Technical Report CNDS-2002-1 Johns Hopkins University, http://www.cnds.jhu.edu/publications

Practical Wide-Area Database Replication[1]

Yair Amir[*], Claudiu Danilov[*], Michal Miskin-Amir[†], Jonathan Stanton[*], Ciprian Tutu[*]

[*] Johns Hopkins University
Department of Computer Science
Baltimore MD 21218
{yairamir, claudiu, jonathan, ciprian}@cnds.jhu.edu

[†] Spread Concepts LLC
5305 Acacia Ave
Bethesda MD 20814
michal@spreadconcepts.com

Abstract

This paper explores the architecture, implementation and performance of a wide and local area database replication system. The architecture provides peer replication, supporting diverse application semantics, based on a group communication paradigm. Network partitions and merges, computer crashes and recoveries, and message omissions are all handled. Using a generic replication engine and the Spread group communication toolkit, we provide replication services for the PostgreSQL database system. We define three different environments to be used as test-beds: a local area cluster, a wide area network that spans the U.S.A, and an emulated wide area test bed. We conduct an extensive set of experiments on these environments, varying the number of replicas and clients, the mix of updates and queries, and the network latency. Our results show that sophisticated algorithms and careful distributed systems design can make symmetric, synchronous, peer database replication a reality for both local and wide area networks.

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, the location of the data can have a significant impact on application response time and availability. A centralized approach manages only one copy of the database. This approach is simple since contradicting views between replicas are not possible. The centralized approach suffers from two major drawbacks:

- Performance problems due to high server load or high communication latency for remote clients.
- Availability problems caused by server downtime or lack of connectivity. Clients in portions of the network that are temporarily disconnected from the server cannot be serviced.

The server load and server downtime problems can be addressed by replicating the database servers to form a cluster of peer servers that coordinate updates. However,

[1] This work was partially funded by grant F30602-00-2-0550 from the Defense Advanced Research Projects Agency (DARPA). The views expressed in this paper are not necessarily endorsed by DARPA.

communication latency and server connectivity remain a problem when clients are scattered on a wide area network and the cluster is limited to a single location. Wide area database replication coupled with a mechanism to direct the clients to the best available server (network-wise and load-wise) [APS98] can greatly enhance both the response time and availability.

A fundamental challenge in database replication is maintaining a low cost of updates while assuring global system consistency. The problem is magnified for wide area replication due to the high latency and the increased likelihood of network partitions in wide area settings.

In this paper, we explore a novel replication architecture and system for local and wide area networks. We investigate the impact of latency, disk operation cost, and query versus update mix, and how they affect the overall performance of database replication. We conclude that for many applications and network settings, symmetric, synchronous, peer database replication is practical today.

The paper focuses on the architecture, implementation and performance of the system. The architecture provides peer replication, where all the replicas serve as master databases that can accept both updates and queries. The failure model includes network partitions and merges, computer crashes and recoveries, and message omissions, all of which are handled by our system. We rely on the lower level network mechanisms to handle message corruptions, and do not consider Byzantine faults.

Our replication architecture includes two components: a wide area group communication toolkit, and a replication server. The group communication toolkit supports the Extended Virtual Synchrony model [MAMA94]. The replication servers use the group communication toolkit to efficiently disseminate and order actions, and to learn about changes in the membership of the connected servers in a consistent manner.

Based on a sophisticated algorithm that utilizes the group communication semantics [AT01, A95], the replication servers avoid the need for end-to-end acknowledgements on a per-action basis without compromising consistency. End-to-end acknowledgments are only required when the membership of the connected servers changes due to network partitions, merges, server crashes and recoveries. This leads to high system performance. When the membership of connected servers is stable, the throughput of the system and the latency of actions are determined mainly by the performance of the group communication and the single node database performance, rather than by other factors such as the number of replicas. When the group communication toolkit scales to wide area networks, our architecture automatically scales to wide area replication.

We implemented the replication system using the Spread Group Communication Toolkit [AS98, ADS00, Spread] and the PostgreSQL database system [Postgres]. We then define three different environments to be used as test-beds: a local area cluster with fourteen replicas, the CAIRN wide area network [CAIRN] that spans the U.S.A with seven sites, and the Emulab emulated wide area test bed [Emulab].

We conducted an extensive set of experiments on the three environments, varying the number of replicas and clients, and varying the mix of updates and queries. Our results show that sophisticated algorithms and careful distributed systems design can make symmetric, synchronous, peer database replication a reality over both local and wide area networks.

