $dequeue(txnQ, T)$
  send(RESULT, $T$) to $p$
```

table level granularity only if there is not enough information to identify records accessed by the transaction.

A SELECT query in SQL retrieves a set of tuples from one or more tables. It can consist of up to six clauses, but only two, SELECT and FROM, are mandatory. Omitting the WHERE clause indicates that all tuples of the table are read. If the WHERE keyword is present it is followed by a logical expression, also known as a predicate, which identifies the tuples to be retrieved. If the predicate is an equality comparison on a unique indexed attribute, the exact rows scanned by the database can be estimated. Unique index of the table can be also composed of several attributes. In both cases the database locks only the records identified by such an index and thus, the accessed records can be identified easily by their unique indices. However, there are many situations when the database scans the whole table to retrieve a set of particular records. The WHERE clause might contain other comparison operators ($<, >, <=, >=$), complex expressions that consist of extended functions (LIKE, BETWEEN, NOT NULL) or another (nested) query. The records physically read by the database processing such queries are DBMS implementation dependent. Since we aim at middleware-based replication, table level granularity is used in all the situations above and the cases not covered.

There are three SQL statements for data modification: INSERT, UPDATE and DELETE. INSERT sets an exclusive lock on the inserted tuples, thus the exact rows can be estimated by their primary keys.[1] The tuples accessed by both, DELETE and INSERT statements, similarly to SELECT, can be retrieved at the record-level only if the WHERE clause contains an equality comparison on unique identifiers. Otherwise, table-level granularity is considered.

5 Evaluation

In this section we evaluate experimentally the performance of BaseCON under different correctness criteria and compare it with primary-backup and optimistic update-everywhere replication solutions.

5.1 Experimental environment

System specification. All experiments were run in a cluster of servers, each equipped with 2 Dual-Core AMD Opteron 2GHz processors, 4GB RAM, and an 80GB HDD. The system runs Linux Ubuntu 2.6 and Java 1.5. We used MySQL 5.0 with InnoDB storage engine as the database server.

TPC-C Benchmark. TPC-C is an industry standard benchmark for online transaction processing (OLTP) [16]. It represents a generic wholesale supplier workload. TPC-C defines 5 transaction types, each of which accounts for different computational resources: *New Order*, *Payment*, *Delivery*, *Order Status* and *Stock Level*. *Order Status* and *Stock Level* are read-only transactions; the others are update transactions.

In all experiments we used a TPC-C database, populated with data for 8 warehouses. The experiments were conducted on a system with 4 replicas under various load conditions. Clients submit transactions as soon as the response of the previously issued transaction is received (i.e., we do not use TPC-C think times). Since providing stronger correctness criterion has greater impact on read-only transactions, we have increased the percentage of read-only transactions in the TPC-C workload mix. In particular, we present results of two workloads: TPC-C 20, which contains only 20% of update transactions; and TPC-C 50, which represents TPC-C workload with balanced mix of update and read-only transactions. We measure throughput in terms of the number of transactions committed per second (tps). The response time reported represents the mean response time of committed transactions in milliseconds (msec). Both throughput and response time are reported separately for update and read-only transactions.

5.2 Replication protocols

Besides BaseCON, we also study the effects of different correctness criteria on primary-backup and optimistic update-everywhere replication solutions.

[1] We assume MySQL InnoDB engine here. Other DBMSs may handle INSERT and DELETE statements differently (e.g., using table locks).

Primary-backup replication. Primary-backup replication requires all transactions to be submitted to the same dedicated server, the primary replica. The requests are executed at the primary and only the updates are sent to the backups. The communication between the primary and the backups has to guarantee that updates are processed in the same order at all replicas.

As an example we have chosen the *Pronto* primary-backup replication protocol [12]. In a nutshell the protocol works as follows. Clients submit update transactions to the primary replica. Once the request is executed and ready to commit, the primary broadcasts update SQL statements to all backups. To ensure consistency despite multiple primaries resulting from incorrect failure suspicions, a total-order broadcast is used to propogate updates. Upon delivery of updates each server executes a deterministic validation test. The validation test ensures that only one primary can successfully execute a transaction. If the replica decides to commit the transaction, the update statements are applied to the database. The response is given to the client as soon as any replica commits the transaction. Read-only transactions are submitted and executed at random replica.

