Robust Snapshot Replication

Uwe Röhm **Michael J. Cahill** **Alan Fekete** **Hyungsoo Jung**
Seung Woo Baek* **Mathew Rodley***

School of Information Technologies
The University of Sydney,
NSW 2006 Australia
Email: {*firstname.lastname*}@sydney.edu.au
*{sbae4673|mrod5488}@uni.sydney.edu.au

Abstract

An important technique to ensure the scalability and availability of clustered computer systems is data replication. This paper describes a new approach to data replication management called Robust Snapshot Replication. It combines an update anywhere approach (so updates can be evaluated on any replica, spreading their load) with lazy update propagation and snapshot isolation concurrency control. The innovation is how we employ snapshot isolation in the replicas to provide consistency, fail safety, and also to achieve high scalability for both readers and updaters, by a system design without middleware or group communication infrastructure. We implemented our approach using the PostgreSQL database system and conducted an extensive experimental evaluation with a small database cluster of 8 nodes. Our results demonstrate the scalability of our algorithm and its performance benefits as compared to a standard consistent replication system based on synchronous propagation. We also evaluated the costs for adding a new cluster node and the robustness of our approach against node failures. It shows that our approach is at a sweet-spot between scalability, consistency and availability: it offers an almost perfect speed-up and load-balancing for our 8 node cluster, while allowing dynamic extension of a cluster with new nodes, and being robust against any number of replica node failures or a master failure.

1 Introduction

In the dominant multi-tier architecture of modern information systems, replication is a basic component for scalability and for availability. Replication in the stateless parts of the system is easy, but one also needs replication in the database tier, where persistent state is maintained and updated. Therefore, database replication management must deal with change detection, data invalidation, and update propagation, so that all replicas receive all the changes made by updaters. In view of its key importance for obtaining high quality-of-service characteristics, replication management continues as an active research area.

In a seminal paper in 1996, Gray et al identified a fundamental difficulty in replica management: the simple system designs that are convenient and flexible for users (data is consistent at all times because changes are propagated within the update transaction, and updates can be submitted anywhere) scale very badly (Gray et al. 1996). This paper led to a flurry of research activity and diverse system designs.

In terms of the consistency of the replicas, the early work followed two main directions. Some systems took 1-copy serializability as an essential requirement (that is, the system should transparently appear just like an unreplicated database with serializable transactions). Other systems allowed replicas to be temporarily inconsistent, so long as eventually a single state was agreed by all replicas. The former approach did not scale well, while the latter was hard for users who need accurate information.

In 1995, a new isolation level called Snapshot Isolation (SI) was described (Berenson et al. 1995). SI has been provided by a growing number of DBMS platforms. Because it offers good performance, and semantics that are fairly intuitive, this has proved very usable. A large number of recent replication proposals have taken the view that 1-copy SI offers an appropriate level of consistency, that is, the system transparently appears to be like an unreplicated system with SI as its concurrency control. 1-copy SI has been much easier to implement effectively than 1-copy serializability (see the section on related work, below).

1.1 Contributions

In this paper, we offer a new replication management approach called Robust Snapshot Replication. It has a combination of features not found together in other systems, that offer high consistency and scalable performance:

- Executions have the Generalized Snapshot Isolation property (Elnikety et al. 2005) with the consistency properties of an unreplicated system with Snapshot Isolation, except that the snapshot seen by a transaction may not include all updates committed before the transaction started.
- Read transactions never block nor delay updaters.
- The system architecture does not suffer the costs from a middleware layer between the clients and the DBMS engines, nor does it use an expensive group communication infrastructure. Instead we modify the DBMS engines, and we use point-to-point messages.
- Lazy propagation is used, so a user-transaction is implemented as a collection of DBMS-transactions, each of which runs locally, on a single DBMS site. There is no need for expensive two-phase commit.
- Each user-transaction can have its processing done on any site. This allows the load of processing the SQL statements (executing expensive joins etc) to be spread evenly. There is a primary site which has the authoritative version of the data, and the writes of each updating transaction are committed at that site before being propagated; this puts some extra load on the primary, but the load of these writes is much less than executing complete transactions.
- Our approach supports dynamic extensibility of the cluster and it is robust against the failure of multiple replicas or the failure of the primary site.

*Work was done while author was affiliated with the University of Sydney.

We implemented our protocol in PostgreSQL, and we give results from an extensive performance evaluation to show how system parameters influence throughput and response time. We have compared our design to Cybercluster, a production replication solution. In these experiments with varying cluster sizes up to 8 nodes, we show that Robust Snapshot Replication provides a significant better scalability than the synchronous replication protocol.

We also quantify the costs for extending the cluster with a new node and for recovery from a master node failure. We define a metric called the *recovery window* that measures the impact of node failure on response times. During the recovery window, transaction response times degrade as replicas are busy coordinating in order to ensure that all in-flight transactions are correctly propagated. We measure the recovery window of our prototype and show that the effect of a failure on client transactions is small, with a window of less than 100ms for our workload.

The remainder of this paper is organised as follows: In Section 3, we present the Robust Snapshot Replication protocol and explain how it guarantees 1-copy Snapshot Isolation. Section 4 gives an overview of our implementation. The results of our experimental evaluation are presented in Section 5. Section 6 discusses related work and Section 7 concludes.

2 Background

The literature on database replication is vast, and we cannot cite all relevant work. We focus here on a few classic papers, as well as those more recent papers that are closest in approach or properties to ours.

2.1 Database Replication

The early history of the field was centred on eager replication, with strong consistency guarantees. Bernstein & Goodman (1985) provided the definition of 1-copy serializable execution as a correctness condition. Many algorithms was given, using directories, quorums, etc; an exposition of these is found in Bernstein et al. (1987). The seminal paper by Gray et al. (1996) provided a classification of replication approaches, and focused the community on lazy algorithms, by showing how badly eager ones scale. Some suggestions aim to avoid the problems of scaling by placing replication below the DBMS level, using distributed shared memory (Amza et al. 2000).

2.2 Snapshot Isolation

Most related to our paper are the system designs that provide some variant of 1-copy SI as the consistency property. SI itself was defined in Berenson et al. (1995), which also showed that this technique does not always ensure serializability. The key properties of SI are that a transaction reads data, not from the most up-to-date versions of the data, but instead from a snapshot, which captures a consistent committed state of the data *before the transaction started*. Thus T never sees any change made by a concurrent transaction, corresponding to the intuitive notion of being isolated from concurrent transactions. Lost updates are prevented by an explicit "First-Committer-Wins" check, that prevents two concurrent transactions from both committing if they modify the same data item. SI does not guarantee serializability, because in SI concurrent transactions do not see one another, whereas in a serializable execution, one will seem to be first (and not see the other), but the other will seem to be second (and see the effects of the first).

2.3 Generalized SI and Strong Session SI

Daudjee & Salem (2006) and Elnikety et al. (2005) explore the use of Snapshot Isolation in a replicated setting.

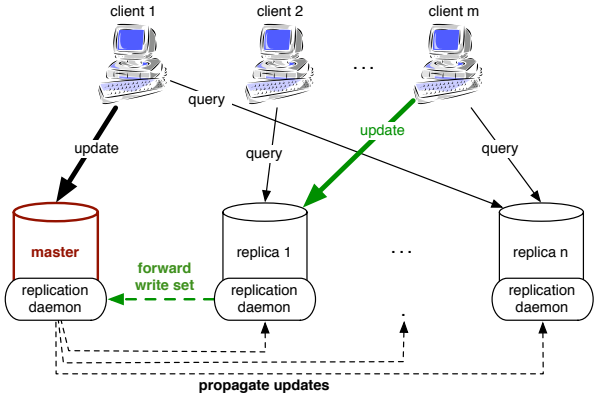


Figure 1: System Architecture.

