On the Performance of Wide-Area Synchronous Database Replication

Yair Amir * Claudiu Danilov * Michal Miskin-Amir † Jonathan Stanton ‡
Ciprian Tutu *

Technical Report
CNDS-2002-4
http://www.cnds.jhu.edu

November 18, 2002

Abstract

A fundamental challenge in database replication is to maintain a low cost of updates while assuring global system consistency. The difficulty of the problem is magnified for wide-area network settings due to the high latency and the increased likelihood of network partitions. As a consequence, most of the research in the area has focused either on improving the performance of local transaction execution or on replication models with weaker semantics, which rely on application knowledge to resolve potential conflicts.

In this work we identify the performance bottleneck of the existing synchronous replication schemes as residing in the update synchronization algorithm. We compare the performance of several such synchronization algorithms and highlight the large performance gap between various methods. We design a generic, synchronous replication scheme that uses an enhanced synchronization algorithm and demonstrate its practicality by building a prototype that replicates a PostgreSQL database system. We claim that the use of an optimized synchronization engine is the key to building a practical synhronous replication system for wide-area network settings.

1 Introduction

Database management systems are among the most important software systems driving the information age. In many Internet applications, a large number of users that are geographically dispersed may routinely query and update the same database. In this environment, a centralized database is exposed to several significant risks:

- performance degradation due to high server load.
- data-loss risk due to server crashes.
- high latency for queries issued by remote clients
- availability issues due to lack of network connectivity or server downtime.

*Computer Science Department, Johns Hopkins University, Baltimore, MD 21218, USA. Email: {yairamir, claudiu, ciprian}@cnds.jhu.edu

†Spread Concepts LLC, Email: michal@spreadconcepts.com

‡Computer Science Department, George Washington University, Washington, DC 20052, USA. Email: jstanton@gwu.edu

The apparent solution to these problems would be to synchronously replicate the database server on a set of peer servers. In such a system queries can be answered by any of the servers, without any additional communication; however, in order for the system to remain consistent, all the transactions that update the database need to be disseminated and synchronized at all the replicas. Obviously, if most of the transactions in the system are updates, a replicated system trades off performance for availability and fault tolerance. By replicating the database on a local cluster, this cost is relatively low and the solution successfully addresses the first two problems. However, response time and availability due to network connectivity remain valid concerns when clients are scattered on a wide-area network and the cluster is limited to a single location.

In wide-area network settings, the cost of synchronizing updates among peer replicas, while maintaining global system consistency is magnified by the high network latency and the increased likelihood of network partitions. As a consequence, synchronous database replication has been deemed impractical and the research in the area has focused on improving the performance of transaction execution on a single server (maximizing the concurrency degree, etc.) and on the design of alternative, asynchronous replication schemes, that ultimately rely on the application to resolve potential conflicts[1].

In this paper we define the building blocks of a generic, peer, synchronous replication system and identify the performance bottleneck of such systems. We examine classical synchronous replication methods [12], more enhanced replication systems [17], as well as some of the more recent, non-generic approaches that exploit application semantics or tightly-coupled integration with the database in order to gain in performance [21, 16]. We notice that the critical component in the performance of the system is the method employed to synchronize the update transactions among the participating replicas.

We present a generic symmetric synchronous replication system that uses an enhanced update synchronization method. We compare the performance of several synchronization methods and show the increased performance and scalability that an enhanced synchronization method can bring in both local and wide-area network settings. We build our solution as a generic system (a black box) that can be employed to replicate a database system without modifying the database and that can be used transparently by a large number of applications. We investigate the limitations imposed by this design and we show how the design can be adapted to alleviate them and provide efficient synchronous replication when tightly integrated with a specific database system or when adapted to the needs of specific applications. Furthermore, we argue that a transparent replication architecture not only avoids complex tailoring to a specific application or database systems, but can also facilitate interoperability between different database systems. We present the performance of a prototype system using the PostgreSQL database system.

The remainder of the paper is organized as follows: Section 2 surveys the related work in the general area. In section 3 we inspect the characteristics of existing replication systems with special emphasis on synchronous replication systems. In section 3.3 we compare the performance of several synchronization algorithms. Section 4 describes our proposed transparent synchronous replication architecture. In section 4.4 we evaluate the performance of our PostgreSQL prototype. Section 5 concludes the paper.

[1] Note that some applications critically require continuous data consistency and therefore would not work with asynchronous replication schemes.

2 Related Work

We present a solution for peer synchronous replication maintaining strict consistency semantics. Throughout the rest of the paper, and in particular in Section 3.2, we provide detailed comparison of existing synchronous replication methods. In this section we give an overview of the state of the art over the entire spectrum of database replication methods.

Despite their inefficiency and lack of scalability, two-phase commit protocols [12] remain the principal technique used by most commercial database systems that try to provide synchronous peer replication.

The Accessible Copies algorithms [1, 2] maintain an approximate view of the connected servers, called *a virtual partition*. A data item can be read/written within a virtual partition only if this virtual partition (which is an approximation of the current connected component) contains a majority of its read/write votes. If this is the case, the data item is considered accessible and read/write operations can be done by collecting sub-quorums in the current component. The maintenance of virtual partitions greatly complicates the algorithm. When the view changes the servers need to execute a protocol to agree on the new view, as well as to recover the most up-to-date item state. Moreover, although view decisions are made only when the "membership" of connected servers changes each update requires end-to-end acknowledgments from the quorum.