The original Pronto protocol guarantees 1SR. The easiest way to provide session consistency in Pronto is to require clients to submit their read-only transactions to the replica which was the first to commit the previous transaction of the same client. To guarantee strong serializability the protocol needs to ensure that the effects of committed transactions are visible to all following transactions. To provide this in Pronto, read-only transactions are atomically broadcast to all replicas. However, there is no need to require all replicas to execute the query: one randomly chosen replica executes the transaction and replies to the client.

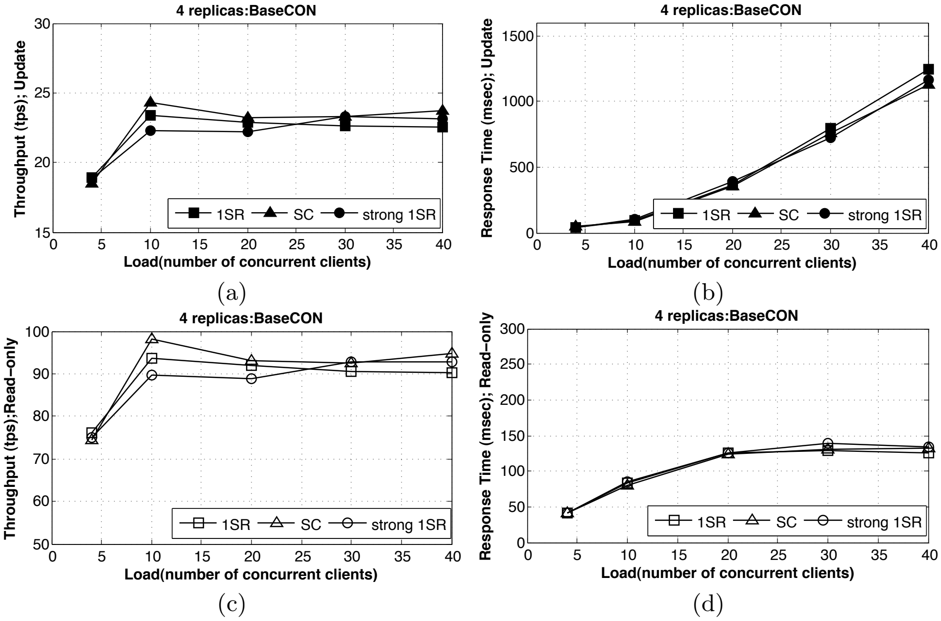
Update everywhere replication. The main difference between primary-backup and update everywhere replication is that in the update everywhere approach any replica can execute any transaction. The key concept is that all replicas receive and process the same sequence of requests in the same order. Consistency is guaranteed if replicas behave deterministically, that is, when provided with the same input each replica will produce the same output.

For the discussion we have chosen the *Database State Machine* replication (DBSM) [13]. The protocol works as follows. Each transaction is executed locally on some server and during the execution there is no interaction between replicas. Read-only transactions are committed immediately locally. When an update transaction is ready to be committed, its updates, readsets, and writesets are broadcast to all replicas. All servers receive the same sequence of requests in the same order and certify them deterministically. The certification procedure ensures that committing transactions do not conflict with concurrent already committed transactions. Once passed the certification the updates are applied to the replica. The response to the client is given by the replica that executed the transaction locally.

Session consistency can be trivially attained enforcing that the client always selects the same replica for executing its transactions. As in primary-backup replication strong serializability is ensured if read-only transactions are atomically broadcast to all replicas. The delivery of such a transaction is ignored by all but one replica: the read-only transaction is executed at a selected database server.

5.3 Performance results

BaseCON. Figures 2 and 3 show the achieved throughput and response time of read-only and update transactions for TPC-C 20 and TPC-C 50, respectively.