Both note that centralized systems providing SI generally ensure that transactions see the most up-to-date data committed before the transaction began. In a replicated system, replicas may not have the most recent updates committed locally, so transactions may be assigned a start timestamp that is stale. This is called Generalized SI or Weak SI. A stronger condition, called Strong Session SI, is defined by Daudjee & Salem (2006), that ensures that the snapshot seen by a transaction includes all previous transactions submitted within the same session. We omit the details in the presentation of our system, but their algorithm can be directly applied to our system to provide the same guarantee. Recently, Krikellas et al. (2010) studied how to strengthen a system design based on GSI to provide Strong Session SI, or pure 1-copy SI.

3 Robust Snapshot Replication

Robust Snapshot Replication (RSR) is a lazy, update-anywhere replication protocol that guarantees 1-copy generalised snapshot consistency among all nodes. To do so, we take advantage of the core characteristic of snapshot isolation concurrency control: Readers never block, even if concurrent updates are executed, because transactions are running isolated on their own database snapshot. This snapshot is the state of the database at the point in time when a transaction starts. And the master node keeps several snapshot versions to be able to determine the correct execution order of update transactions.

The essential idea behind Robust Snapshot Replication is to execute the code of an updating transaction locally at some replica, in order to compute a write-set (a list of what items are modified, and what values they are set to). The replica also determines an effective start-timestamp for the transaction, namely the one used by the replica when doing the local computation. The local transaction at the replica is then *aborted*, and the write-set and effective start-time are sent to the primary. At the primary, the write-set is installed in a local transaction whose snapshot is not the beginning of the local transaction (as usual in SI); instead the installation transaction uses a snapshot equal to the effective start-time sent from the replica. This depends on a form of "time-travel", where a DBMS can run a local transaction in the past; this feature is present in some SI-based systems, and can be added easily to others. Once the installation transaction has committed, its changes are lazily propagated to all replicas (including the one where the computation was first performed). A read-only transaction is done at one arbitrary replica.

3.1 System Architecture

Figure 1 gives an overview of the system model behind snapshot replication. We are assuming a set of nodes with snapshot isolation as concurrency control over which some data is replicated, such as a database cluster. One

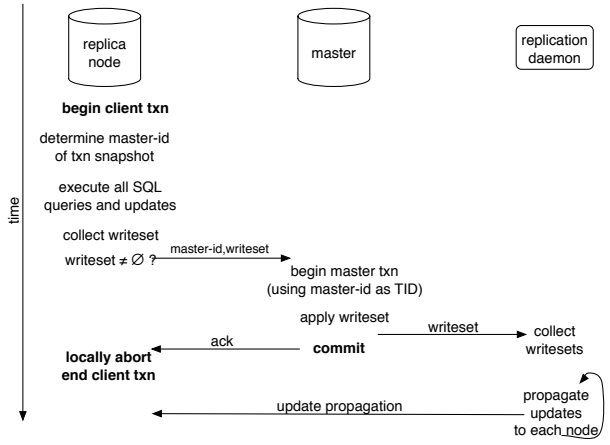


Figure 2: Update Transaction Execution under RSR.

is the dedicated **master** node which holds the primary copies of the replicated data, while the remaining nodes store **replicas** (we sometimes shortly refer to those nodes as replicas too).

For clients the system looks and behaves like 'update anywhere': all cluster nodes are the same and all nodes can execute both read and update transactions. Clients are free to send update transactions or query transactions directly to any node; alternatively, a load balancer could spread incoming requests among all available nodes. Clients can also connect directly to the master node.

Internally, all updates are calculated at some replica, but the apparent execution order is determined at the master node, because each replica forwards the write sets of any update transaction to the master which then decides whether these *remote transactions* can commit or not. However, installing the updates at the master is in a different DBMS transaction from the original computation at the replica (which is actually aborted after the calculation has been done). Thus each DBMS transaction is purely local to one site, and two-phase commit is never needed. Whenever an update transaction commits on the master, the master forwards the write set to the **replication daemon**, which then asynchronously propagates the updates to all cluster nodes with individual *propagation transactions*. Note that our system architecture requires neither coordination middleware, nor group communication infrastructure.

3.2 Time Travel Transactions

The core of every replication protocol is its handling of update transactions. The main idea of Robust Snapshot Replication is that each node executes its client transactions locally, including any query or update statements, but that write sets are forwarded to the master which decides on the authoritative schedule of the update transactions. Note that read-only transactions can always commit directly because we are assuming that each node provides local snapshot isolation. Figure 2 visualises the steps for executing an update transaction under RSR.

Each node keeps track of its database version in terms of the last master transaction ID (TID) that has been propagated to it. If a client starts an update transaction on some node of the cluster, the transaction gets tagged with the logical timestamp of the current snapshot, which is represented by the effective master TID. The transaction then executes normally on the node, performing all queries and updates locally. When a transaction wants to complete, the node first determines the transaction's write set, i.e. which rows have been modified by the transaction (we are typically building the write set up during the transaction execution – see the Implementation section for details). This write set is then forwarded to the master together with the timestamp of the snapshot on which the transaction was locally executing.

An important feature is that snapshot replication does not forward the whole update transaction to the master node, but rather just its write set. The benefit is that complicated processing (joins, queries and sub-queries) needed as part of an updating transaction are only executed locally on the replica. This is why we call our technique *update anywhere*. We will discuss this in more detail in Section 4.

The master then determines whether the client transaction is allowed to commit. The core idea of Robust Snapshot Replication is to get control over the snapshot version at the master. We developed a mechanism to start an update transaction "in the past" at the correct point of time with regard to the global execution order rather than the current wall clock. This is similar to the notion of "time travel" as discussed by Lomet et al. (2006), but explicitly for executing the write sets of update transactions at the master node. The master applies the replica's write set into its own database under the master transaction ID that has been send by the replica. This way, the updates are applied to the correct snapshot version 'in the past' of the master database which will be an older version of its data then the up-to-date state. If no conflict is detected by the master's "First Committer Wins" mechanism, then the client transaction can successfully commit. Otherwise, the client transaction is aborted.

The actual commit of an update transaction consists of three steps. First, the master sends the write set to the replication daemon which propagates it throughout the cluster asynchronously. Second, the master node commits. Third, and most noteworthy, the replica node aborts its own local transaction, but acknowledges the client that its transaction has correctly committed.

The reason for this last step is that we try to minimise the update costs of Robust Snapshot Replication. By aborting their local update transactions, replica nodes avoid all disk writes; all logging of update transaction is done at the master. The updates are then applied to the replica later after the end of the client transaction via update propagation transactions send from the replication daemon.

Algorithm 1 summarises the additions which Robust Snapshot Replication is introducing into the start of a transaction and the transaction commit handling.