Most of the state-of-the-art commercial database systems provide some level of database replication. However, in all cases, their solutions are highly tuned to specific environment settings and require a lot of effort in their setup and maintenance. Oracle [27], supports both asynchronous and synchronous replication. However, the former requires some level of application decision in conflict resolution, while the latter requires that all the replicas in the system are available to be able to function, making it impractical. Informix [14], Sybase [33] and DB2 [9] support only asynchronous replication which again ultimately relies on the application for conflict resolution.

In the open-source database community, two database systems emerge as clear leaders: MySQL [26] and PostgreSQL [31]. By default both systems only provide limited master-slave replication capabilities. Other projects exist that provide more advanced replication methods for Postgres such as Postgres Replicator, which uses a trigger-based store and forward asynchronous replication method [29].

The most evolved of these approaches is Postgres-R [30], a project that combines open-source expertise with academic research. Postgres-R implements algorithms designed by Kemme and Alonso [21] into the PostgreSQL database manager in order to provide synchronous replication. The current version uses Spread [32] as the underlying group communication system and focuses on integrating the method with version 7.2 of the PostgreSQL system.

Research on protocols to support group communication across wide area networks such as the Internet has begun to expand. Recently, new group communication protocols designed for such wide area networks have been proposed [20, 19, 3, 5] which continue to provide the traditional strong semantic properties such as reliability, ordering, and membership. The only group communication systems we are aware of that currently exist, are available for use, and can provide the Extended Virtual Synchrony semantics are Horus [34], Ensemble [13], and Spread [6].

3 Synchronous Database Replication

3.1 Replication Methods

Any replicated database system needs to perform two main functions: execute transactions locally and make sure that the outcome of each transaction is reflected at all replicas. The later task can

be achieved in several ways. In the master-slave approach, one master replica is in charge of executing/committing transactions locally before it disseminates the transactions (or just their outcome) to slave replicas. This method is the closest to the single-server solution both in its performance (all concurrent transaction execution optimizations can be applied) as well as in limitations (high latency for remote clients, reduced availability in face of network problems). In this model the consistency of the replicas is guaranteed by the consistency of execution on one replica, but the slave replicas are only lazily updated and cannot provide accurate responses to queries.

The asynchronous replication approach disseminates the transactions for parallel execution on all replicas. Each replica can answer queries and can initiate the execution of a transaction, but consistency is not always guaranteed, since conflicting transactions may be executed in different order at different replicas. These situations are detected and conflict resolution procedures are applied in order to restore consistency. If no conflicts arise, the transaction execution in the system can proceed very fast as each replica can locally execute the transactions as soon as they are received.

Synchronous replication methods guarantee that all replicas are maintained consistent at all times by executing each transaction locally only after all replicas have agreed on the execution order. This way a very strict level of consistency is maintained while providing high availability and good latency response to clients. However, because of the strict synchronization between replicas that is required for each transaction, synchronous replication methods have been deemed inpractical [11] and often times a centralized or master-slave approach is preferred for systems that critically require strict consistency. It is believed that the trade-off between performance (throughput) and client response time is too unbalanced for practical needs. In the following sections we argue that there exist efficient synchronization algorithms that can be integrated in a complete architecture for synchronous database replication with good performance on **both** local and wide-area networks.

3.2 Synchronous Replication Methods

Two-Phase Commit (2PC) is the most widely used synchronous replication method. Analyzing 2PC, it becomes obvious why the performance tradeoff was deemed too high for this kind of replication: 2PC requires 3n unicast messages per transaction and one forced disk write per transaction per replica. Under local area network settings, the number of forced disk writes required in order to guarantee consistency will impact the performance, while on wide-area settings, the communication price is too high. Furthermore, 2PC is also vulnerable to faults at critical points (the crash of a transaction coordinator blocks the system until the crashed server recovers). In order to enhance the fault-tolerance of 2PC, three-phase commit (3PC) algorithms [18] were introduced, but they pay an even higher communication cost and therefore are rarely used in practice.

Keidar [17] introduces COReL, a replication algorithm that exploits group communication properties in order to reduce the communication costs of the synchronization procedure. The algorithm establishes a global consistent persistent order of the transactions and commits them on all replicas according to this order. The algorithm requires n multicast messages per transaction, plus one forced disk write per action per replica. The use of multicast instead of unicast (as in 2PC) is a clear improvement, but the COReL algorithm still performs per-transaction end-to-end synchronizations in order to guarantee correctness. Lamport introduces Paxos [23, 24] a highly robust replication algorithm that pays a similar price to the COReL algorithm (n unicast messages per transaction plus one forced disk write per transaction per replica).

In [7, 4] Amir and Tutu introduce and prove the correctness of a synchronization algorithm, based on an enhanced group communication paradigm (Extended Virtual Synchrony - EVS [25]). The algorithm only requires one "safe" multicast message per transaction and 1/n forced disk writes per transaction per replica. Similar to COReL each replica commits transactions according to the

global consistent persistent order established, but avoids the use of end-to-end acknowledgements by benefiting from the lower level guarantees that the group communication provides. In particular, the EVS-based group communication system provides complete and accurate membership information informing the synchronization algorithm of the currently connected members and a special type of total order delivery called *safe delivery*[2]. A message is safely delivered to a server only when the total order of the message has been established **and** the group communication guarantees that **all** the currently connected members have received the message. The EVS system provides a special delivery in the case where a message is received, but a network event (partition/crash) occurs before the stability of the message can be determined. Using this knowledge, the algorithm can guarantee consistency when no faults occur, without resorting to the use of end-to-end acknowledgements per transaction, but using the implicit network-level acknowledgements that the group communication primitives provide[3]. Expensive end-to-end synchronization is required only when faults occur and the set of currently connected servers changes.