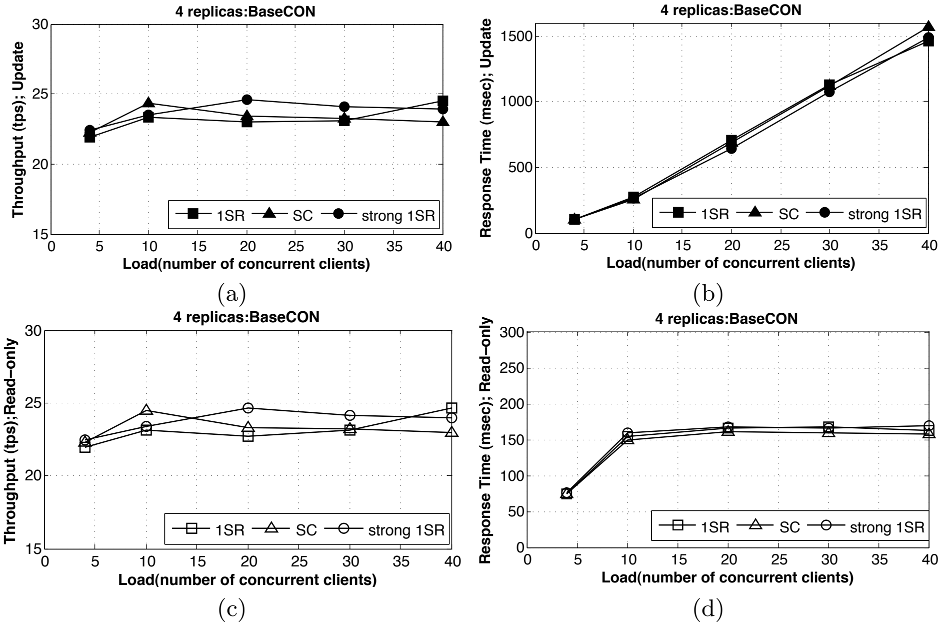


(a) (b) (c) (d)

Fig. 2. BaseCON, TPC-C 20: Throughput and response time

There is no significant performance difference among distinct variants of BaseCON: neither throughput nor response time suffers from stronger correctness requirements. BaseCON scheduler assigns read-only transactions to replicas so that there is no waiting involved: there is always at least one replica where read-only transaction can execute. Thus, the scheduler does not introduce any notable overhead to assign read-only transaction to a specific replica instead of randomly chosen. Response time of update and read-only transactions (Figs. 2(b) and (d)) is the same independently of the correctness criterion considered. This holds for all load conditions and different workloads considered (see also Fig. 3).

Since BaseCON implements conservative transactions execution there are no aborted transactions. On the other hand, due to conservative concurrency

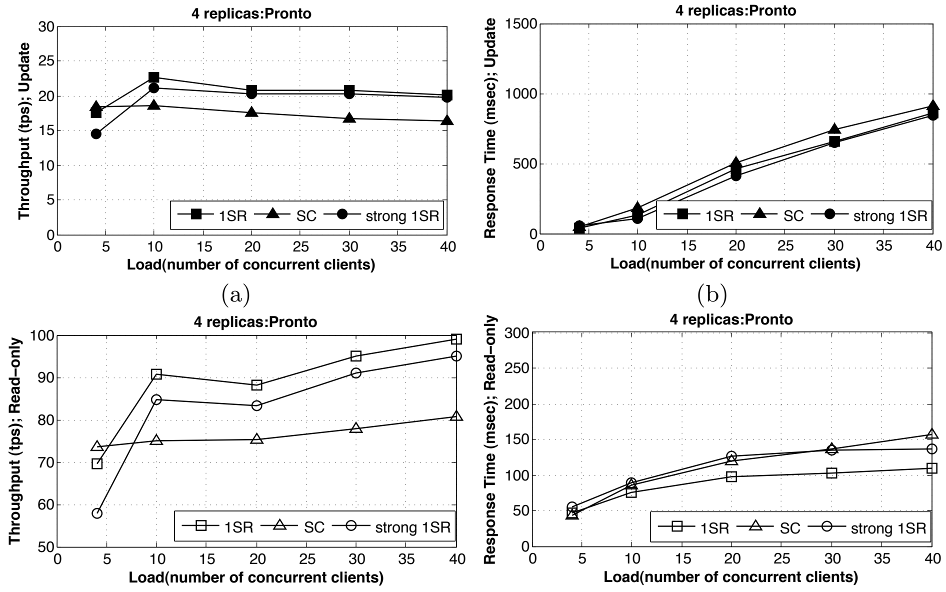


(a) (b) (c) (d)

Fig. 3. BaseCON, TPC-C 50: Throughput and response time

control response time of update transactions grows with increasing load, and thus throughput is not improved. Read-only transactions are significantly simpler and do not require additional synchronization, thus the growth in response time is lower. Differently from update transactions, which are fully executed at all replicas, read-only transactions execute only on a selected replica.