Algorithm 1 General Transaction Handling under RSR

```
1:  ON BEGIN TRANSACTION t:
2:  version(t) ← latest master_version
3:  if node is replica and t is update propagation then
4:      disable write set collection for t
5:  end if
6:  ON COMMIT TRANSACTION t:
7:  if writeset(t) = ∅ then
8:      commit(t) {read-only transactions just commit}
9:  else
10:     if node is not master then
11:         forward_to_master(writeset(t))
12:         {wait for ACK from master}
13:         abort(t) {note: this is a local abort}
14:     else
15:         notify_replicationdaemon(writeset(t))
16:         commit(t)
17:     end if
18: end if
19: ON ABORT TRANSACTION t:
20: abort(t) {all local, no other node involved}
```

3.3 Update Propagation

Our protocol uses lazy update propagation: updates are propagated from the master node throughout the cluster after the installation of the writeset has committed on the master. Although we follow an update anywhere approach, this does not affect the 1-copy snapshot isolation property because all updates are checked for conflicts on the master. Because each cluster node is using snapshot isolation as concurrency control mechanism, we get an elegant way to synchronise refresh transactions and local client transactions for free: With snapshot isolation, readers never block and hence read transactions do not have to wait while our protocol is refreshing a cluster node.

Update transactions are also not a problem because they locally always abort (we only commit on the master node) so that there will never be the case that a cluster node has to apply the 'first-committer-wins' rule on update versus refresh transactions. The only transactions that actually change the database on each cluster node are refresh transactions that propagate updates from the master.

3.4 Guaranteeing 1-copy Snapshot Isolation

Robust Snapshot Replication not only can provide high scalability and performance, it also guarantees 1-copy Snapshot Isolation. In the following, we want to give a sketch of proof on how this consistency guarantee is achieved. Firstly, each node is running under snapshot isolation so that read-only transactions are guaranteed to access a consistent snapshot of the data. This might not be the latest snapshot as available at the master node at the real time when the transaction is started, though, because RSR uses asynchronous update propagation. Thus we offer 1-copy *Generalised* SI.

Furthermore, all updates are executed at the master node on their corresponding snapshot version by using a time-travelling mechanism. This allows the master to establish the authoritative schedule of all update transactions. Updates which are inconsistent with the up-to-date version of the master will be aborted by the master's First-Committer-Wins SI mechanism.

Finally, the update transactions are propagated throughout the cluster from the master in the global execution order. Because the replica nodes locally abort all client update transactions, this is the only way how the snapshots in the cluster can change. At each point in time, every replica node reflects a consistent snapshot of the master node. Without new updates, all nodes will eventually agree on the same snapshot of the database, namely the up-to-date state of the master node.

3.5 Adding a node to the cluster

Cecchet et al. (2008) in their analysis of database replication protocols, raised the concern that research proposals for database replication do not often address concerns such as adding nodes to the cluster. They argue that such an operation should have no effect on the normal operation of the cluster, contrasting this with current practice that requires the whole system to be shut down to add a new replica.

We propose a solution similar to the Total Copy Strategy (TCS) that Liang & Kemme (2008) present but with a number of optimisations and differences for its application with our lazy protocol. We assume that the node joining the cluster has no previous knowledge of the cluster or cluster configuration and consists of just an empty database. As the nodes join the cluster, they begin buffering write set propagations and request a snapshot of the master version at a time just prior to when they joined the cluster. Once a node has received this snapshot, it is applied, followed by the buffered propagations up until the point it is synchronized with the cluster. This approach varies from the TCS approach, as it allows the joining node to buffer propagations, instead of requiring its recovery peer to do so.

3.6 Fail Safety

We distinguish two cases of fail safety: Protection against replica failure and protection against the failure of the master node. The former is the easier case, because the cluster is still functional as long as the master node is available. When a recovered replica wants to re-join the cluster, we handle it with a similar algorithm as extending the cluster by one more node (cf. previous Section 3.5).

The second failure scenario is more severe. The single master node introduces a single point of failure which we must remove to improve the robustness of the protocol. We propose an optimistic solution to fail safety that rebuilds on demand the information lost when the master fails. The basic idea is to keep local (transient) state information at each replica that allows reconstruction of the master's propagation buffer in the (rare) case of a master node failure. In case of the master node failing, our recovery algorithm enables the system to re-elect a new master, cope with rebuilding the lost global state information, and switching the systems master node before resuming normal operation.

Our approach requires retaining some additional information so the system is able to rebuild the master's propagation buffer using information distributed across the replicas:

- Write Set Identity $writeset\_id(t)$ – A unique identity for a write set which also identifies the node from which the write set of transaction t originated.
- Limbo Buffer $limbo[t]$ – Each replication daemon keeps the write sets of update transactions in a local transient limbo buffer until the update propagation is received back from the master. Write sets are only removed from this buffer when either an explicit abort response is received from the master or the write set has been propagated back by the master. The limbo buffer also keeps track of the communication state between master and replica: Each write set for a transaction t has an associated flag ($limbo[t].master\_committed$) that keeps track whether a commit acknowledgment has been received from the master.
- Master to Local Version Mapping $ledger$ – A function that maintains a mapping from a master snapshot to a corresponding local snapshot. For example, if we had for some master snapshot a transaction t, that executed at the master, then $local\_version(master\_version(t))$ would return the latest local version which includes the effects of transaction t.
- Propagation Replay Buffer – A buffer which retains the last *threshold* propagations at each replica.

3.6.1 Normal Transaction Handling

We have to maintain the update propagation state at each node. To do so, we have to extend the normal transaction handling under Robust Snapshot Replication, as shown previously in Algorithm 1, as follows:

When we receive an update propagation transaction, we check whether the corresponding master commit acknowledgement got lost in between and update the limbo buffer's master-committed flag accordingly (Algorithm 2, lines 5–8).

We further have to extend the Abort and the Commit handling. The Abort process remains largely unmodified except that the master now explicitly acknowledges the write set by its $writeset\_id(t)$, along with an ABORT indication.

Algorithm 2 Begin-Transaction Handling in RSR

```
1: ON BEGIN TRANSACTION t:
2: version(t) ← latest master_version
3: if node is replica and t is update propagation then
4:    disable write set collection for t
5:    if limbo[t].master_committed = false then
6:       send_ack(writeset_id(t), COMMIT)
7:       limbo[t].master_committed ← true
8:    end if
9: end if
```

Algorithm 3 Abort Handling with RSR in Detail

```
1: ON ABORT TRANSACTION t
2: if node is master and t is from a replica then
3:    send_ack(writeset_id(t), ABORT)
4:    abort(t)
5: end if
```

Most state maintenance is done during commit handling: On commit of a client transaction, we keep a copy of its write set in the node's limbo buffer until the write set is propagated back from the master (or the master tells us that he had to abort). When an update propagation commits, then the propagated write set is removed from the limbo buffer and a new mapping of the current local replica version to the master version of that write set is added to the ledger.

Algorithm 4 Commit Handling with RSR in Detail

```
1: ON COMMIT TRANSACTION t
2: if writeset(t) = ∅ then
3:    commit(t) {read-only transactions still just commit}
4: else if node is replica then
5:    if t is an update propagation then
6:       limbo[t] ← ∅
7:       Add (master_version(t), local_version(t)) to ledger
8:       commit(t)
9:    else
10:      limbo[t] ← writeset(t)
11:      forward_to_master(writeset(t))
12:      response ← recv_ack(writeset_id(t))
13:      if response is ABORT then
14:         limbo ← limbo − writeset(t)
15:      else
16:         limbo[t].master_committed ← true
17:      end if
18:      abort(t)
19:   end if
20: else if node is master then
21:   Add writeset(t) to propagation_buffer
22:   send_ack(writeset_id(t), COMMIT)
23:   commit(t)
24: end if
```

3.6.2 Master-Failure Recovery Algorithm

We have seen how RSR keeps a copy of the master's propagation state distributed around the cluster. Algorithm 5 shows the recovery algorithm after a master failure that relies on this information. It consists of three components: The first part is the election of a new master, followed by the recovery process required to bring that failed master to the required state, including all write sets up to the last acknowledged write set.