The remainder of this paper is organized as follows. The following subsection discusses related work. Section 2 presents our architecture for transparent database replication. Section 3 presents the Spread group communication toolkit and the optimization we implemented to support efficient wide area replication. Section 4 details our replication server. Section 5 presents the experiments we constructed to evaluate the performance of our system, and Section 6 concludes the paper.

Related Work

Despite their inefficiency and lack of scalability, two-phase commit protocols [GR93] remain the principal technique used by most commercial database systems that try to provide synchronous peer replication. Other approaches investigated methods to implement efficient lazy replication algorithms using epidemic propagation [Demers87, HAE00].

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 [Oracle], 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 [Informix], Sybase [Sybase] and DB2 [DB2] support only asynchronous replication, which again ultimately rely on the application for conflict resolution.

In the open-source database community, two database systems emerge as clear leaders: MySQL [Mysql] and PostgreSQL [Postgres]. 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 [Pgrep].

The more evolved of these approaches is Postgres-R [Postgres-R], a project that combines open-source expertise with academic research. This work implements algorithms designed by Kemme and Alonso [KA00] into the PostgreSQL database manager in order to provide synchronous replication. The current work focuses on integrating the method with the upcoming 7.2 release of the PostgreSQL system.

Kemme and Alonso introduce the Postgres-R approach in [KA00] and study its performance on Local Area settings. They use an eager-replication method that exploits group communication ordering guarantees to serialize write conflicts at all sites. The initial work was done on version 6.4.2 of PostgreSQL. In similar approaches, Jimenez-Peris and Patino-Martinez, together with Kemme and Alonso analyze various other methods and their performance in local area settings [JPKA00, PJKA00].

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 [KSMD00, KK00, AMMB98, ADS00] 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[RBM96], Ensemble[H98], and Spread[AS98]. The JGroups[Mon00] system provides an object-oriented group communication system, but its semantics differ in substantial detail from Extended Virtual Synchrony.

2. An Architecture for Transparent Database Replication

Our architecture provides peer replication, supporting diverse application semantics, based on a group communication paradigm. Peer replication is a symmetric approach where each of the replicas is guaranteed to invoke the same set of actions in the same order. In contrast with the common Master/Slave replication model, in peer replication each replica acts as a master. 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.

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. 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 among the servers group and to help reach an agreement about the final global order of the set of actions.

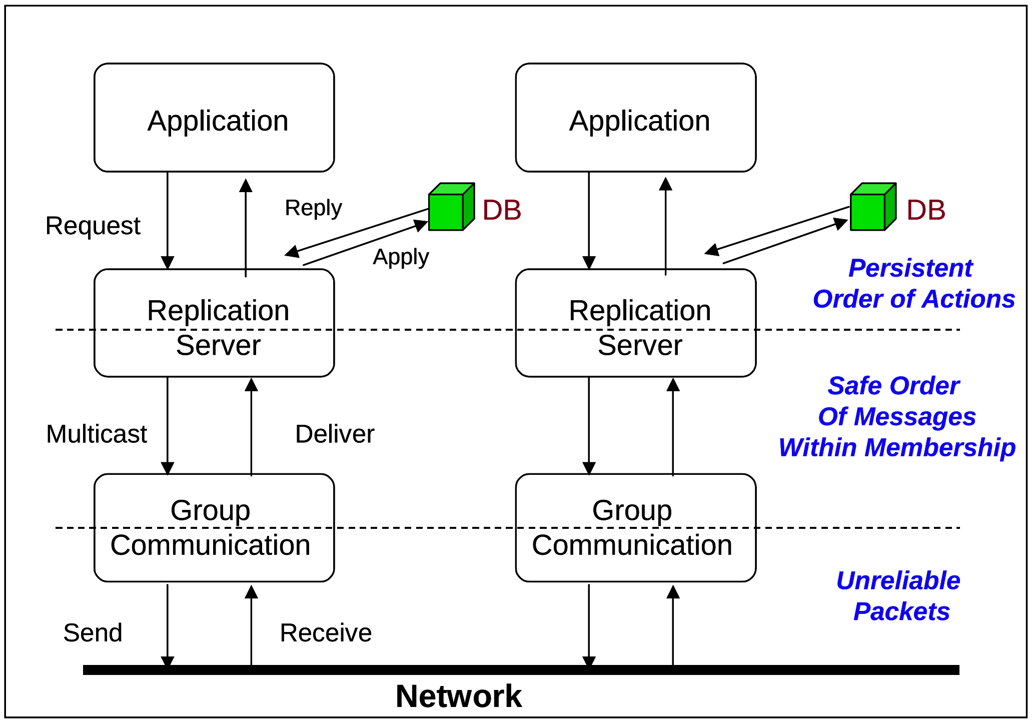

Figure 1: A Database Replication Architecture

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 [MAMA94]. 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 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. On notification 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 [KBB01].