In [21] Kemme and Alonso demonstrate the practicality of synchronous database replication in a local area network environment, where network partitions cannot occur. Their approach is tightly integrated with the database system and uses group communication in order to totally order the write-sets of each transaction and establish the order in which locks are acquired to avoid deadlocks. In contrast to the method above, this algorithm requires two multicast messages (one total order multicast and one generic multicast) for each transaction. However, this method allows concurrent execution of transactions at each site, if allowed by the local concurrency manager, although transactions may be aborted due to conflicts. The [21] also presents performance results of a PostgreSQL replicated database prototype (Postgres-R). In [22], the authors examine several methods to provide online reconfiguration for a replicated database system, in order to make it cope with network partitions and server crashes and recoveries.

In [28, 16] the authors present yet another approach to the replication problem, also targeted at local area clusters, without supporting network partitions and merges or server recoveries. This solution is not tightly integrated with the database nor is it completely transparent, but it requires the application to provide *conflict class* information in order to allow concurrent processing of non-conflicting transactions. Similar to [7], each transaction is encapsulated in a message that is multicast to all sites. The transaction will be executed only at the "master" site for its conflict class based on the optimistic delivery order. A commit message containing the write-set of the transaction is issued if the optimistic order does not conflict with the final total order. The write-set is then applied to all replicas according to the total order. All of the more recent methods take advantage of the primitives offered by the modern group communication systems. However, the various solutions are different in the degree in which they exploit the group communication properties. We argue that an optimal usage of the group communication primitives can lead to significant performance improvements for the synchronization algorithm and that these improvements, in turn, can be used to build a practical synchronous database replication solution for both local and wide-area networks.

3.3 Synchronization Methods Evaluation

In this section we compare the performance of three of the synchronization methods described above. We compare the EVS-based replication engine described in [7, 4], a COReL [17] implementation and an *upper-bound* 2PC implementation. We chose to use these three algorithms in our comparison for several reasons: they can all cope with wide-area network settings, the complete algorithm

[2] In the group communication literature this type of delivery is sometimes refered to as *stable delivery*

[3] The vast amount of research in the group communication area has lead to the design and implementation of several group communication systems with highly-efficient ordered communication primitives: spread, ensemble, etc.

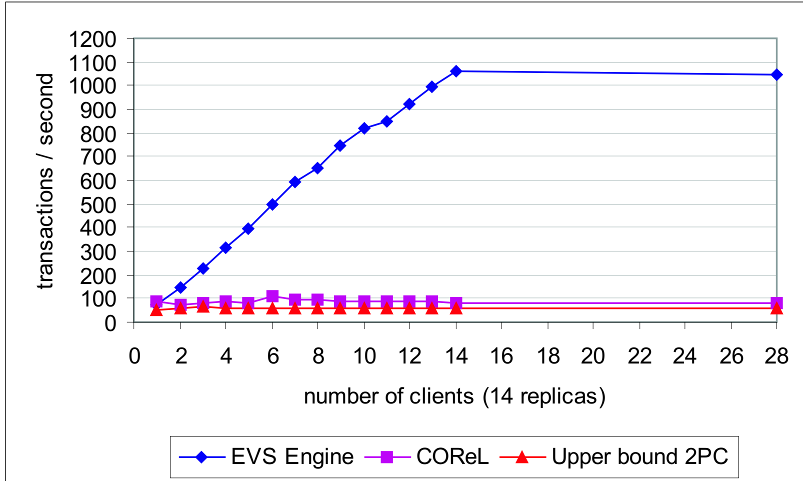

Figure 1: LAN Synchronization Comparison

pseudocodes were available together with correctness proofs and they can be naturally implemented independently of the database system. Our 2PC implementation will assume that all the locks are granted instantly, thus ensuring the maximum level of concurrency the 2PC method can support. In essence, we just send the necessary messages and perform the necessary forced writes to disk. For this reason, we call this implementation *upper-bound* 2PC. The COReL and the EVS engines are implemented using Spread [32] as the underlying group communication.

We explore the performance of the synchronization methods in two setups. A local area cluster and the Emulab wide-area testbed [10]. The cluster contains 14 Linux Dual PIII 667 computers with 256Mbytes and 9G SCSI disk. Emulab[4] (the Utah Network Test-bed) provides a configurable test-bed where the network setup sets the latency, throughput and link-loss characteristics of each of the links. The configured network is then emulated in the Emulab local area network, using actual routers and in-between computers that emulate the required latency, loss and capacity constraints. We use 7 Emulab Linux computers to emulate the CAIRN [8] network depicted in Figure 2. Each of the Emulab machines is a PIII 850 with 512Mbytes and 40G IDE disk.

Our tests assume that all the actions represent update transactions and are therefore replicated and committed on all replicas. Each client submits one action at a time. Once that action is committed, the client receives an acknowledgement from its peer server and can generate a new action.