These extensions make RSR robust against any number of replica failures and against a single master failure.

There are two limitations: Firstly, the discussed recovery algorithm does not tolerate the failure of the master node and one replica at the same time. Luckily, the master recovery algorithm is typically very fast as we will see in the evaluation Section 5.6, keeping this time period short. Secondly, if we allow clients to send update transactions directly to the master node, then we also need to introduce a master's peer node that keeps a limbo buffer of client transactions committing directly on the master. This is a straightforward extension to the commit handling on the master, whose description we had to omit here due to lack of space.

Algorithm 5 Master-Failure Recovery Algorithm

```
1: ON NODE DETECTING MASTER FAILURE:
2: self ← {master_version, identifier, commit_order}
3: nodelist ← self
4: send_all(NODE_STATUS, self)
5: for each remaining node do
6:    receive(NODE_STATUS, info)
7:    nodelist ← nodelist + info
8: end for
9: elected ← node in nodelist with latest master_version {ties
   broken by lexicographical ordering of node id}
10: ws_buffer ←most recent commit_order in nodelist
11: if own identifier = elected then
12:    become_master()
13:    master_transition(nodelist, ws_buffer)
14: else
15:    update_master(elected)
16:    replica_transition(ws_buffer)
17: end if
```

Algorithm 6 New Master Algorithm

```
1: BECOME_MASTER()
2: resume_version ← master_version
3: min_version ← oldest master_version in nodelist\self
4: for each wset ∈ replay_buffer order by version do
5:    send_all(WS_REPLAY, wset)
6: end for
7: recovery_list ← ∅
8: for each wset ∈ limbo_list ∩ ws_buffer do
9:    recovery_list ← recovery_list + wset
10: end for
11: for each remaining node N_r do
12:    receive(N_r, LIMBO_REPLAY, list)
13:    for each wset ∈ list do
14:       recovery_list ← recovery_list + wset
15:    end for
16: end for
17: for each wset ∈ recovery_list order by ws_buffer do
18:    local_version ← local_version(master_version(wset))
19:    apply_ws(wset, local_version)
20: end for
21: send_all(RESUME, resume_version)
```

```
1: REPLICA_TRANSITION()
2: replay_version ← −1
3: drop_out ← false
4: while replay_version < elected.master_version do
5:    receive(WS_REPLAY, writeset)
6:    if replay_version = −1 ∧ writeset.master_version >
      self.master_version then
7:       drop_out ← true
8:       break loop
9:    end if
10:   replay_version ← version(writeset)
11:   if version(writeset) > master_version then
12:      apply_ws(writeset)
13:   end if
14: end while
15: list ← ∅
16: for each wset ∈ limbo_list ∩ ws_buffer do
17:    list ← list + wset
18: end for
19: send(elected.identifier, LIMBO_REPLAY, list)
20: receive(RESUME, new_master_version)
21: if drop_out = true then
22:    Exit the cluster and rejoin
23: end if
24: master_version ← new_master_version
```

3.7 Discussion

There are several benefits of our approach as compared to known replication protocols. Robust Snapshot Replication is an interesting combination of update anywhere and primary copy replication: Clients' transactions, both read-only and update transactions, can have their processing workload placed locally on any cluster node. However, the master node is where the updates are actually installed first. The master is mainly checking for conflicts between the write sets. In particular it does not need to execute any joins, any queries, or the query part of a potentially complex update. The update propagation throughout the cluster is done asynchronously through point-to-point communication between two nodes for each propagation transaction.

RSR as described above provides 1-copy Generalized SI, but not Strong Session SI. Strong Session SI could however be added relatively easily: a replica would need to delay the start of a client transaction until previous updates of the same session have been propagated back to the replica. The limbo buffer at each node contains all necessary information to implement this, including any outstanding updates from previous actions in the same session.

4 Implementation

We built a prototype implementation of Robust Snapshot Replication in the PostgreSQL open source database engine. We chose PostgreSQL because it provides a multi-version storage management that keeps old versions of updated records around until an explicit VACUUM command is run (this is typically done periodically by the system).

Our implementation consists of three components:

1. Write-set extraction,
2. Time-travel for remote update transactions, and
3. A replication daemon

4.1 Write-set Extraction

There are three principle ways to extract write sets of update transactions from a replica:

- *Middleware-based* approaches gather write sets in form of the original DML statements at the SQL level in a middleware component outside the database.
- *Trigger-based* approaches collect the after-images of changed rows inside the database engines as part of the transaction execution.
- *Log sniffers* extract the physical write sets from the DBMS' transaction log.

Depending on the approach, the write set is either a set of SQL statements, a set of updated tuples, or a set of transaction log records.

A middleware-based approach has the advantage that one does not need to change the existing database schema or the DBMS itself. However, it also has some major disadvantages: Firstly, its additional layer between clients and the database introduces some overhead. Secondly, it captures whole SQL statements which have to be sent, parsed and executed on the master[1]. This means that the master has to also evaluate any joins, any filtering conditions or sub-queries which are part of those SQL update transactions. This can concentrate a heavy workload on the master. Even worse, if clients invoke a server-side stored procedure, the middleware can only capture the CALL statement but not the individual SQL statements inside the stored procedure, so that the master will have to execute the whole stored procedure code again. As many applications implement transactions as stored procedures[2], this is a major restriction of a middleware-based approach.

In contrast, a log sniffing approach is very low-level and totally specific for one database engine. A log sniffer needs to know the exact structure of the transaction log of the monitored system, and although the captured write sets are very generic (e.g. there is no problem to support stored procedures), there is no straightforward way of applying the captured log records at the master node. One either has to transform them back into simple SQL statements and feed those into the master's query processor, or one must modify the transaction log code which is one of the most system specific components of every database engine.

For these reasons, our implementation uses a trigger-based write-set extraction mechanism. We extended the database schema with row-level triggers which are fired on every insert, delete and update of a replicated table and which then extract the write set of the update transaction. In more detail, we are using external triggers written in C which are dynamically linked at runtime as a shared library into PostgreSQL's execution process. These replication triggers transform the after-images of the modified tuples into simple INSERT, DELETE or UPDATE statements for the master, and append them to the transaction's write-set buffer.

Note that those resulting DML statements are qualified by *primary key*: Even if the update transaction on the replica calculated a filter condition, the write set will only consist of simple update statements for each modified tuple that is identified by its primary key. For example, if an update transaction includes the following statement:

```
UPDATE customer SET phone='999'
WHERE name='Clouseau'
```

then its write set will include a simple:

```
UPDATE customer SET phone='999'
WHERE cid=ID
```

This means that the master will not have to evaluate the original, potentially complex filtering condition again. It also supports stored procedures very naturally, as triggers are fired for each individual DML statement inside a stored procedure.

