On the Performance of Consistent Wide-Area Database Replication

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

Technical Report
CNDS-2003-3
http://www.cnds.jhu.edu

December 10th, 2003

Abstract

In this paper we design a generic, consistent replication architecture that enables transparent database replication and we present the optimizations and tradeoffs of the chosen design. We demonstrate the practicality of our approach by building a prototype that replicates a PostgreSQL database system. We provide experimental results for consistent wide-area database replication. We claim that the use of an optimized synchronization engine is the key to building a practical synchronous replication system for wide-area network settings.

1 Introduction

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.

The apparent solution to these problems would be to consistently 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, but 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

*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

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. Furthermore, a local cluster cannot provide fault-tolerance in the presence of catastrophic failures that may affect an entire location. Wide-area database replication, coupled with a mechanism to direct the clients to the best available server (network- and server-wise) [1] can greatly enhance both the response time and availability. However, 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.

1.1 Our Contribution

In this paper, we introduce an optimized architecture for consistent replication[1] in local and wide-area networks. The architecture provides peer replication, where all the replicas serve as master databases that can accept both updates and queries. The supported failure model includes network partitions and merges, computer crashes and recoveries, and message omissions, all of which are handled by our system.

The replication system can be employed as a black box to replicate a database system without modifying the database and can be used transparently by a significant number of applications. We investigate the limitations imposed by this design and we show how it can be adapted to alleviate these limits and provide even more 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 system, but also facilitates interoperability between different database systems.

To validate our architecture design, we have developed a prototype system that replicates a PostgreSQL database system. The prototype uses the Spread group communication toolkit [2, 3] and the replication algorithm presented in [4]. Our prototype is able to replicate seamlessly existing applications that use standards such as JDBC or ODBC. We evaluate the architecture's performance both in local area and wide-area network settings, investigating the impact of latency, disk operation cost, and query versus update mix on the overall replication performance. The results help identifying the update synchronization algorithm as the bottleneck of a synchronous replicated system. To our knowledge, our results are the first to show that consistent database replication can be practical for wide-area network settings, being able to maintain a throughput that is sufficient for a large number of modern applications. We show that high latency is not an inherent obstacle in achieving wide-area transaction throughput similar to that achieved in local-area clusters.

The remainder of the paper is organized as follows: Section 2 surveys the related work. Section 3 describes our proposed transparent synchronous replication architecture while Section 4 details its main components and discusses the design optimizations and tradeoffs. In section 5 we evaluate the performance of our system. Section 6 concludes the paper.

2 Related Work

Despite their inefficiency and lack of scalability, two-phase commit protocols [5] remain the principal technique used by most commercial database systems that provide synchronous peer replication.

[1] Throughout the paper we use the terms consistent replication and synchronous replication interchangeably as they reflect the nomenclature used in the distributed and database communities, respectively.

The Accessible Copies algorithms [6, 7] 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, as a view change requires the servers 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 entire 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 [8], supports both asynchronous and synchronous replication. However, the former requires some level of application decision for conflict resolution, while the latter requires that all the replicas in the system are available to be able to function, making it impractical. Informix [9], Sybase [10] and DB2 [11] support only asynchronous replication which again ultimately relies on the application for conflict resolution.

In the open-source database community, two database systems have emerged as leaders: MySQL [12] and PostgreSQL [13]. By default both systems only provide limited master-slave replication capabilities. Other projects exist that provide more advanced replication methods for PostgreSQL such as Postgres Replicator, which uses a trigger-based store and forward asynchronous replication method [14].

The most evolved of these approaches is Postgres-R [15], a project that combines open-source expertise with academic research. Postgres-R implements algorithms designed by Kemme and Alonso [16] into the PostgreSQL database manager in order to provide synchronous replication. The current version also uses Spread [2, 17] as the underlying group communication system and focuses on integrating the method with version 7.2 of the PostgreSQL system. However, the existing implementation only provides master-slave replication.