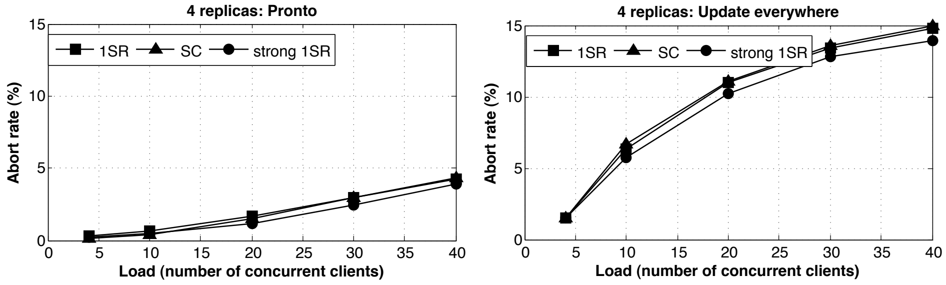
Primary-backup replication. Figure 4 depicts the attained throughput and response time of Pronto running TPC-C 20. The throughput of update transactions is limited by a single primary replica handling most of the update transactions load (backup replicas apply only updates). On the contrary, read-only transactions are distributed over all replicas and consequently higher transactions load results in higher throughput for both 1SR and strong 1SR. However, the performance of Pronto implementing SC is considerably worse. To guarantee SC read-only transactions must execute on a replica which was the first to commit previous transaction of the same client. Since upon delivery of updates the primary replica just performs the certification test and can commit the transaction, while backups still need to apply the received SQL statements, the primary replica is the fastest one to respond. As a result, both read-only and update transactions execute locally at the primary replica overloading it. To guarantee strong 1SR Pronto totally orders read-only transactions with respect to all other transactions but executes them only on selected replicas. In this way the load of read-only transactions is distributed over the replicas. Furthermore, such transactions execute in isolation, as opposite to SC, where read-only



(a) (b)

(c) (d)

Fig. 4. Pronto, TPC-C 20: Throughput and response time

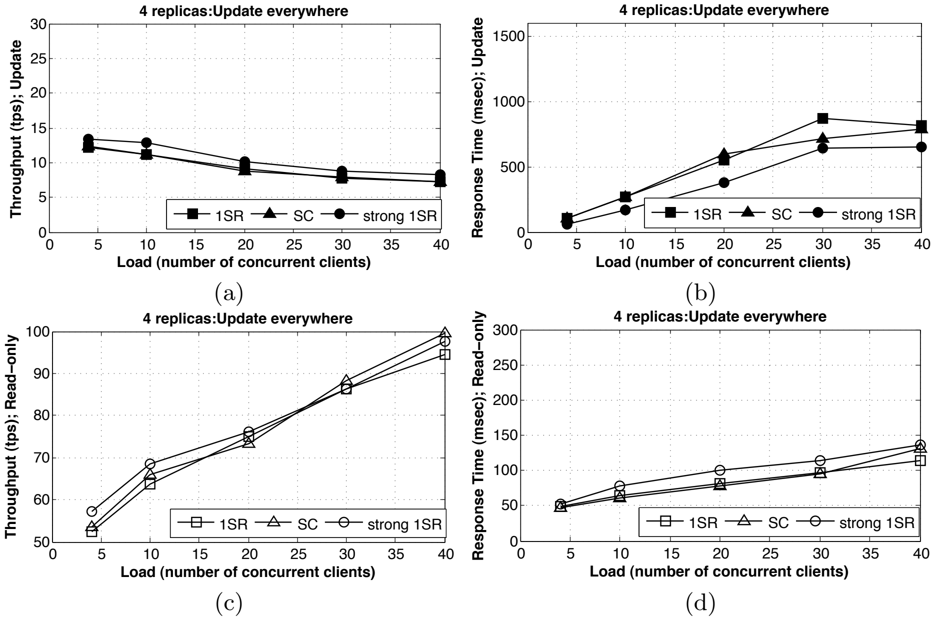


(a) Pronto (b) Update everywhere

Fig. 5. TPC-C 20: Abort rates

transactions execute concurrently with all the load submitted to the primary replica. Further, differently from BaseCON, Pronto aborts some transactions due to local timeouts and deadlocks (Fig. 5(a)).

Update everywhre replication. Figure 6 depicts performance graphs of DBSM running TPC-C 20. As in the case of BaseCON, implementing different correctness criteria with DBSM does not introduce any notable overhead and thus has no significant influence on system's performance. Even though the response time of committed transactions is lower when compared to BaseCON, the improvement is achieved at the price of high abort rate (see Fig. 5(b)). With increasing number of concurrent clients more update transactions will execute in



(a)

(b)

(c)

(d)

Fig. 6. Update everywhere, TPC-C 20: Throughput and response time

parallel without synchronization and consequently more will be aborted by the certification test to ensure strong consistency. Thus the throughput of update transactions degrades leaving more room for executing read-only transactions.