There is one disadvantage in that the write set can potentially become quite large. For example, imagine an unqualified update statement such as

```
UPDATE employee SET salary=1.1*salary
```

which would modify each tuple of a relation. In this case, the triggers could use a temporary table in the database engine to build the write set (as e.g. done by the Slony-I Replication System (2011)). However, such large update sets occur seldom in practice, and also note that a log sniffing approach would face the same challenge. Applications could be modified to send such updates directly to the master in order to avoid some of the associated communication overhead.

4.2 Time-travelling for Update Transactions

When an update transaction[3] locally completes on a cluster node, the replication triggers collect the write set in the

[1] There are some hybrid approaches where the middleware uses a 'hook' in the database engine to retrieve the write set — depending on the hook, we would classify those as log sniffer or trigger-based.

[2] Besides some performance benefits, stored procedures are helpful to prevent SQL injection attacks.

[3] We can easily separate between update and read-only transactions using the write set buffer: If it is non-empty, it was an update transaction which has to be forwarded to the master, otherwise it was a read-only transaction that can directly commit locally.

transaction's write set buffer; this buffer is then forwarded to the master node as a remote update transaction together with the corresponding master transaction ID of the transaction's snapshot. We call this the *snapshot ID* in the following discussion. The master sends the snapshot ID with each update propagation (details follow below), and each node stores as its local snapshot ID the master transaction ID of the latest updates it has received. We extended the internal transaction context of PostgreSQL so that each local transaction not only has a write set buffer, but also is tagged with the corresponding snapshot ID.

We extended PostgreSQL to allow transactions to explicitly specify their snapshot ID. To do so, we introduced two new SQL keywords to change snapshot IDs easily:

- `SHOW TRANSACTION START` shows the current snapshot ID. It is mainly used for debugging.
- `SET TRANSACTION START [NUMBER]` changes the current snapshot ID to a specific value instead of the latest transaction ID seen from the master.

Remote transactions use this special command to explicitly specify their snapshot ID. This (possibly older) snapshot ID specifies the corresponding master transaction ID at which remote transactions were executed: instead of reading the up-to-date data versions of data, these 'time travel' remote update transactions read data of some earlier snapshot as specified by their master ID (as long at the ID is not too low so that the corresponding tuples have been freed by a VACUUM operation). The master simply executes the remote transaction with the specified master ID. All other query execution code of PostgreSQL remains unchanged.

In order to support our approach to dynamic scalability, we also extended PGDump with our SET TRANSACTION START command to provide a dump at a specific snapshot. The output of PGDump contains both the schema (including stored procedures and replication triggers) and the data. This approach allows a new node to start out completely empty, with just a blank database and no configuration.

4.3 Replication Daemon

After an update transaction committed on the master, its write set has to be propagated throughout the cluster. The master captures its write sets with the same trigger-based approach that is used by its replicas[4]. We implemented the update propagation as a separate daemon process running on the master The snapshot replication daemon uses the standard `libpq` library of PostgreSQL to propagate the write set of committed master transactions to each node with a separate propagation transaction. Because it is asynchronous replication, the master's update transaction commits after its write set was received by the replication daemon; the replication daemon then propagates each write set independently per node throughout the cluster.

The cluster nodes use the same back-end extensions as the master node to distinguish between normal client transactions and propagated update transactions: The replication daemon propagates the updates together with the latest transaction ID from the master using a special prefix on its commands. The PostgreSQL back-end on a cluster nodes stores this latest master transaction ID, and executes and commits the updates locally. From then on, new client transactions will run on the new snapshot and, if they are update transactions, they will use the corresponding master ID when forwarding their write set back to the master.

[4] As an optimisation, one could introduce a short-cut for remote transactions by using the received write set also for update propagation instead of capturing it again via the replication triggers.

Because each cluster node is using snapshot isolation concurrency control, no further synchronisation between local client transactions and the propagated update transaction is needed. In particular, update propagation transactions will never conflict with local update transactions because the latter are always locally aborted after their write set has been send to the master. In other words, each cluster node is only updated via update propagation transactions from the master.

5 Evaluation

The following section presents the results of an experimental evaluation of Robust Snapshot Replication.

5.1 Experimental Setup

All experiments have been conducted on a database server cluster with eight nodes, each with a 3.2GHz Intel Pentium IV CPU and 2GB RAM under RedHat Enterprise Linux version 4 (Linux kernel 2.6.9-11), and interconnected with a gigabit Ethernet network. Each node was running a PostgreSQL 8.2.4 instance with our snapshot replication extensions. We have chosen one of these node as the master node for snapshot replication.

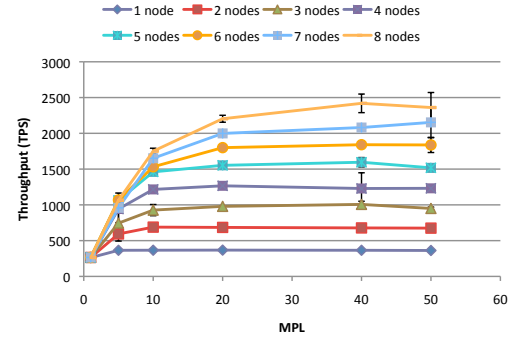
We use the OSDL-developed DBT-1 benchmark (v2.1) client as evaluation scenario, which is a simplified, open-source version of the TPC-W benchmark (Open Source Development Lab 2004, TPC 2002). This benchmark consists of a variety of 14 different transactions which are executed in a probabilistic sequence that models a typical e-business scenario. We used the 'Shopping Mix' of the DBT-1 benchmark which is a fairly update intensive workload consisting of 80% read-only browsing transactions and 20% update transactions. The DBT-1 database consists of 8 tables. We generated a test database with a scaling factor of 1000 items and 40 clients (about 700 MB of data). For each measurement, we ensured that the system was in its stable state, i.e. that the throughput fluctuated by at most 1% during the measuring interval: We first allowed a 1 minute ramp-up time for all clients, before we measured the mean response times, the throughput and the abort rates of the DBT-1 transactions for an interval of 5 minutes.

Because we are interested in evaluating the database tier, we configured DBT-1 to directly access the databases without application server in between. The client emulations were run on eight dedicated Linux machines (3.2 GHz Pentium IV CPU, 1 GB memory, RedHat 4.1.2-12) that were connected via a local 100MBit Ethernet to the database cluster. As the DBT-1 benchmark is not designed for load-balancing over a database cluster, we started a dedicated 'dbdriver' on each client for one of the database cluster nodes and spread the load uniformly over all available client emulators. For example, if we are measuring a load of 40 concurrent clients (MPL 40) against a four node cluster, four client emulators were used, each emulating 10 independent client connections ($4 \times 10 = 40$).

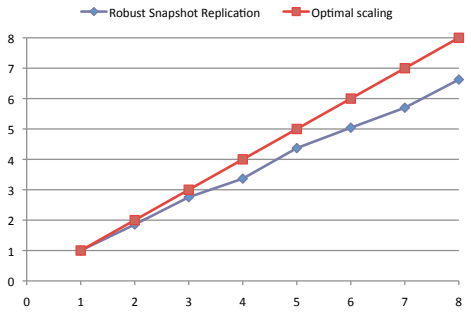
5.2 Snapshot Replication Overhead

First, we want to quantify the performance overhead induced by our update anywhere snapshot replication protocol. In this test, only a single cluster node is involved that is put under increasing load. We are varying the multiprogramming level from 1 to 50, with all clients trying to execute transactions as fast as possible without any think time in between.