In [16] the authors 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 our method [4], 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. [16] also presents performance results of a PostgreSQL replicated database prototype (Postgres-R). In [18], 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 [19, 20] 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 [4], 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. [21] presents an enhanced online

reconfiguration technique for the algorithms described above, that allows parallel recovery of nodes. The algorithm is based on a primary component membership model according to which only nodes in the primary component make progress, while nodes separated from the primary component are considered failed until they rejoin the primary component. In contrast, our solution allows multiple components to make progress by exchanging information and applying updates for which the commit order has already been determined.

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.

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 [22, 23, 24, 3] 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 [25], Ensemble [26], and Spread [17].

3 A Transparent Synchronous Replication Architecture

We present an architecture that provides transparent peer replication, supporting diverse application semantics. At the core of the system we use the synchronization algorithm introduced in [4]. 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 be augmented to support more generic transaction types. In the 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 1).

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

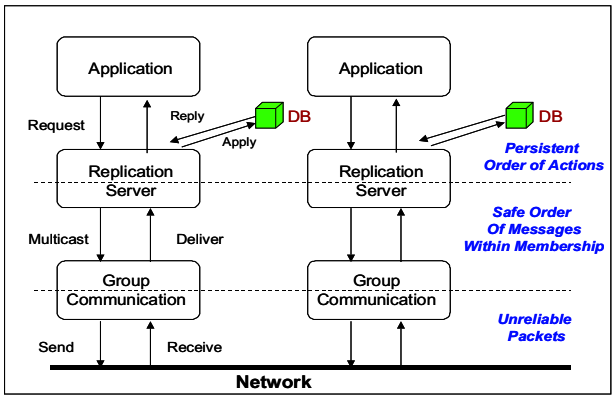

Figure 1: Synchronous Database Replication Architecture

help reach an agreement about the final global order of the set of actions.

In a typical scenario, 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 [27]. For this work, we are particularly interested in the Safe Delivery property of this model. Delivering a message according to Safe Delivery requires the group communication toolkit both to determine the total order of the message and to know that every other daemon in the membership already has the message. The careful use of Safe Delivery and Extended Virtual Synchrony allows us to eliminate end-to-end acknowledgments on a per-action basis. As long as no membership change takes place, the system eventually reaches consistency. End to end acknowledgements and state synchronization are only needed once a membership change takes place. Moving the acknowledgements from the application level (end-to-end) to the group communication level provides a fundamental performance increase since the synchronization between end points will be determined exclusively by the speed of the network without adding the extra overhead of application synchronization (extra IPC, writes to disk, etc.). Furthermore, by aggregating acknowledgements, the group communication layer additionally increases the efficiency of this phase.

We chose Spread [2] 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 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 [18].

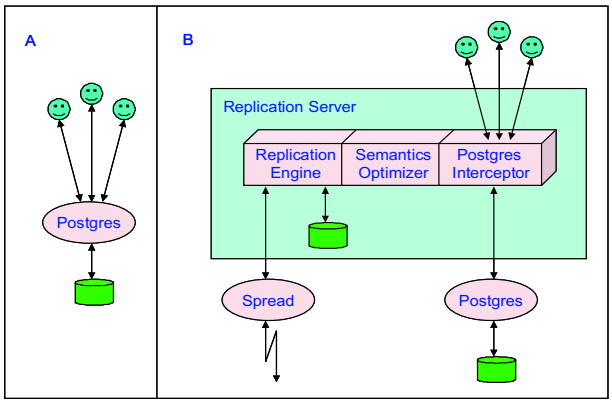

Figure 2: Transparent PostgreSQL Replication Architecture

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 system that uniquely identifies one of the connected components as the *primary* component of the system. Examples of quorum systems include monarchy, majority and dynamic linear voting [28]. Updates can continue to be committed only in the primary component, while the other components continue to remain active and answer queries consistently.