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. A detailed description of the replication algorithm we are using is given in [AT01, A95].

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.

The latency and throughput of the system for updates is obviously highly dependent on the performance of the group communication Safe Delivery service. Read-only queries will not be sent over the network.

An important property 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).

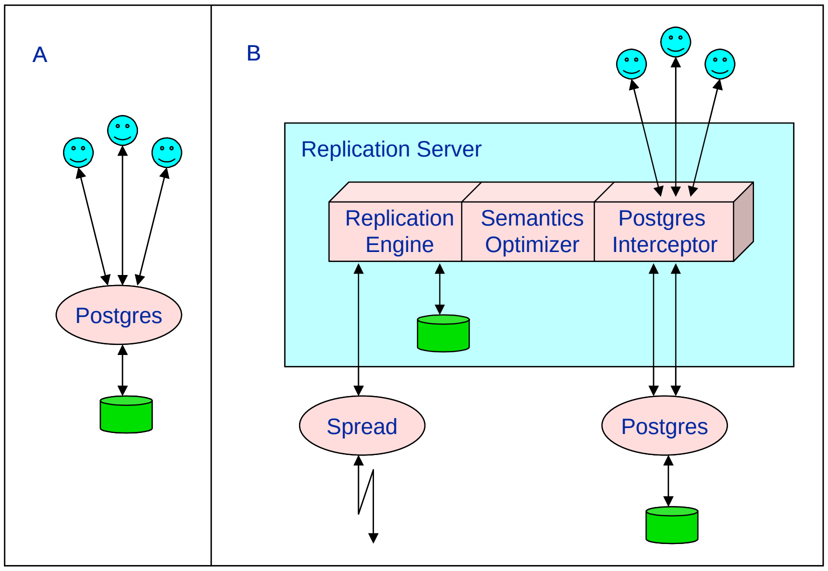

Figure 2: Modular Software Architecture

Figure 2.A presents a non-replicated database system that is based on the Postgres database manager. Figure 2.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 Postgres 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 Postgres 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 provably correct generic Replication Engine that includes all of the replication logic, 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 [AT01, A95].
- A Semantics Optimizer that decides whether to replicate transactions and when to apply them based on the required semantics, 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. 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.

To optimize performance, the replication server could be integrated with the database manager, allowing more accurate determination of which actions could be parallelized internally.

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.

3. The Spread Toolkit

Spread is a client-server group messaging toolkit that provides reliable multicast, ordering of messages and group membership notifications under the Extended Virtual Synchrony semantics [MAMA94] over local and wide-area networks to clients that connect to daemons.

To provide efficient dissemination of multicast messages on wide-area networks, all Spread daemons located in one local area network are aggregated into a group called a site, and a dissemination tree is rooted at each site, with other sites forming the nodes of the tree. Messages that originate at a Spread daemon in a site will first be multicast to all the daemons of the site. Then, one of those daemons will forward the message onto the dissemination tree where it will be forwarded down the tree until all of the sites have received the message. Therefore, the expected latency to deliver a message is the latency of the local area network plus the sum of the latencies of the links on the longest path down the tree.

The total order of messages is provided by assigning a sequence number and a Lamport time stamp to each message when it is created by a daemon. The Lamport time stamps provide a global partial causal order on messages and can be augmented to provide a total causal order by breaking ties based on the site identifier of the message [L78]. To know the total order of a message that has been received, a server must receive a message from every site that contains a Lamport time stamp at least equal to the Lamport time stamp of the message that is to be delivered. Thus, each site must receive a message from every other site prior to delivering a totally ordered message.

The architecture and reliability algorithms of the Spread toolkit [AS98, ADS00], and its global flow control [AADS01] 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.

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. The latter property (sometimes called message stability) was traditionally implemented by group communication systems for garbage collection purposes, and therefore was not optimized. For this work, we designed and implemented scalable, high performance wide-area Safe Delivery for Spread.

Scalable Safe Delivery for Wide Area Networks

A naive algorithm will have all the daemons 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 one network diameter. Overall, this leads to a 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 ack explosion 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 latency.