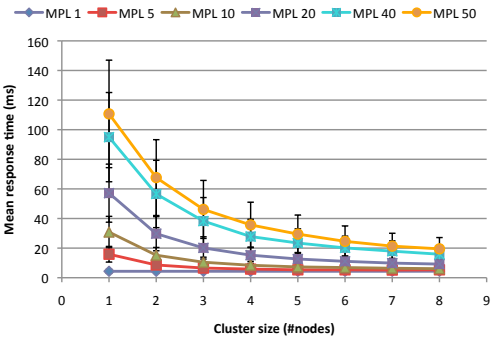
Figure 4 shows the results. We are plotting the throughput achieved on a single node with varying MPL on a plain PostgreSQL 8.2.4 instance (blue line), and with PostgreSQL 8.2.4 with our snapshot replication handling compiled in. Both curves are almost indistinguishable, which means that the additional overhead of snapshot



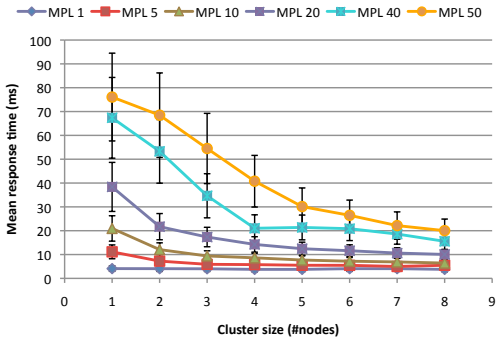
(a) Throughput over MPL.



(b) Scaleup at MPL=40.



(c) MRT (Browsing) over Cluster Size.



(d) MRT (Ordering) over Cluster Size.

Figure 3: Scalability of Robust Snapshot Replication.

replication for capturing the write sets of the update transactions is negligible. On average, it reduces the throughput by only 5% to 6%. The reason for the reduction is that even on this one node 'cluster', RSR does execute the triggers to capture the write sets and does send them at commit to the replication daemon. However, the results clearly show that our C-based trigger implementation imposes only a very small overhead.

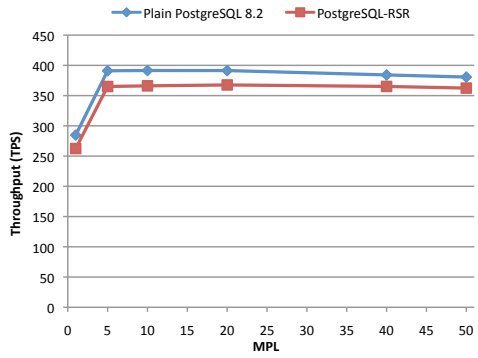


Figure 4: Overhead of RSR on single node.

5.3 Scalability Evaluation

Next, we are interested in the overall scalability of our approach with increasing cluster size. As in the previous experiment, we are varying the multiprogramming level from 1 to 50, with all clients executing their transactions without any think time in between. This gives the maximum load on the system. We measured the throughput and the mean response times with varying cluster sizes up to eight nodes.

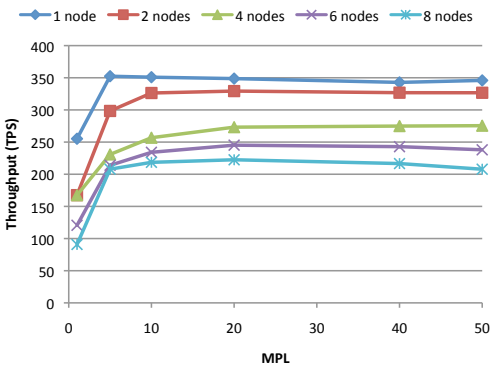
Figure 3(a) shows the sustained throughputs per multiprogramming level with different cluster sizes. All curves start from the same basic throughput for MPL 1 (just one client, single threaded) and they show an increasing throughput while the system is more and more loaded with increasing multi-programming level, until they reach a (stable) plateau when the system saturates. Note that with increasing cluster sizes, the system needs a higher MPL to saturate (the gradients of curves for larger clusters are lower) and it can reach increasingly higher throughputs. The plateaus of each curve means that update anywhere snapshot replication can sustain its maximum throughput even after the system has saturated.

In Figure 3(b), we visualised the scale-up of our RSR system for MPL 40. Snapshot replication shows almost perfectly linear scalability up to the full cluster size of eight nodes, only slightly lower than the ideal linear function with slope 1. An eight node cluster is about 6.6 times faster than a single node.

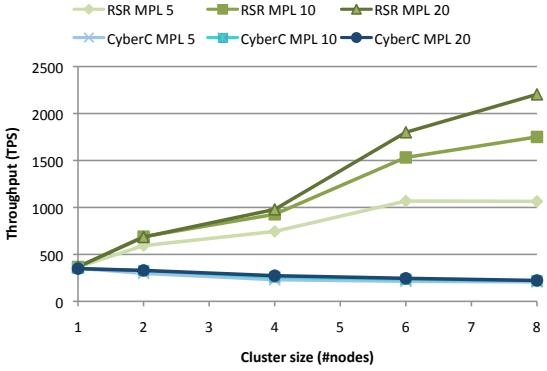
Larger clusters give clients faster response times, both for read-only transactions (Figure 3(c)) and also for update transactions (Figure 3(d)). In those figures, we plotted the mean response times for the browsing (read-only) part of the TPC-W workload and for the ordering part with the different cluster sizes. Larger clusters allow us to spread the workload better among several nodes, so that the response times for individual transactions go down (less waiting times in the request queue). This is a benefit of the point-to-point character of snapshot replication: No update transaction has to block the whole cluster in order to establish a global order or propagate to all replicas. And because each node is running under snapshot isolation, read-only transaction do not block anyway. At the same time, the abort rates were very low across all workloads, in the range of 0.01% and 0.02% only.

5.4 Comparison with Eager Update-Anywhere Replication

Next, we want to compare the scalability of our approach with a state-of-the-art production replication approach. A company called Cybertec released a synchronous, update anywhere ('multi-master') replication system for PostgreSQL 8.2, called Cybercluster, that provides 1-copy serialisability (Cybercluster Project 2007). Cybercluster is based on the older PGCluster replication system code (PGCluster Project 2009) and is made available as open source project. We downloaded and installed this system on our evaluation environment; then we repeated our



(a) Throughput of Synchronous Replication.



(b) Comparison with Robust Snapshot Replication.

Figure 5: Robust Snapshot Replication versus Synchronous Replication.

scalability experiments with the same settings as in Section 5.3 using the Cybercluster system.

Synchronous (eager) replication propagates updates within the same transaction context among all cluster nodes, and for this, it requires a distributed, two-phase commit protocol at the end. This results in a far worse scalability behaviour than with snapshot replication: As we can see when we compare the different curves in Figure 5(b), the peak throughput with synchronous replication actually decreases on larger clusters. The reason is the increasing synchronisation effort for propagating updates.

Our RSR system scales much better than Cybercluster with increasing workload and cluster size. In Figure 5(b), we plotted the throughput for different multiprogramming levels (MPL 5, 10 and 20) on varying cluster sizes from 1 to 8 nodes with both approaches. The lower three curves (in blue colors) are with the synchronous replication system, showing the slight decrease of throughput over increasing cluster size. In contrast, the upper three (greenish) curves are with lazy RSR, which provides increasingly faster throughput with larger clusters. On an eight node cluster and with MPL 20, RSR is an order of magnitude faster than synchronous replication (factor 10).