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. We will dedicate the following subsection to explaining the optimizations we implemented in the Safe Delivery mechanism of the Spread Toolkit. 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 2.A presents a non-replicated database system that is based on the PostgreSQL database manager. Figure 2.B. presents the building blocks of our implementation, replicating the PostgreSQL database system. The building blocks include a replication server and the Spread group communication toolkit. The database clients see the system as in figure 2.A., and are not aware of the replication although they access the database through our replication server. Similarly, any instance 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 2.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 [4, 29].

- A Semantics Optimizer that can decide whether to replicate transactions and when to apply them based on application semantics (if they are specified), on the actual content of the transaction, and on 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 PostgreSQL, we created a PostgreSQL specific interceptor. Existing applications can transparently use our interceptor layer to provide them with an interface identical to the PostgreSQL interface, while the PostgreSQL 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 Design Details and Considerations

4.1 Scalable Safe Delivery for Wide Area Networks

The architecture and reliability algorithms of the Spread toolkit [3, 17], provide basic services for reliable, FIFO, total order and Safe delivery. However, our architecture requires a high-performance implementation of the Safe Delivery service for wide-area networks, something not developed in previous work.

Spread uses a three level hierarchy, in which a set of daemons provide networking services for client applications. The Spread daemons are organized in sites, where a site represents one or more daemons connected through a local area network. Each site chooses a representative that is responsible for forwarding messages to and from other sites.

Delivering a message according to Safe Delivery requires the group communication toolkit to determine the total order of the message and to know that every other daemon in the membership already has the message before delivering it to the application. The latter property (sometimes called message stability) is traditionally implemented by group communication systems for garbage collection purposes, and therefore is usually not optimized.

The straightforward way to achieve message stability would require all the daemons to immediately acknowledge each Safe message. These acknowledgements are sent to all of the daemons, thus reaching all the possible destinations of the original Safe message within the time of one network diameter, which leads to a message latency of twice the network diameter. However, this algorithm is obviously not scalable. For a system with N sites, each message requires N broadcast acknowledgements, leading to N times more control messages than the data messages.

Our approach avoids this problem by aggregating information into cumulative acknowledgements. However, minimizing the bandwidth at all costs will cause extremely high latency for Safe messages, which is also undesirable. Therefore our approach permits tuning the tradeoff between bandwidth and message latency.

The structure of our acknowledgements, referred to as an *ARU_update* (all received up-to), is as follows when originating at a certain site A:

Site_Sequence is the sequence number of the last message originated by any daemon at site A and forwarded in order to other sites (Spread may forward messages even out of order to other sites to optimize performance). This number guarantees that no messages will be originated in the future from site A with a lower sequence number.

Site_Lts is the Lamport timestamp [30] that guarantees that site A will never send a message with a lower Lamport timestamp and a sequence number higher then *Site_Sequence*.

Site_Aru is the Lamport timestamp such that all the messages with a lower Lamport timestamp that originated from any site are already received at site A.

Each daemon keeps track of these three values received from each of the currently connected sites. In addition, each daemon updates a local variable, *Global_Aru*, which is the minimum of the *Site_Aru* values received from all of the sites. This represents the Lamport timestamp of the last message received in order by all the daemons in the system. A Safe message can be delivered to the local application (the replication server, in our case) when its Lamport timestamp is smaller or equal to the *Global_Aru*.

In order to achieve minimum latency, an *ARU_update* message should be sent immediately upon a site receiving a Safe message at any of the daemons in the site. However, if one *ARU_update* is sent for every message by every site, the traffic on the network will increase linearly with the number of sites. Spread optimizes this by trading bandwidth for improved latency when the load of messages is low, and by sending the *ARU_update* after a number of messages have been received when the message load is high. The delay between two consecutive *ARU_update* messages is bounded by a certain timeout *delta*. For example, if five Safe messages are received within one *delta* time, only one *ARU_update* will be sent for an overhead of 20%, however, if a message is received and no *ARU_update* has been sent in the last *delta* interval, then an *ARU_update* is sent immediately. Note that this is a simplification of the actual implementation which piggybacks control information on data messages.