In Figure 1 we present the throughput comparison between the three algorithms on our local area network testbed. The 2PC algorithm reaches a maximum throughput of approximately 60 actions per second while COReL reaches approximately 100 actions per second. The EVS-based synchronization engine reaches its peak at over 1000 actions per second.

We next evaluated the algorithms on Emulab, emulating the CAIRN wide-area network depicted in Figure 2 as accurately as possible. The diameter of the network as measured by *ping* is approximately 45ms involving seven sites. We validated Emulab against CAIRN by measuring the Safe Delivery latency under different message loads. Under all loads both networks provided very similar message latency (details are available in a Technical Report).

In Figure 3 we compare the EVS synchronization engine with the upper-bound 2PC implementation on the wide area network depicted in Figure 2. The upper-bound 2PC engine reaches its

[4]Emulab is available at www.emulab.net and is primarily supported by NSF grant ANI-00-82493 and Cisco Systems.

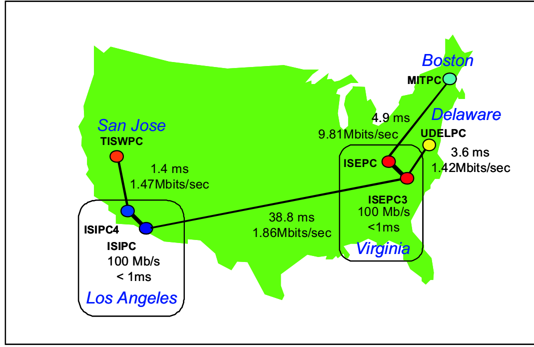

Figure 2: Layout of the CAIRN Network

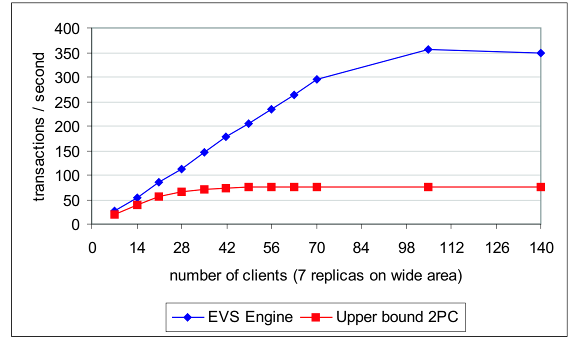

Figure 3: WAN Synchronization Comparison

maximum capacity at approximately 80 actions per second, while the engine is saturated at 350 actions per second.

The results obtained in these experiments show that there can be significant performance difference between various synchronization techniques. The more important aspect is the fact that there also exist very efficient synchronization methods (such as the EVS-based engine) which can sustain a high amount of throughput in both local and wide-area network conditions. While it is difficult to compare in isolation the EVS synchronization engine with the core engines employed in [21, 16] because of their tight integration with the database or dependence on additional application semantics, we argue that the gain in performance due to optimizing the network synchronization can outweigh the gains obtained through parallelization in transaction processing.

4 A Transparent Synchronous Replication Scheme

To further validate our hypothesis which identifies the synchronization module as the main performance bottleneck in a synchronous replication scheme, we develop and evaluate a complete architecture that replicates a Postgres database system.

4.1 Architecture Overview

We present an architecture that provides transparent peer replication, supporting diverse application semantics. At the core of the system we use the synchronization algorithm benchmarked in the previous section. Peer replication is a symmetric approach where each of the replicas is guaranteed to invoke the same set of actions in the same order. This approach requires the next state of the database to be determined by the current state and the next action, and it guarantees that all of the replicas reach the same database state.

Throughout this section we use the generic term *action* to refer to any non-interactive deterministic, multi-operation database transaction, such that the state of the database after the execution of a new action is defined exclusively by the state of the database before the execution of that action and the action itself. Each database transaction (e.g. a multi-operation SQL transaction) will be packed into one *action* by our replication engine. In this model, a user cannot abort transactions submitted into the system after their initiation. Since the transactions are assumed deterministic, if an abort operation is present within the transaction boundaries, it will be executed on all replicas or on none of them, as dictated by the database state and the transaction itself at the time of the execution. In Section 4.3, we discuss how the architecture can support more generic transaction types. In the

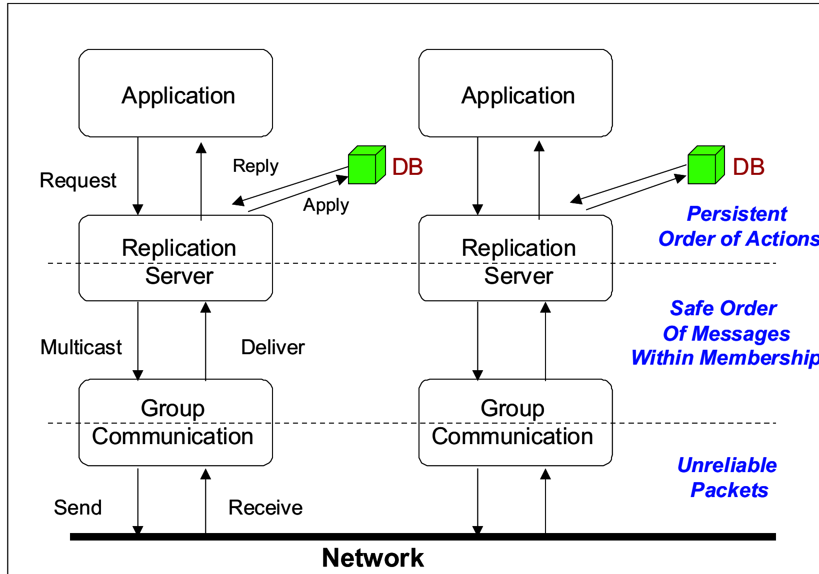

Figure 4: Synchronous Database Replication Architecture

following sections we will refer to read-only transactions as *queries* while non read-only transactions (those that contain update operations) will be referred to as *updates*.