The structure of our acknowledgements, referred to as *ARU_updates* (all received up-to), is as follows for an *ARU_update* originating at 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 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 highest 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 another 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 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_updates* it 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*. Therefore, the Safe Delivery latency is between $3 * Dn$ and $2 * delta + 2 * Dn$.

This 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, this 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 we elected not to report results based on it in order to preserve the scalability of the system (e.g. 100 sites).

A detailed example of how Safe Delivery is performed, is provided in Appendix 1.

4. The Replication Server

A good group communication system is unfortunately not sufficient to support consistent synchronous peer database replication. The replication server bridges the gap. The replication server is constructed with three main modules, as depicted in Figure 2.B in Section 2. Each of the modules is responsible for the following tasks: the Replication Engine consistently and efficiently orders actions; the database interceptor manages the user connections and the replication server connections to the database; the Semantics Optimizer optimizes the handling of actions based on the required application semantics and the current connectivity. Below we discuss these modules as they are used to replicate the Postgres database.

The Replication Engine

The specifics of the Replication Engine and its algorithm are discussed elsewhere [AT01, A95] and are not part of this paper. However, it is important to summarize the properties of the engine in order to understand how the replication server uses it.

The Replication Engine implements a complete and provably correct algorithm that provides global persistent consistent order of actions in a partitionable environment without the need for end-to-end acknowledgements on a per action basis. End-to-end acknowledgements are only used once for every network connectivity change event such as network partition or merge, or server crash or recovery. The engine uses Spread as the group communication toolkit.

The Replication Engine minimizes synchronous disk writes: it requires only a single synchronous disk write per action at the originating replica, just before the action is multicast (see Figure 1) via the group communication. Any peer replication scheme will have to include at least this operation in order to cope with a failure model that includes crashes and recoveries as well as network partitions and merges, and to allow a recovered database to work without connecting to the primary component first.

The above properties lead to high performance of the replication server: the throughput and latency of actions while in the primary component are mainly determined by the Safe Delivery performance of the group communication and are less influenced by other factors such as the number of replicas and the performance of the synchronous disk writes. Of course, the performance of actually applying updates to the database depends on the quality of the database and the nature of the updates.

In the presence of network partitions, the replication servers identify at most a single component of the servers group as the primary component. The other components of the partitioned server group are non-primary components. While in the primary component, actions are immediately applied to the database upon their delivery by the group communication according to the Safe Delivery service. While in a non-primary component, they will be applied according to the decision made by the Semantic Optimizer, described below. Updates that are generated in non-primary components will propagate to other components as the network connectivity changes. These updates will be ordered in the final persistent consistent order upon the formation of the first primary component that includes them.

Every architecture that separates the replication functionality from the database manager needs to guarantee the *exactly once* semantics for updates. The difficulty occurs when the database manager crashes and the replication server does not know if the update was committed. One solution to this problem is to use the ordinal of the update assigned by the consistent persistent order. This ordinal will be stored in the database and updated as part of each update transaction. Upon recovery, the replication server can query the database for the highest ordinal transaction committed and re-apply actions with higher ordinal values. Since the database guarantees all-or-none semantics, this solution guarantees exactly once semantics.

The Replication Engine uses the Dynamic Linear Voting [JM90] quorum system to select the primary component. Dynamic Linear Voting allows the component that contains a weighted majority of the last primary component to form the next primary component. This quorum system performs well in practice [PL88], establishing an active primary component with high probability (better than weighted majority, for example).

The Replication Engine employs the propagation by eventual path technique, where replication servers that merge into a network component exchange their knowledge immediately upon the delivery of a membership notification. If the servers stay connected long enough to complete a re-conciliation phase, they share the same knowledge and will appear identical to clients. This technique is optimal in the sense that if no eventual path exists between two servers, no algorithm can share information between these servers. If such an eventual path exists between two servers, the Replication Engine will be able to propagate the information between them.

Employing the Dynamic Linear Voting quorum system and the propagation by eventual path techniques contributes to the high availability of our system.

The Postgres Interceptor

The interceptor module provides the link between the replication server and a specific database. Therefore, it is the only non-generic module of the architecture. The interceptor facilitates handling client connections, sends client messages to the database, gets back results from the database, and forwards the results back to the relevant clients.

Our interceptor was implemented for the Postgres version 7.1.3 database system. We intercept the clients that use the TCP interface to connect to the database. Once a client message is received, it is passed to the Semantics Optimizer that decides the next processing step, as described in the following subsection. The Postgres Interceptor reuses, with minor modifications, parts of the Postgres libpq library implementation in order to communicate with the database and the database clients.