In practical settings, *delta* will be selected higher than the network diameter *Dn*, which leads to a Safe Delivery latency between $3 * Dn$ and $2 * delta + 2 * Dn$.

The above latency bounds could be optimized by including in the *ARU_update* the complete table with information about **all** of the currently connected sites, instead of just the three local values described above. This would reduce the Safe Delivery latency to be between $2 * Dn$ and $delta + 2 * Dn$. However, such a scheme is not scalable, since the size of the *ARU_update* will grow linearly with the number of sites. For a small number of sites (e.g. 10), this technique could be useful, but in order to preserve the scalability of the system (e.g. 100's sites) we used in our experiments the more scalable method described above that leads to a message latency between $3 * Dn$ and $2 * delta + 2 * Dn$.

4.1.1 Example of Safe Delivery in Action

Table 1: Scalable Safe Delivery for Wide Area Networks

seq	lts	aru	seq	lts	aru	seq	lts	aru	seq	lts	aru
0	0	0	0	0	0	0	1	0	0	1	1
0	0	0	0	0	0	0	1	0	0	1	1
0	0	0	0	0	0	0	1	0	0	1	1
0	0	0	0	1	0	0	1	1	0	1	1
0	0	0	0	0	0	0	1	0	0	1	1
Global_Aru: 0			Global_Aru: 0			Global_Aru: 0			Global_Aru: 1		
(a)			(b)			(c)			(d)		

Let's consider a network of five daemons N_1– N_5 where each of the daemons represents a site. Table 1 shows how the Safe Delivery mechanism works at daemon N_4, when a Safe message is sent by N_1, considering the worst case scenario when the latency from N_1 to N_4 is the diameter of the network. Initially all the daemons have all their variables initialized with zero (Table 1.a).

Upon receiving the Safe message from N_1, the daemon N_4 updates its *Site_lts* (Table 1.b). Assuming there are no losses and no other messages in the system, all the daemons will behave

similarly, updating their *Site_lts* value. It takes one network diameter *Dn* for the message to get from N_1 to all the other daemons.

After at most a *delta* interval, every daemon sends an *ARU_update* containing the row representing themselves in their matrix. Depending on the time each daemon waits until sending its Aru update, the daemons will receive information from all the other daemons in between *Dn* and *delta* + *Dn* time. Upon receiving these updates, daemon N_4 knows that all of the daemons increased their *Site_lts* to 1, and since it did not detect any loss, it can update its *Site_Aru* to 1 (Table 1.c). Similarly, all the other daemons will update their *Site_Aru* values to 1, assuming there are no losses. At this point, N_4 knows that no daemon will create a message with a Lamport timestamp lower than 1 in the future, so according to total order delivery, it could deliver this message to the upper layer. However, this is not enough for Safe Delivery; N_4 does not know yet whether all of the daemons received all of the *ARU_updates*.

Already, at least *Dn* time has elapsed at N_1 (the farthest daemon from N_4) between the time it sent its last *Aru_update* and the time N_4 got it. Therefore, after waiting at most *delta* – *Dn* time, N_1 (as well as the other daemons) can send another *ARU_update* containing their *Site_Aru* value advanced to 1. Finally, after one more *Dn* time, when N_4 receives all these *Aru_updates*, it can advance its *Global_Aru* to 1 (Table A1.d), and deliver the Safe message. The total latency in the worst case is *Dn* + (*delta* + *Dn*) + ((*delta* - *Dn*) + *Dn*), which is equal to 2 * *delta* + 2 * *Dn*.

Note that *ARU_updates* are periodic and cumulative, and as more Safe messages are sent in a *delta* interval, this delay will be amortized between different messages. However, the expected latency for a Safe message is at least 3 * *Dn*, as the delivery mechanism includes three rounds in this scalable approach.