The architecture is structured into two layers: a replication server and a group communication toolkit (Figure 4).

Each of the replication servers maintains a private copy of the database. The client application *requests* an action from one of the replication servers. The replication servers agree on the order of actions to be performed on the replicated database. As soon as a replication server knows the final order of an action, it *applies* this action to the database (execute and commit). The replication server that initiated the action returns the database *reply* to the client application. The replication servers use the group communication toolkit to disseminate the actions to the servers group and to help reach an agreement about the final global order of the set of actions.

In a typical operation, when an application submits a request to a replication server, this server logically *multicasts* a message containing the action through the group communication. The local group communication toolkit *sends* the message over the network. Each of the **currently connected** group communication daemons eventually *receives* the message and then *delivers* the message in the same order to their replication servers.

The group communication toolkit provides multicast and membership services according to the Extended Virtual Synchrony model [25]. For this work, we are particularly interested in the Safe Delivery property of this model. Similarly to the current Postgres-R [30] approach we chose Spread [32] as our supporting group communication system. The group communication toolkit overcomes message omission faults and notifies the replication server of changes in the membership of the currently connected servers. These notifications correspond to server crashes and recoveries or to network partitions and re-merges. When notified of a membership change by the group communication layer, the replication servers exchange information about actions sent before the membership change. This exchange of information ensures that every action known to any member of the currently connected servers becomes known to all of them. Moreover, knowledge of final order of actions is also shared among the currently connected servers. As a consequence, after this exchange is completed, the state of the database at each of the connected servers is identical. The cost of such synchronization

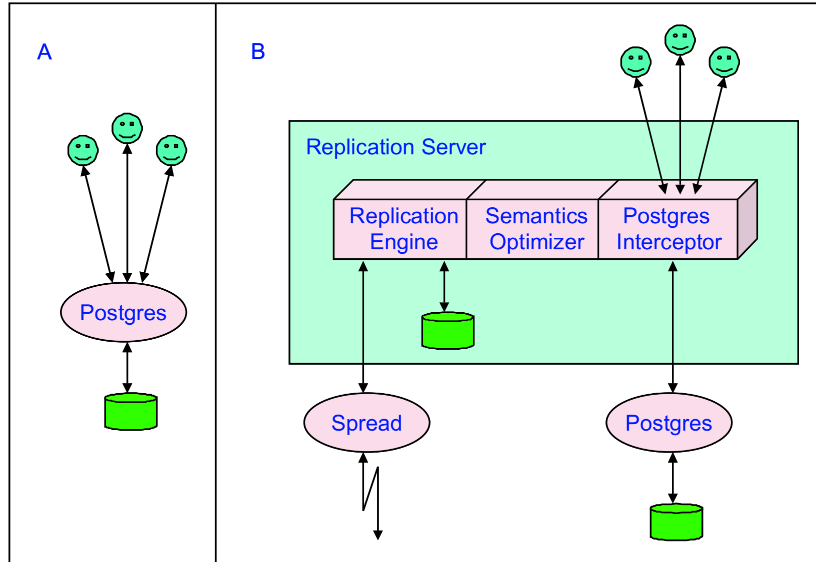

Figure 5: Transparent Postgres Replication Architecture

amounts to one message exchange among all connected servers plus the retransmission of all updates that at least one connected server has and at least one connected server does not have. Of course, if a site was disconnected for an extended period of time, it might be more efficient to transfer a current snapshot [22].

The consistency of the system in the presence of network partitions and remerges or server crashes and recoveries is guaranteed through the use of a quorum[5] system that uniquely identifies one of the connected components as the *primary* component of the system. Updates can continue to be committed only in the primary component, but the other components continue to remain active and answer queries consistently (although outdated).

Advanced replication systems that support a peer-to-peer environment must address the possibility of conflicts between the different replicas. Our architecture eliminates the problem of conflicts because updates are always invoked in the same order at all the replicas. Of course, in its basic form this model also excludes the possibility of concurrently executing updates on the same replica. This restriction is inherent to any completely generic solution that does not assume any knowledge about the application semantics nor does it rely on the application to resolve potential conflicts.

The latency and throughput of the system for updates are obviously highly dependent on the performance of the group communication Safe Delivery service. Queries will not be sent over the network as they can be answered immediately by the local database.

An important property that our architecture achieves is transparency - it allows replicating a database without modifying the existing database manager or the applications accessing the database. The architecture does not require extending the database API and can be implemented directly above the database or as a part of a standard database access layer (e.g. ODBC or JDBC).

Figure 5.A presents a non-replicated database system that is based on the Postgres database manager. Figure 5.B. presents the building blocks of our implementation, replicating the Postgres database system. The building blocks include a replication server and the Spread group communication toolkit. The database clients see the system as in figure 5.A., and are not aware of the replication although they access the database through our replication server. Similarly, any instance

[5]Examples of quorum systems include monarchy, majority, dynamic linear voting [15]

of the database manager sees the local replication server as a client.