In order to capture the client-database communication, the interceptor listens for client connections on the standard Postgres database port, while Postgres itself listens on a different private port, known only to the interceptor. The clients that attempt to connect to Postgres will transparently connect to the interceptor, which will take care of redirecting their queries to the database and passing back the response. The interceptor can handle multiple clients simultaneously, managing their connections and identifying transactions initiated by a local client from transactions initiated by a client connected to a different replica. This method can be applied to any database manager with a documented client-server communication protocol.

When the interceptor receives an action from the Replication Engine, it submits the action to the database. Under normal execution, Postgres creates one process for each client connected to the database, taking advantage of the parallelism of transaction execution when there are no conflicts between the transactions issued by different clients.

Our architecture however, has to make sure that the order assigned to the transactions by the replication engine is **not** modified by this parallelization. Since we do not control the Postgres scheduler, as our implementation resides entirely outside of the database, in order to maintain the established ordering we need to know which actions can be parallelized. Our current implementation, benchmarked in the next section, ensures a correct execution by maintaining a single connection to the database on each replica, serializing the execution of transactions.

The performance could be improved by implementing a partial SQL parser that can allow us to parallelize the execution of local queries. Any such action, that does not depend on an update transaction issued by the same client and that was not yet executed, can be sent to the database independently of other transactions submitted by other clients (local or remote). For this purpose, the interceptor can maintain a pool of connections to the database where one connection is used for all update transactions (and thus maintains their order) and the others are shared by all local independent queries. This optimization is not implemented yet in our system.

A lower-level integration of our replication server inside the database manager could exploit the same parallelization as the database exploits. Of course, that would incur the price of losing the database transparency.

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.

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 because the system distinguishes between queries and actions that also update the database. For this purpose 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. Our system allows each client to specify its required semantics individually, upon connecting to the system. Our system can even support such specification for each action but that will require the client to be aware of our replication service.

In addition to the strict consistency semantics and the standard weak consistency, our 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.

5. Performance Evaluation

We evaluated the performance of our system in three different environments.

- A local area cluster at our lab at Johns Hopkins University. The cluster contains 14 Linux computers, each of which has the characteristics described by the Fog machine in Figure 3.
- The CAIRN wide-area network [CAIRN]. We used seven sites spanning the U.S.A. as depicted in Figure 4. CAIRN machines generally serve as routers for networking experiments. As a consequence, many of the CAIRN machines are weak machines with slow disks and not a lot of memory. The characteristics of each of the different CAIRN machines used in our evaluation is described in Figure 3. Especially note the low Postgres performance achieved by some of these machines without replication.

- The Emulab wide-area test-bed [Emulab]. Emulab[2] (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, each has the characteristics described by the Emu machine in Figure 3. Most of our Emulab experiments emulated the CAIRN network depicted in Figure 4.

Our experiments were run using PostgreSQL version 7.1.3 standard installations. We use a database and experiment setup similar to that introduced by [KA00, JPKA00].

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 10MB. 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.

Machine	**Processor**	**Memory (MB)**	**HDD (GB)**	**Postgres Updates/sec**	**Postgres Queries/sec**
Local Area cluster					
Fog [1-14]	Dual PIII 667	256	9G SCSI	119.9	181.8
Cairn wide area testbed					
TISWPC	PII 400	128	4G IDE	37.26	77.49
ISIPC4	Pentium 133	64	6G IDE	25.44	11.97
ISIPC	PII 450	64	19G IDE	42.34	90.41
ISIEPC3	PII 450	64	6G IDE	50.10	93.41
ISIEPC	PII 450	64	19G IDE	43.52	92.72
MITPC2	Ppro 200	128	6G IDE	48.22	40.40
UDELPC	Ppro 200	128	4G SCSI	23.75	39.07
Emulab emulated wide-area testbed					
Emu[1-7]	PIII 850	512	40 G IDE	118.3	211.4