5.5 Dynamic Scalability Performance

Much research on database replication focuses its evaluation almost exclusively on performance, scalability and consistency while failing to evaluate activities such as adding a new node to a database cluster or how to handle a failure case (Cecchet et al. 2008). The following experiments attempt to validate specifically the ability of our approach to extend the cluster by adding a node and to handle the failure of the master node (Sec. 5.6).

Figure 6 shows the time taken for the joining operation to complete on a seven node cluster while adding an eighth node against varied throughput. We ran the DBT-1 benchmark while performing this operation with a small database of approximately 62 MB[5]. We vary the throughput by modifying the number of clients and their think time that are connected to the system. For example, to produce a throughput of 600 TPS we can emulate 300 clients with a think time of 0.5 seconds per client. While doing this we note that our actual observed throughput is close to our desired throughput, indicating that the joining process has little observable effect on the throughput of the cluster while a node is joining.

From Figure 6 we notice that the total time increases in a super-linear fashion. In order to explain this, we break the node joining operation down into its individual stages. The first stage is to request a snapshot from a recovery peer. We see the time taken for this stage increases as the load on the system increases and this is expected. As the load on the system increases there is more contention for

[5]We use a small database because our buffer space for propagated write sets is, at present, limited to main memory in our implementation.

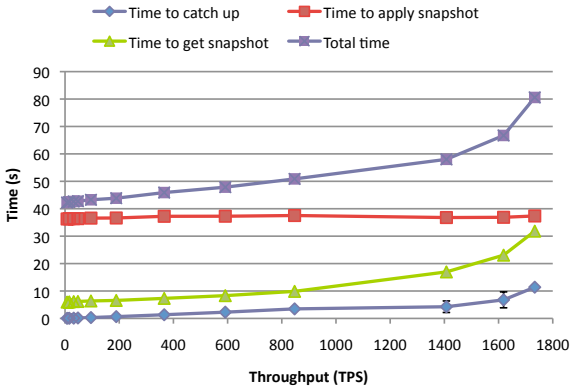


Figure 6: Dynamic Scalability – Time Taken vs Load

access to disk and other resources of the DB system on the recovery peer. This causes the 'Get Snapshot' stage to take longer as load increases. After this stage is complete the joining node applies the snapshot to an empty database, marked as the 'Apply Snapshot' stage. This takes consistently the same amount of time independent of load on the cluster because the joining node is not yet affected by this load. However, after this stage the system must apply buffered propagated write sets and the time for this stage increases as both the recovery time prior to it and the number of transactions per second increases. This behavior occurs because all propagated write sets have to be buffered for the total time of recovery, both of which increase as throughput increase.

5.6 Master Recovery Performance

Finally, we are interested in the performance of the protocol while providing failover to a new master in the event the master node fails.

The setup of this experiment differs to the normal configuration due to the requirements of introducing and dealing with a failure. For this experiment, we opted to provide the system with a throughput of 400 update transactions per second, roughly 20% of the saturation throughput for a seven node cluster. Since our workload is 20% updates and this experiment is only interested in update transactions (as they are what are rebuilt during recovery and read-only transactions are in principal, not affected) we chose this 20% figure to match the upload workload of a saturated seven node cluster. The workload of the test client consists of SQL INSERT queries that insert numbers in a monotonically increasing sequence. This produces a sufficiently large amount of work on the master node as it has to insert data appropriately and maintain the *primary key* B-Tree. It also allowed us to verify consistency at the end of the experiment by running a query to analyse the inserted data, checking for gaps in the sequence.

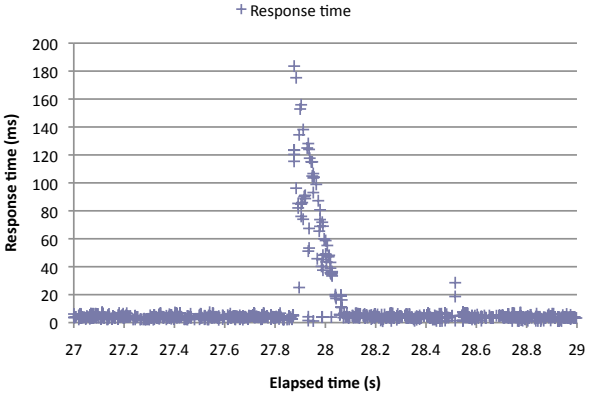


Figure 7: Master Failover Performance.

At some point during execution, we simulated a master node failure by deliberately killing the original master node. The recovery process then begins during which time the test client collects information about the response times of each transaction it submits. During this process each replication daemon also reports its recovery times to a log file: the time it takes from detecting the failure until the recovery process is complete. In all cases this never took longer than 100ms while the average was 80ms.

Figure 7 is a visualisation of the failover process. When a failure is introduced at around 28 seconds into the experiment, we see a spike in response times as request queries must block until the cluster has completed failover to a new master. The reason is twofold: firstly, any in-flight update transactions which started before the failure may have to be repeated increasing their response time. The second cause is that there is a window of approximately 100 milliseconds where update transactions will block waiting for recovery to complete. At 400 TPS this 100ms window is roughly equal to 40 transactions. Finally, after recovery completes and replayed transactions have completed there is a backlog of transactions that had been accumulated in the recovery time which must also now be completed. This creates a spike of load on the system and pushes up response times as there is a higher level of concurrency taking place due to the spike of requests. This backlog however, is eventually processed and the system reverts to its normal characteristics a small period of time after recovery is complete.

These results show strong performance during recovery and a minimum affect on the response time of update queries while recovery takes place. Such a fast recovery time is important, as our robustness mechanism only protects from single-master node failures as discussed in Section 3.6. We checked for gaps in the inserted sequence after every experiment in order to verify consistency. Each check confirmed that there were no gaps in the inserted sequence, showing that no write sets were lost during the recovery process.

6 Related Work

The popularity of SI in DBMS platforms such as Oracle and PostgreSQL led many researchers to explore this level of consistency as sufficient, in a replicated system. We point to a range of papers that address this, and we try to stress in each case how our design differs from theirs.

Patiño-Martínez et al. (2005) describes a solution using a middleware layer to coordinate the transactions, with group communication infrastructure to keep message orders consistent everywhere. All updates at all replicas are performed within a transaction. This solution parses the SQL in the middleware, to determine table-granularity read and write sets. An improvement by Lin et al. (2005) is to allow record-level concurrency. Elnikety et al. (2005) proposes also a solution with middleware; at transaction end, the middleware does a certification step calculating whether write sets intersect. An innovation here is the use of generalized SI as correctness condition, that allows a transaction T to see a state that does not include every transaction that committed before T started.

Pacitti et al. (2005) give another design with middleware in front of each replica, and a group communication system (albeit this paper requires only reliable and FIFO multicast, rather than totally ordered multicast as in the previous works). The main focus of this paper is to ensure that conflicting transactions are not running concurrently; it works with either SI or traditional serializable DBMS engines.

Plattner et al. (2006) use middleware, but with point-to-point communication rather than group communication. They execute the updating transactions entirely on a primary site, before propagating changes to the replicas, which are used to spread the load of read-only queries.

Proposals using a middleware architecture continue to appear. Pangea (Mishima & Nakamura 2009) uses eager propagation, in order to mimic exactly the choices of an unreplicated SI system when it must abort one of a pair of conflicting concurrent transactions. Krikellas et al. (2010) focus on modifying the design of Elnikety et al. (2005) to give either Strong Session SI or even 1-copy SI, rather than GSI; they do so by delaying the start of a transaction until the replica where it runs is sufficiently up-to-date.