The replication server consists of several independent modules that together provide the database integration and consistency services (Figure 5.B). They include:

- A Replication Engine that includes all of the replication logic from the synchronizer algorithm, and can be applied to any database or application. The engine maintains a consistent state and can recover from a wide range of network and server failures. The replication engine is based on the algorithm presented in [7, 4].
- A Semantics Optimizer that can decide whether to replicate transactions and when to apply them based on application semantics if such is available, the actual content of the transaction, and whether the replica is in a primary component or not.
- A database specific interceptor that interfaces the replication engine with the DBMS client-server protocol. As a proof of concept, to replicate Postgres, we created a Postgres specific interceptor. Existing applications can transparently use our interceptor layer to provide them with an interface identical to the Postgres interface, while the Postgres database server sees our interceptor as a regular client. The database itself does not need to be modified nor do the applications. A similar interceptor could be created for other databases.

4.2 The Semantics Optimizer

The Semantics Optimizer provides an important contribution to the ability of the system to support various application requirements as well as to the overall performance. The level of sophistication that the semantics optimizer incorporates or the degree of integration with the database system will determine the amount of additional optimizations that the replication system can take advantage of (concurrent transaction execution, data partitioning). It is not the purpose of this paper to detail how this integration can be achieved, but we outline some basic variations from the standard implementation to illustrate the flexibility of the architecture.

In the strictest model of consistency, updates can be applied to the database only while in a primary component, when the global consistent persistent order of the action has been determined. However, read-only actions (queries) do not need to be replicated. A query can be answered immediately by the local database if there is no update pending generated by the same client. A significant performance improvement is achieved if the system distinguishes between queries and actions that also update the database. For this purpose in the Postgres prototype that we built, the Semantics Optimizer implements a very basic SQL parser that identifies the queries from the other actions.

If the replica handling the query is not part of the primary component, it cannot guarantee that the answer of its local database reflects the current state of the system, as determined by the primary component. Some applications may require only the most updated information and will prefer to block until that information is available, while others may be content with outdated information that is based on a prior consistent state (*weak consistency*), preferring to receive an immediate response. The system can allow each client to specify its required semantics individually, upon connecting to the system. The system can even support such specification for each action but that will require the client to be aware of the replication service.

In addition to the strict consistency semantics and the standard weak consistency, the implementation supports, but is not limited to, two other types of consistency requirements: *delay updates* and *cancel actions,* where both names refer to the execution of updates/actions in a non-primary component. In the *delay updates* semantics, transactions that update the database are ordered locally, but are not applied to the database until their global order is determined. The client is not blocked and can continue submitting updates or even querying the local database, but needs to be aware that the responses to its queries may not yet incorporate the effect of its previous updates.

In the *cancel actions* semantics a client instructs the Semantics Optimizer to immediately abort the actions that are issued in a non-primary component. This specification can also be used as a method of polling the availability of the primary component from a client perspective. These decisions are made by the Semantics Optimizer based on the semantic specifications that the client or the system setup provided.

The following examples demonstrate how the Semantics Optimizer determines the path of the action as it enters the replication server. After the Interceptor reads the action from the client, it passes it on to the Semantics Optimizer. The optimizer detects whether the action is a query and, based on the desired semantics and the current connectivity of the replica, decides whether to send the action to the Replication Engine, send it directly to the database for immediate processing, or cancel it altogether.

If the action is sent through the Replication Engine, the Semantics Optimizer is again involved in the decision process once the action is ordered. Some applications may request that the action is optimistically applied to the database once the action is locally ordered. This can happen either when the application knows that its update semantics is commutative (i.e. order is not important) or when the application is willing to resolve the possible inconsistencies that may arise as the result of a conflict. Barring these cases, an action is applied to the database when it is globally ordered.

4.3 Design Tradeoffs

From a design perspective, we can distinguish between three possible approaches to the architecture of a replicated database. We opted for the *black box* approach that does not assume any knowledge about the internal structure of the database system or about the application semantics. The flexibility of this architecture enables the replication system to support heterogeneous replication where different database managers from different vendors replicate the same logical database.

In contrast, a *white box* approach [21] will integrate the replication mechanism within the database itself, attempting to exploit the powerful mechanisms (concurrency control, conflict resolution) that are implemented inside the database, at the price of losing transparency. A middle-way *gray box* approach [16] assumes that the database system is enhanced by providing additional primitives that can be used from the outside but does not include the replication mechanism inside the database itself. [16] also exploits the data partitioning common in many databases, assuming that the application can provide information about the conflict classes that are addressed by each transaction.

We argue that although our design does not allow, in its basic form, concurrent transaction execution, it doesn't suffer a performance drawback because it uses a more efficient synchronization algorithm. As well, even a highly concurrent synchronous replication system cannot overcome the fundamental cost of global synchronization and would be therefore limited by the performance of they synchronization module.

In the presentation of our model we mention that our replication architecture assumes that each (possibly multi-operation) transaction is deterministic and can be encapsulated in one action, thus removing the possibility of executing normal interactive transactions. This assumption can be relaxed to a certain degree. We can allow, for example, a transaction to execute a deterministic procedure that is specified in the body of the transaction and that depends solely on the current state of the database. These transactions are called active transactions.