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 PostgreSQL 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 [16] 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 [20] 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. [20] 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. This approach also permits the use of row replication, where a transaction is executed on just one database and its effect is replicated to the other servers. This reduces the load on each database server but may increase significantly the network load as the modified row data can be significantly larger than the standard SQL 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 an 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 the synchronization module.

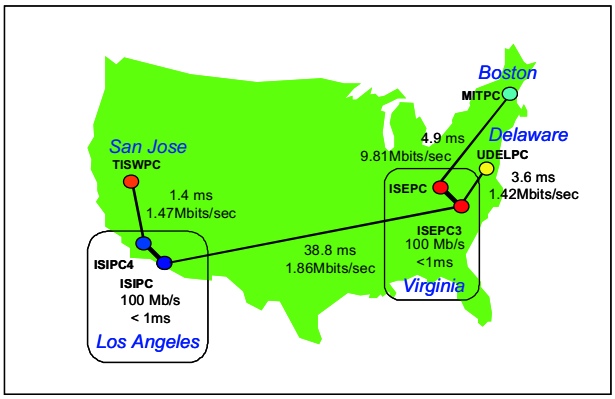

Figure 3: Layout of the CAIRN Network

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 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.

5 Prototype Performance Evaluation

We evaluated the performance of the PostgreSQL replication prototype in two environments. A local area cluster and the Emulab wide-area testbed [31]. The cluster contains 14 Linux Dual PIII 667 computers with 256 Mbytes memory and 9G SCSI disks. Emulab[2] (the Utah Network Testbed) provides a configurable test-bed where the latency, throughput and link-loss characteristics of each of the links can be controlled. The configured network is then emulated in the Emulab local area network, using actual routers and in-between computers that enforce the required latency, loss and capacity constraints. We used 7 Emulab Linux computers to emulate the real CAIRN [32] network depicted in Figure 3. Each of the Emulab machines is a PIII 850 with 512Mbytes and 40G IDE disk.

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 PostgreSQL database approximately 120 update transactions or 180 read-only transactions per second, as defined below. The

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

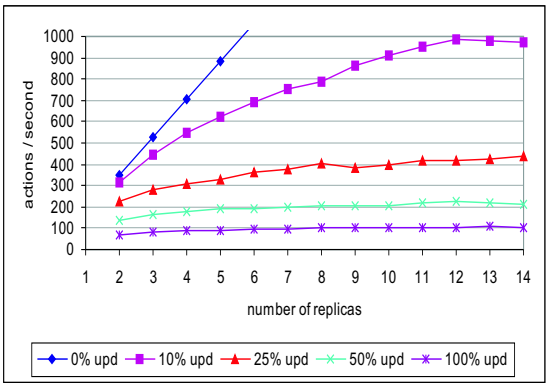

Figure 4: LAN Postgres Throughput under Varying Number of Replicas

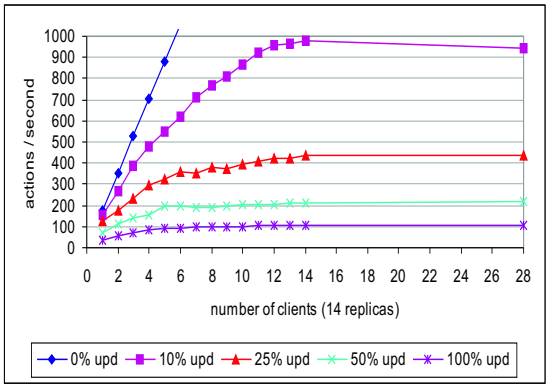

Figure 5: LAN Postgres Throughput under Varying Number of Clients

computers used in the wide-area experiment can perform on a non-replicated PostgreSQL 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 similar to that introduced by [16, 20].

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 PostgreSQL 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.