Wu & Kemme (2005) a solution implemented in the Postgres kernel (that is, without middleware) but it uses group communication infrastructure for ordering messages, within the boundary of each transaction.

6.1 Commercial Cluster Products

Cluster of PCs have become an attractive hardware platform for commercial database vendors, too. Today, every database vendor enabled its product for clusters. While *Oracle Real Application Clusters 11g* follows a variant of shared-disk approach (Oracle 2012), most products such as *IBM DB2 UDB EEE* or *Microsoft SQL Server 2012* (IBM 1998, Microsoft 2012) favour a shared-nothing architecture. There are also clustering solutions available for the popular open source databases. For example for PostgreSQL, there are the Slony-I Replication System (2011), the PGCluster Project (2009), and the Cybercluster system, which we use in our performance evaluation (Cybercluster Project 2007). However, these systems do not address the problem of efficient replication management for large cluster sizes with correctness guarantees. For example, the available replication mechanisms exploit either standard 2PC (e.g. Cybercluster) or full asynchronous replication protocols without any up-to-date guarantees (e.g. the 'peer-to-peer transactional replication' of SQL Server 2012).

The following table summarizes the relationship between our algorithm and the related work:

Algorithm	Where?	When?	Comms
(Lin et al. 2005)	Middleware	Eager	Group
(Mishima & Nakamura 2009)	Middleware	Eager	Point
(Krikellas et al. 2010)	Middleware	Lazy	Point
(Plattner et al. 2006)	Middleware	Lazy	Master
(Cybercluster Project 2007)	Kernel	Eager	2PC
(Wu & Kemme 2005)	Kernel	Eager	Group
(Akal et al. 2005)	Kernel	Lazy	Master
RSR	**Kernel**	**Lazy**	**Point**

7 Summary and Conclusions

This paper presented *Robust Snapshot Replication* (RSR). This is a novel replication algorithm, which provides the consistency property of 1-copy Generalized Snapshot Isolation, in a system design which avoids the costs of either middleware, or group communication infrastructure. The computational workload of a user transaction

can be performed on any of the replicas, thus allowing work to be distributed fairly evenly; also each user-transaction involves the execution of one or more local DBMS-transactions which do not need two-phase commit.

We implemented the RSR algorithm in a cluster, combining separate PostgreSQL DBMS. We measured its performance, showing almost perfect speedup on 8 replicas; in contrast, a production replication algorithm based on eager replication slowed down with increasing cluster size. We also showed that RSR graciously handles the addition of a new node and that it is robust against node failures, e.g. for our workload performing a master failover within just 100 ms.

In future work, we want to extend our algorithm to provide Strong Session SI. We also wish to run experiments against some of the other system designs, which will require porting some to the current version of PostgreSQL.

Acknowledgment

Funding for part of this work was provided by Australian Research Council grant DP0987900.

References

Akal, F., Türker, C., Schek, H.-J., Breitbart, Y., Grabs, T. & Veen, L. (2005), Fine-grained replication and scheduling with freshness and correctness guarantees, *in* 'Proc VLDB'05', pp. 565–576.

Amza, C., Cox, A. L. & Zwaenepoel, W. (2000), Data replication strategies for fault tolerance and availability on commodity clusters, *in* 'Proc DSN'00'.

Berenson, H., Bernstein, P. A., Gray, J., Melton, J., O'Neil, E. J. & O'Neil, P. E. (1995), A critique of ansi sql isolation levels, *in* 'Proc SIGMOD'95'.

Bernstein, P. A. & Goodman, N. (1985), 'Serializability theory for replicated databases', *J. Comput. Syst. Sci.* **31**(3), 355–374.

Bernstein, P. A., Hadzilacos, V. & Goodman, N. (1987), *Concurrency Control and Recovery in Database Systems*, Addison-Wesley.

Cecchet, E., Candea, G. & Ailamaki, A. (2008), Middleware-based database replication: the gaps between theory and practice, *in* 'Proc SIGMOD'08'.

Cybercluster Project (2007).
URL: *http://pgfoundry.org/projects/cybercluster/*

Daudjee, K. & Salem, K. (2006), Lazy database replication with snapshot isolation, *in* 'Proc VLDB'06', pp. 715–726.

Elnikety, S., Zwaenepoel, W. & Pedone, F. (2005), Database replication using generalized snapshot isolation, *in* 'Proc SRDS'05', pp. 73–84.

Gray, J., Helland, P., O'Neil, P. E. & Shasha, D. (1996), The dangers of replication and a solution, *in* 'Proc SIGMOD'96', pp. 173–182.

IBM (1998), IBM DB2 universal database on IBM Netfinity and GigaNet cLan clusters, Technical report, IBM Corporation.

Krikellas, K., Elnikety, S., Vagena, Z. & Hodson, O. (2010), Strongly consistent replication for a bargain, *in* 'Proc ICDE '10', pp. 52–63.

Liang, W. & Kemme, B. (2008), Online recovery in cluster databases, *in* 'Proc EDBT'08', pp. 121–132.

Lin, Y., Kemme, B., Patiño-Martínez, M. & Jiménez-Peris, R. (2005), Middleware based data replication providing snapshot isolation., *in* 'Proc. SIGMOD'05', pp. 419–430.

Lomet, D. B., Barga, R. S., Mokbel, M. F. & Shegalov, G. (2006), Transaction time support inside a database engine., *in* 'Proc ICDE'06', p. 35.

Microsoft (2012), 'Microsoft SQL Server'.
URL: *http://www.microsoft.com/sql*

Mishima, T. & Nakamura, H. (2009), Pangea: An eager database replication middleware guaranteeing snapshot isolation without modification of database servers, *in* 'Proc. VLDB'09', pp. 1066–1077.

Open Source Development Lab (2004), *Descriptions and Documentation of OSDL-DBT-1*.
URL: *http://osdldbt.sourceforge.net/*

Oracle (2012), 'Oracle Real Application Clusters', **URL:** http://www.oracle.com/technology/products/database/clustering.

Pacitti, E., Coulon, C., Valduriez, P. & Özsu, M. T. (2005), 'Preventive replication in a database cluster', *Distributed and Parallel Databases* **18**(3).

Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B. & Alonso, G. (2005), 'Middle-r: Consistent database replication at the middleware level', *ACM Trans. Comput. Syst.* **23**(4), 375–423.

PGCluster Project (2009).
URL: *http://pgfoundry.org/projects/pgcluster/*

Plattner, C. & Alonso, G. (2004), Ganymed: Scalable replication for transactional web applications, *in* 'Proc Middleware'04', pp. 155–174.

Plattner, C., Alonso, G. & Özsu, M. T. (2006), DBFarm: a scalable cluster for multiple databases, *in* 'Proc. Middleware'06', pp. 180–200.

PostgreSQL Global Development Group (2007), 'PostgreSQL 8.2 documentation'.
URL: *http://www.postgresql.org/docs/8.2/static/*

Slony-I Replication System (2011).
URL: *http://slony.info*

TPC (2002), *TPC-W Benchmark Specification Rev. 1.8*, Transaction Processing Performance Council.
URL: *http://www.tpc.org/tpcw/*

Wu, S. & Kemme, B. (2005), Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation, *in* 'Proc ICDE'05'.