With the help of active transactions, one can mimic interactive transactions where a user starts a transaction, reads some data then makes a user-level interactive decision regarding updates. Such transactions can be simulated with the help of two actions in our model. The first action contains the query part of the transaction. The second action is an active action that encapsulates the updates dictated by the user, but first checks whether the values of the data read by the first action are still

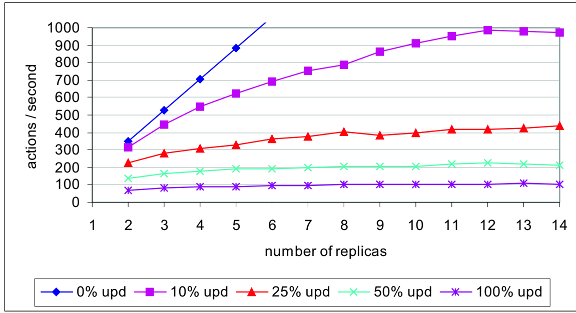

Figure 6: LAN Postgres Throughput under Varying Number of Replicas

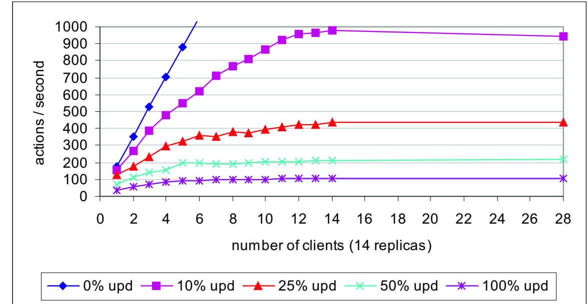

Figure 7: LAN Posgres Throughput

valid. If they are not valid, the second action is aborted (as if the transaction was aborted in the traditional sense.) Note that if one server aborts, all of the servers abort that (trans)action since they apply an identical deterministic rule to an identical state of the database, as guaranteed by the algorithm.

4.4 Prototype Performance Evaluation

We evaluated the performance of the Postgres replication prototype in the two environments described in Section 3.3.

In order to better understand the context of the results we measured the throughput that a single non-replicated database can sustain from a single, serialized stream of transactions. The computers used in the local area experiments can perform on a non-replicated Postgres database approximately 120 update transactions or 180 read-only transactions per second, as defined below. The computers used in the wide-area experiment can perform on a non-replicated Postgres database approximately 120 update transactions or 210 read-only transactions per second, as defined below.

Our experiments were run using PostgreSQL version 7.1.3 standard installations. In order to facilitate comparisons, we use a database and experiment setup similar to that introduced by [21, 16].

The database consists of 10 tables, each with 1000 tuples. Each table has five attributes (two integers, one 50 character string, one float and one date). The overall tuple size is slightly over 100 bytes, which yields a database size of more than 1MB. We use transactions that contain either only queries or only updates in order to simplify the analysis of the impact each poses on the system. We control the percentage of update versus query transactions for each experiment. Each action used in the experiments was of one of the two types described below, where table-i is a randomly selected table and the value of t-id is a randomly selected number:

```
update table-i set attr1="randomtext",
                   int_attr=int_attr+4
where t-id=random(1000);}

select avg(float_attr), sum(float_attr) from table-i;
```

Before each experiment, the Postgres database was “vacuumed” to avoid side effects from previous experiments. Each client only submits one action (update or query) at a time. Once that action has completed, the client can generate a new action.

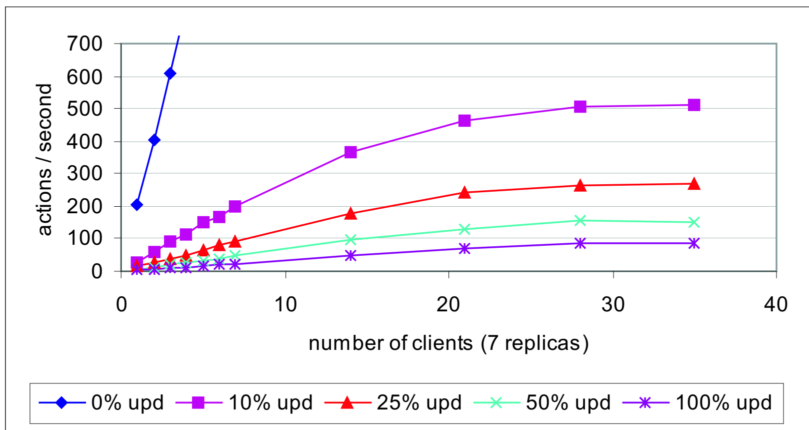

Figure 8: WAN Postgres Throughput

The first experiment tested the scalability of the system as the number of replicas of the database increases. Each replica executes on a separate computer with one local client over a local area network. Figure 6 shows five separate experiments. Each experiment uses a different proportion of updates to queries.

The 100% update line shows the disk bound performance of the system. As replicas are added, the throughput of the system increases until the maximum updates per second the disks can support is reached – about 107 updates per second with replication (which adds one additional disk sync for each N updates, N being the number of replicas). Once the maximum is reached, the system maintains a stable throughput. The achieved update throughput, when the overhead of the Replication Server disk syncs are taken into account, matches the potential number of updates the machine is capable of. The significant improvement in the number of sustained actions per second when the proportion of queries to updates increases is attributed to the Semantics Optimizer, which executes each query locally without any replication overhead. The maximum throughput of the entire system actually improves because each replica can handle an additional load of queries. The throughput with 100% queries increases linearly, reaching 2473 queries per second with 14 replicas as expected.