5.1 Local Area Performance Evaluation

In [4] we benchmarked the replication engine used by our architecture in the same local area network setup used in this experiment. We compared our replication engine with another group communication based method and with an upper-bound two phase commit implementation and noticed that our engine outperforms the other solutions by up to an order of magnitude, reaching up to 1000 updates per second. In this work we analyze the overall performance of the PostgreSQL replication prototype.

The first experiment conducted with our new architecture tested the scalability of the system as the number of replicas of the database increases. Each replica executed on a separate computer

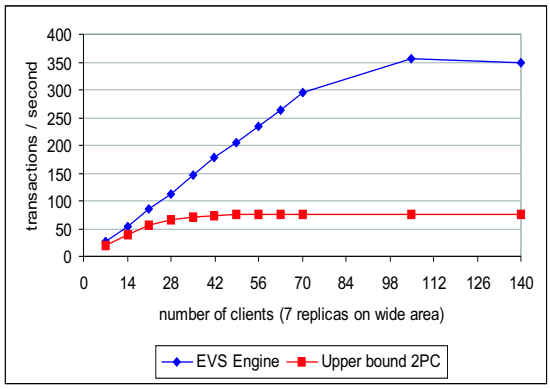

Figure 6: WAN Synchronization Engine Comparison

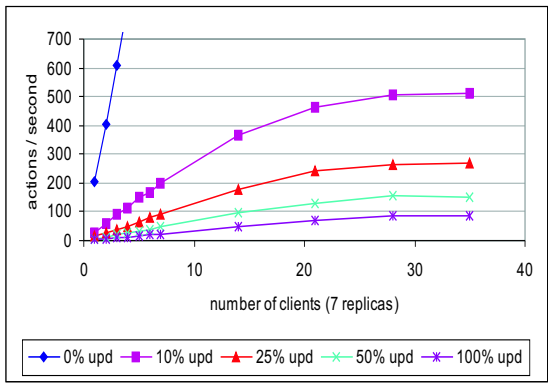

Figure 7: WAN PostgreSQL Throughput with varied update/query load

with one local client over a local area network. Figure 4 shows five separate experiments. Each experiment used 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 as presented in the previous subsection. Further analysis of the difference between the synchronization engine potential performance and the observed prototype performance is presented in the following subsection discussing the wide-area experimentation.

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 fixed the number of replicas at 14, one replica on each computer on the local area network. The number of clients connected to the system increased from 1 to 28, evenly spread among the replicas. In Figure 5 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.

5.2 Wide-Area Performance Evaluation

In order to visualize the performance gain introduced by our synchronization algorithm through the use of optimized Safe Delivery and avoiding end-to-end acknowledgements, we first compared the performance of our replication engine, using full disk syncing in order to guarantee data consistency,