6 Related Work and Final Remarks

In this paper we are interested in the correctness criteria used in middleware-based database replication systems. The majority of the protocols proposed in the literature ensure either 1SR (e.g., [5, 7, 13]) or SI (e.g., [3, 17]). However, some applications may require stronger guarantees. In [10] the authors address the problem of transactions inversions in lazily replicated systems that ensure 1SR by introducing strong session serializability. Strong session serializability prevents transactions reordering within a client's session. Causality expected by the clients is studied in [18] as well. Differently from us, [18] considers only DBSM, and presents two ways (optimistic and conservative) to ensure expected causality. The results reported confirm the results we have attained.

Besides uniform characterization of consistency degrees, we also show how each of them can be achieved in the context of BaseCON and two other replication solutions. BaseCON was originally inspired by conflict-aware scheduling, a replication technique by Amza et al. [19]. The authors use transaction scheduling to design a lazy replication scheme. To avoid aborting conflicting transactions the scheduler is augmented with a sequence numbering scheme and conflict-awareness to provide strong consistency. Differently from BaseCON, failures in

[19] are handled in an ad hoc manner. Moreover, the correctness of the protocol relies on stronger assumptions than ours.

We show experimentally that stronger consistency does not necessarily imply worse performance in the context of middleware-based replication. On the contrary, two of the three protocols evaluated are able to provide different consistency guarantees without penalizing system's performance. Even though the implementation of strong serializability requires ordering read-only transactions in all protocols studied, the overhead introduced by total order primitives is insignificant in middleware-based replication. Moreover, implementation of session consistency in primary-backup protocol exhibits worse performance than strong serializability.

References

1. Cecchet, E., Marguerite, J., Zwaenepoel, W.: C-JDBC: Flexible database clustering middleware. In: Proceedings of ATEC, Freenix track. (June 2004) 9–18
2. Correia, A., Sousa, A., Soares, L., J.Pereira, Moura, F., Oliveira, R.: Group-based replication of on-line transaction processing servers. In: LADC. (October 2005) 245–260
3. Lin, Y., Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R.: Middleware based data replication providing snapshot isolation. In: Proceedings of ACM SIGMOD. (June 2005) 419–430
4. Muñoz-Escoí, F.D., Pla-Civera, J., Ruiz-Fuertes, M.I., Irún-Briz, L., Decker, H., Armendáriz-Iñigo, J.E., de Mendívil, J.R.G.: Managing transaction conflicts in middleware-based database replication architectures. In: Proceedings of IEEE SRDS. (2006)
5. Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: Consistent Database Replication at the Middleware Level. ACM TOCS **23**(4) (2005)
6. Plattner, C., Alonso, G.: Ganymed: scalable replication for transactional web applications. In: Proceedings of Middleware. (October 2004) 155–174
7. Rodrigues, L., Miranda, H., Almeida, R., Martins, J., Vicente, P.: The GlobData fault-tolerant replicated distributed object database. In: Proceedings of EurAsia-ICT. (October 2002) 426–433
8. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
9. Breitbart, Y., Garcia-Molina, H., Silberschatz, A.: Overview of multidatabase transaction management. The VLDB Journal **1**(2) (1992)
10. Daudjee, K., Salem, K.: Lazy database replication with ordering guarantees. In: Proceedings of IEEE ICDE. (2004) 424–435
11. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: Proceedings of ACM SIGMOD international conference on Management of data. (1996)
12. Pedone, F., Frolund, S.: Pronto: A fast failover protocol for off-the-shelf commercial databases. In: Proceedings of IEEE SRDS. (2000) 176–185
13. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Journal of Distributed and Parallel Databases and Technology **14** (2003) 71–98
14. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996)

15. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proceedings of ACM SIGMOD. (1995) 1–10
16. Transaction Proccesing Performance Council (TPC): TPC benchmark C. Standard Specification (2005)
17. Elnikety, S., Zwaenepoel, W., Pedone, F.: Database replication using generalized snapshot isolation. In: Proceedings of IEEE SRDS. (2005) 73–84
18. Oliveira, R., Pereira, J., Correia, A., Archibald, E.: Revisiting 1-copy equivalence in clustered databases. In: Proceedings of ACM SAC. (2006) 728–732
19. Amza, C., Cox, A., Zwaenepoel, W.: Conflict-Aware Scheduling for Dynamic Content Applications. In: Proceedings of USITS. (March 2003)

A Appendix

In the following we show that each variant of BaseCON guarantees its corresponding correctness criteria.