The next experiment fixes the number of replicas at 14, one replica on each computer on the local area network. The number of clients connected to the system is increased from 1 to 28, evenly spread among the replicas. In Figure 7 one sees that a small number of clients cannot produce maximum throughput for updates. The two reasons for this are: first, each client can only have one transaction active at a time, so the latency of each update limits the number of updates each client can produce per second. Second, because the Replication Server only has to sync updates generated by locally connected clients, the work of syncing those updates is more evenly distributed between the computers as clients are added. Again, the throughput for queries increases linearly up to 14 clients (reaching 2450 in the 100% queries case), and is flat after that as each database replica is already saturated.

The wide area experiment conducted on Emulab measured the throughput under varying client set and action mix. The system was able to achieve a throughput close to that achieved on a local area network with a similar number of replicas (seven), but with more clients as depicted in Figure 8. The latency each update experiences in this experiment is 268ms when no other load is present and reaches 331ms at the point the system reaches the maximum throughput of 85 updates per second. For queries, similarly to the LAN test, the performance increases linearly with the number of clients until the seven servers reach 1422 queries per second. The results for in-between mixes are as expected.

The Postgres proof of concept demonstrates that our overall architecture can replicate a database in a useful and efficient manner. By using a more efficient synchronization method than previous solutions we are able to sustain throughput rates that would be sufficient for a large number of applications while maintaining a reasonably low response time even in wide-area settings. Comparing the results in our experiments to those presented in [21, 16] we can notice a significant improvement in both latency and overall throughput[6]. We attribute the performance gain to the use of a more efficient synchronization algorithm, despite giving up on additional gains through use of concurrent transaction execution.

5 Conclusion

In this paper we have presented a transparent peer synchronous database replication architecture that employs an optimized update synchronization algorithm in order to improve the performance of the system. In contrast with the existing techniques we sacrifice the performance gains attained through parallelization of transaction execution in favor of an enhanced synchronization method - the real bottleneck in a synchronous replication system, and we show the viability of the approach through practical experimentation. The improvements are noticeable both on the local area network but especially in wide-area experiments. Furthermore, the synchronization engine that was used to build a generic, transparent solution in this work, can be adapted and employed in replication solutions that are integrated with the database or that exploit specific application information, creating an exciting new realm of opportunities in database replication.

References

[1] A. E. Abbadi, D. Skeen, and F. Cristian. An efficient fault-tolerant algorithm for replicated data management. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 215–229. ACM, March 1985.

[2] A. E. Abbadi and S. Toueg. Availability in partitioned replicated databases. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 240–251. ACM, March 1986.

[3] D. Agarwal, L. Moser, P. Melliar-Smith, , and R. Budhia. The totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, May 1998.

[4] Y. Amir. *Replication Using Group Communication over a Partitioned Network*. PhD thesis, Hebrew University of Jerusalem, Jerusalem, Israel, 1995. http://www.cnds.jhu.edu/publications/yair-phd.ps.

[5] Y. Amir, C. Danilov, and J. Stanton. Loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2000)*, pages 327–336, June 2000.

[6] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS 98-4, Johns Hopkins University, Center for Networking and Distributed Systems, 1998.

[6] We are aware of the fact that the database models employed have different assumptions and limitations, but each method could be adapted in order to serve a reasonable variety of modern applications

[7] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 494–503, Vienna, Austria, July 2002. IEEE.

[8] CAIRN. http://www.cairn.net.

[9] DB2. http://www.ibm.com/software/data/db2/.

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

[11] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996. ACM-SIGMOD.

[12] J. N. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[13] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[14] Informix. http://www.informix.com.

[15] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.

[16] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 477–484. IEEE, July 2002.

[17] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1994.

[18] I. Keidar and D. Dolev. Increasing the resilience of atomic commit at no additional cost. In *Symposium on Principles of Database Systems*, pages 245–254, 1995.

[19] I. Keidar and R. Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*. IEEE, April 2000.

[20] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A client-server oriented algorithm for virtually synchronous group membership in WANs. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 356–365. IEEE, April 2000.

[21] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the* 26th *International Conference on Very Large Databases (VLDB)*, Cairo, Egypt, Sept. 2000.

[22] B. Kemme, A. Bartoli, and O. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the Internationnal Conference on Dependable Systems and Networks*, Göteborg, Sweden, 2001.

[23] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[24] L. Lamport. Private communication, talk at the future directions of distributed computing, June 2002.

[25] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *International Conference on Distributed Computing Systems*, pages 56–65, 1994.

[26] MySQL. http://www.mysql.com/doc/r/e/replication.html.

[27] Oracle. http://www.oracle.com.

[28] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proceedings of* 14th *International Symposium on DIStributed Computing (DISC'2000)*, 2000.

[29] PGReplicator. http://pgreplicator.sourceforge.net.

[30] Postgres-R. http://gborg.postgresql.org/project/pgreplication/projdisplay.php.

[31] PostgreSQL. http://www.postgresql.com.

[32] Spread. http://www.spread.org.

[33] Sybase. http://www.sybase.com.

[34] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr. 1996.