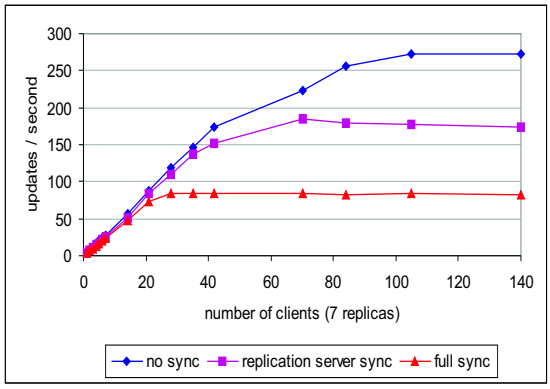

Figure 8: Impact of forced sync to disk on throughput

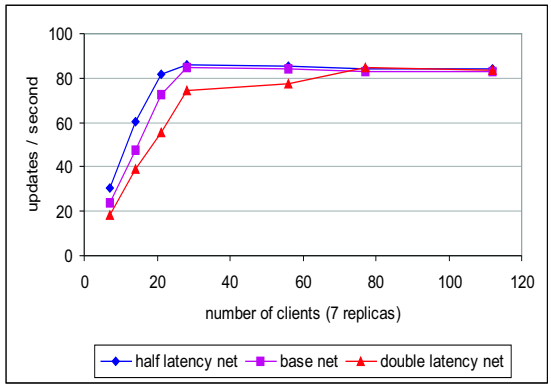

Figure 9: Throughput under varied network latency

against an *upper-bound*[3] two-phase commit algorithm (2PC). (Figure 6) The upper-bound 2PC engine reaches its maximum capacity at approximately 80 actions per second, which is consistent with the performance reported in similar setups by commercial databases that use 2PC at the core of their synchronous replication method. Our replication engine is saturated at 350 actions per second due to the optimizations mentioned in the previous section.

The wide area experiment conducted on Emulab measured the throughput under a 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 7. 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.

Notice that the throughput achieved for update replication is not close to the performance exhibited by the synchronization algorithm (Figure 6). One of the difficulties in conducting database experiments is that real production database servers are very expensive and are not always available for academic research. We conducted our experiments on standard, inexpensive Intel PCs whose disk and memory performance is significantly poorer and that lack specialized hardware such as flash RAM logging disks. To evaluate the potential performance of our system on such hardware we reran the same experiment as in Figure 7 for the 100% updates scenario, under varied disk sync conditions (Figure 8. When only the Replication Server uses forced syncs to disk, disabling this feature in the database, the system achieves a maximum throughput of 183 updates per second (with 70 clients). When synchronous disk sync is disabled for both PostgreSQL and the replication engine, a maximum throughput of 272 updates per second is reached (with 105 clients in the system) which is close to the potential exhibited by the replication engine alone.

Finally, to test the impact of network latency on performance, we used Emulab to construct two additional networks, identical in topology to our original network, but with either one half or double the latency on each link. In effect, this explores the performance of the system as the diameter of the network changes. The original network has a diameter of about 45ms (typical for a network spanning the US), and the additional networks have about 22ms and 90ms diameters

[3]Our 2PC implementation will assume that all the locks are granted instantly, thus ensuring the maximum level of concurrency the 2PC method can support.

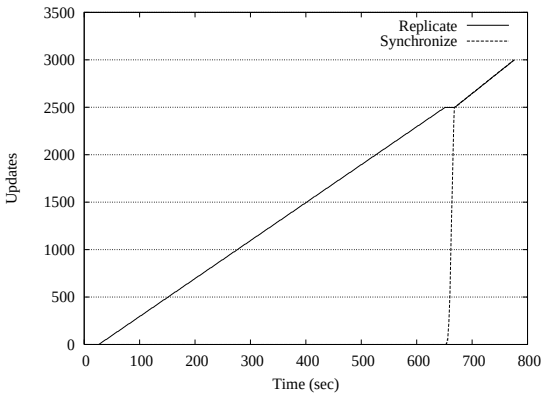

Figure 10: Replication and recovery of 2500 updates

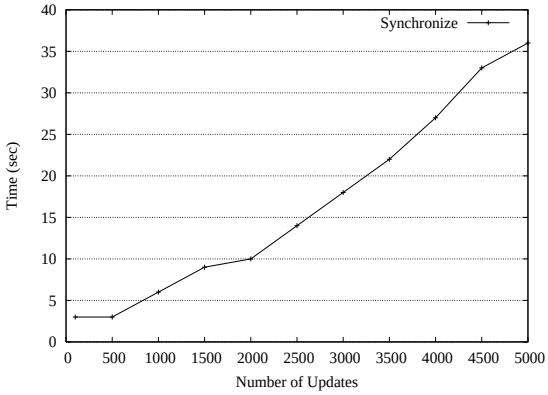

Figure 11: Time to synchronize with varying number of updates

respectively. Figure 9 illustrates that under any of these diameters, the system can achieve the same maximum throughput of 85 updates per second. However, as the diameter increases, more clients are required to achieve the same throughput.