Proposition 1. *The BaseCON-1SR ensures one-copy serializability*

Proof. We show that every history H produced by BaseCON-1SR has an acyclic multiversion serialization graph (MVSG). From [8], if $MVSG(H)$ is acyclic, then H is one-copy serializable. MVSG is a directed graph with the nodes representing committed transactions and edges representing read-from and version order relationships between them. From the protocol, the version order on every data object is defined by the commit order of transactions. If $\ll$ is a version order and T_i and T_j update data object x, then $x_i \ll x_j \Leftrightarrow \mathit{commit}(T_i) < \mathit{commit}(T_j)$.

To prove that $MVSG(H)$ is acyclic we have to show that for every edge $T_i \rightarrow T_j \in MVSG$ there is $\mathit{commit}(T_i) < \mathit{commit}(T_j)$. In the following we consider each edge type of $MVSG(H)$.

– *Read-from edge.* If T_j reads data object x from T_i, then $T_i \rightarrow T_j \in MVSG(H)$. We have to show that $\mathit{commit}(T_i) < \mathit{commit}(T_j)$.
  If both T_i and T_j are update transactions, they will be executed sequentially, and T_j can read the updates of T_i only if $\mathit{commit}(T_i) < \mathit{commit}(T_j)$. If T_j is a read-only transaction, due to 2PL, T_j can read only committed data and so $\mathit{commit}(T_i) < \mathit{commit}(T_j)$.
– *Version order edge.*
  - If both T_i and T_j update x, such that $x_i \ll x_j$, then $T_i \rightarrow T_j \in MVSG(H)$.
    From the definition of version order we have that $x_i \ll x_j \Leftrightarrow \mathit{commit}(T_i) < \mathit{commit}(T_j)$.
  - If T_i reads x_k and both T_k and T_j write x such that $x_k \ll x_j$, then $T_i \rightarrow T_j \in MVSG(H)$.
    Since we assume 2PL and BaseCON executes conflicting transactions sequentually, there are two ways for T_i to read x_k: or (a) T_j updates x and commits before T_k executes, or (b) T_j updates x and commits after T_i executes. If (a) is true, i.e., $\mathit{commit}(T_j) < \mathit{commit}(T_k)$, then $x_j \ll x_k$ contradicting that $x_k \ll x_j$. Thus, it follows that $\mathit{commit}(T_i) < \mathit{commit}(T_j)$.

□

Proposition 2. *The BaseCON-SC ensures session consistency.*

Proof. To prove that every history H produced by BaseCON-SC is session consistent, we have to show that (a) H is 1SR and that (b) for any two transactions T_i and T_j that belong to the same session, if the commit of T_i precedes the submission of T_j in real time, then $\mathit{commit}(T_i) < \mathit{commit}(T_j)$.

a) Similarly to BaseCON-1SR (see Proposition 1) BaseCON-SC ensures one-copy serializability.

b) In the case where both T_i and T_j are update transactions, if the client received a commit for T_i, then all replicas will commit T_i and T_j in the same order (ensured by total-order broadcast primitives): $commit(T_i) < commit(T_j)$.

   If T_j is a read-only transaction, it is scheduled to the replica which has already executed T_i. Thus, $commit(T_i) < commit(T_j)$. Let's consider the case with scheduler failures. The scheduler D_k sends the response of T_i to the client and fails. The new scheduler D_{k+1} receives T_j, but has no information about previous T_i. Since T_j carries the information where T_i has executed, the scheduler forwards transaction to the specified replica. Hence, $commit(T_i) < commit(T_j)$.

□

Proposition 3. *The BaseCON-strong 1SR ensures strong serializability.*

Proof. We show that (a) every history H produced by BaseCON-strong 1SR is 1SR and (b) for any two transactions T_i and T_j, if the commit of T_i precedes the submission of T_j in real time, then $commit(T_i) < commit(T_j)$.

a) Analogously to BaseCON-1SR (see Proposition 1) BaseCON-strong 1SR ensures one-copy serializability.
b) If both T_i and T_j are update transactions and the client receives commit of T_i, then T_j will be ordered after T_i by total-order broadcast and thus, $commit(T_i) < commit(T_j)$. If T_j is a read-only transaction the scheduler forwards it for execution to the replica which has committed T_i already. Let's consider the situation, where the scheduler was suspected incorrectly and based on incomplete state information, it schedules T_j to the replica where T_i has not been executed. When T_j is delivered, the NewScheduler message must have been delivered before and thus, T_j will fail the validation test and must be rescheduled. Thus, $commit(T_i) < commit(T_j)$ is always enforced.

□