It is difficult to directly compare our results with those presented in [16, 20] due to significant difference in the hardware platforms. Our PostgreSQL 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 high throughput rates, despite giving up on additional gains through use of concurrent transaction execution or semantic knowledge about the application. The throughput that our prototype was able to sustain would be sufficient for a large number of applications while maintaining a reasonably low response time even in wide-area settings.

5.3 Partition behaviour

In order to evaluate the performance of our prototype in the presence of temporary network failures (partitions/remerges) we ran the following experiment. On the wide area Emulab setup we partitioned off the node MITPC in Boston from ISEPC in Virginia. While the node is partitioned we submit a varying number of updates into the database, to the 6 remaining connected sites. We then reconnect the MITPC node to the rest of the system and record the time necessary for it to be updated with the data it missed. In parallel, we run clients that query the databases every second, and check how many updates have been applied. We record the progress noted by these clients.

In figure 10 we generate updates into the system at a steady rate of 4 updates per second. These updates are replicated among the six connected replicas. After generating 2500 updates, we reconnect the 7th node to the rest of the replicas, bring it up to date and then continue submitting updates at the same rate as before. It can be noted that the time needed for the MITPC node to catch up to the other nodes is 17 seconds, only a fraction of the time that was needed to replicate the updates at the given steady rate.

Figure 11 plots the synchronization time after remerge from the previous graph while we change the number of updates applied during the partition time from 100 to 5000. We note a linear increase of the synchronization time with the number of outstanding updates. This shows a constant time per synchronized update regardless of the synchronization load.

6 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 notable on the local area network and even more so 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 wide-area database replication.

References

[1] A. Peterson Y. Amir and D. Shaw, "Seamlessly selecting the best copy from internet-wide replicated web servers," in *International Symposium on Distributed Computing (DISC98)*, 1998, pp. 22–23.

[2] Spread, "http://www.spread.org," .

[3] 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)*, June 2000, pp. 327–336.

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

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

[6] A. El 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*. ACM, March 1985, pp. 215–229.

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

[8] Oracle, "http://www.oracle.com," .

[9] Informix, "http://www.informix.com," .

[10] Sybase, "http://www.sybase.com," .

[11] DB2, "http://www.ibm.com/software/data/db2/," .

[12] MySQL, "http://www.mysql.com/doc/r/e/replication.html," .

[13] PostgreSQL, "http://www.postgresql.com," .

[14] PGReplicator, "http://pgreplicator.sourceforge.net," .

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

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

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

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

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

[20] 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)*. IEEE, July 2002, pp. 477–484.

[21] R. Jimenez-Peris, M. Patino-Martinez, and G. Alonso, "An algorithm for non-intrusive, parallel recovery of replicated data and its correctness," in *Proceedings of* 21^{st} *IEEE Int. Conf. on Reliable Distributed Systems (SRDS'2002)*, 2002, pp. 150–159.

[22] 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*. IEEE, April 2000, pp. 356–365.

[23] 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.

[24] D.A. Agarwal, L.E. Moser, P.M. Melliar-Smith, , and R.K. Budhia, "The totem multiple-ring ordering and topology maintenance protocol," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 93–132, May 1998.

[25] R. van Renesse, K. P. Birman, and Silvano Maffeis, "Horus: A flexible group communication system," *Communications of the ACM*, vol. 39, no. 4, pp. 76–83, Apr. 1996.

[26] M. Hayden, *The Ensemble System*, Ph.D. thesis, Cornell University, 1998.

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

[28] S. Jajodia and D. Mutchler, "Dynamic voting algorithms for maintaining the consistency of a replicated database," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 230–280, 1990.

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

[30] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[31] "B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *OSDI02*, Boston, MA, Dec 2002, USENIXASSOC, pp. 255–270.

[32] CAIRN, "http://www.cairn.net," .