Figure 3: System Specification for the Three Test-beds

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

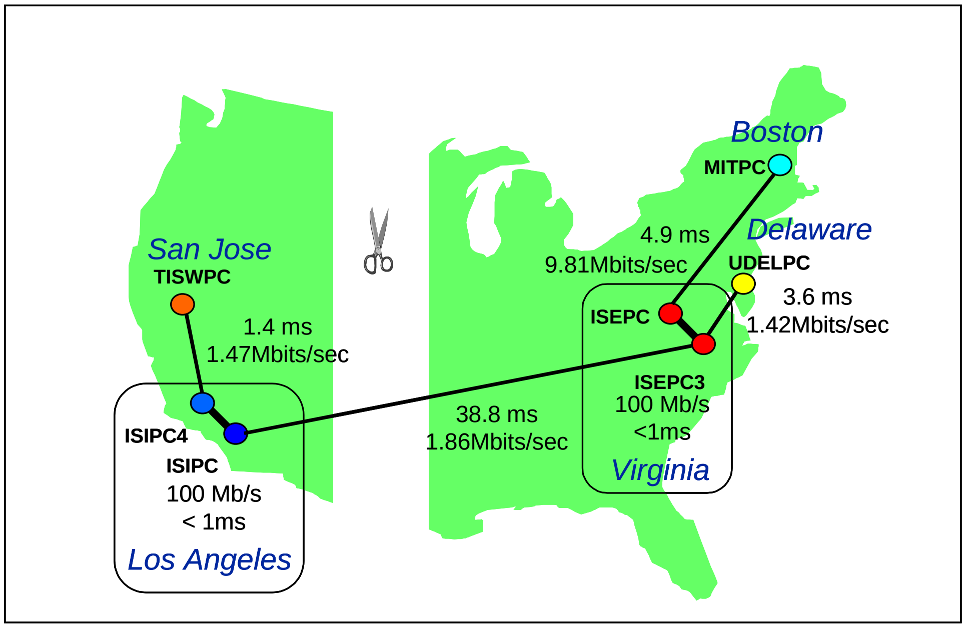

Figure 4: Layout of The CAIRN Test-bed

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 conducted several tests either with Postgres not syncing its writes to disk (forced writes that the OS must not cache), or with both Postgres and our Replication Server not syncing the writes to disk. In all of these tests we also report the full-sync version. All tests that do not specify *no-sync*, or *replication server sync*, were conducted with full data sync on all required operations.

Each client only submits one action (update or query) at a time. Once that action has completed, the client can generate a new action.

Note that **any** database replication scheme has to guarantee that any query can be answered **only after** the effects of all of the updates preceding the query are reflected in the database. If the replication scheme is transparent, all the above updates have to be applied to the database before answering the query. Therefore, any such scheme is limited by the native speed of the database to execute updates. In our experiments, the local Fog and Emulab machines are limited to about 120 updates/sec as can be seen in Figure 3. Therefore any transparent replication method is limited to less than 120 updates/sec in a similar setting. Using our scheme to replicate a database with better performance could achieve better results as is indicated by the *replication server sync* tests, since our replication scheme is not the bottleneck.

First, we evaluate our system over the local area cluster defined in Figure 3. The first experiment tested the scalability of the system as the number of replicas of the database increased. Each replica executes on a separate Fog machine with one local client. Figure 5 shows five separate experiments. Each experiment used a different proportion of updates to queries.

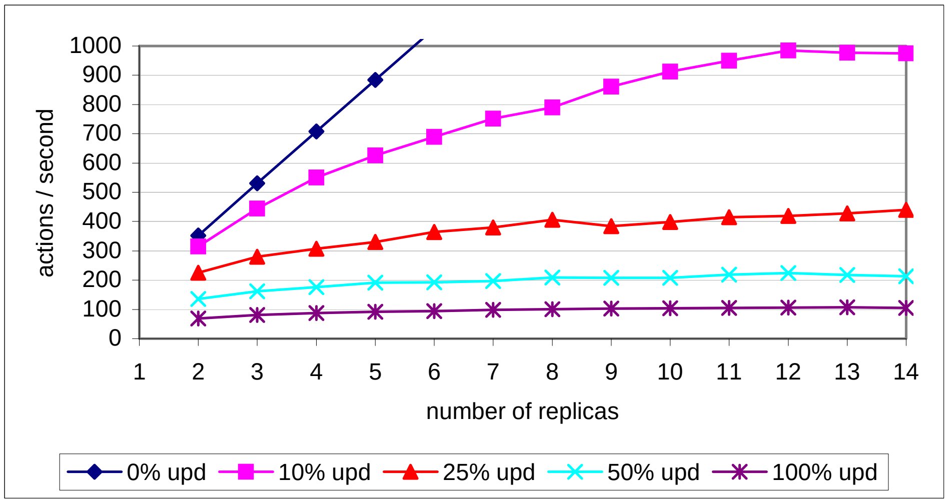

Figure 5: Max Throughput under a Varying Number of Replicas (LAN)

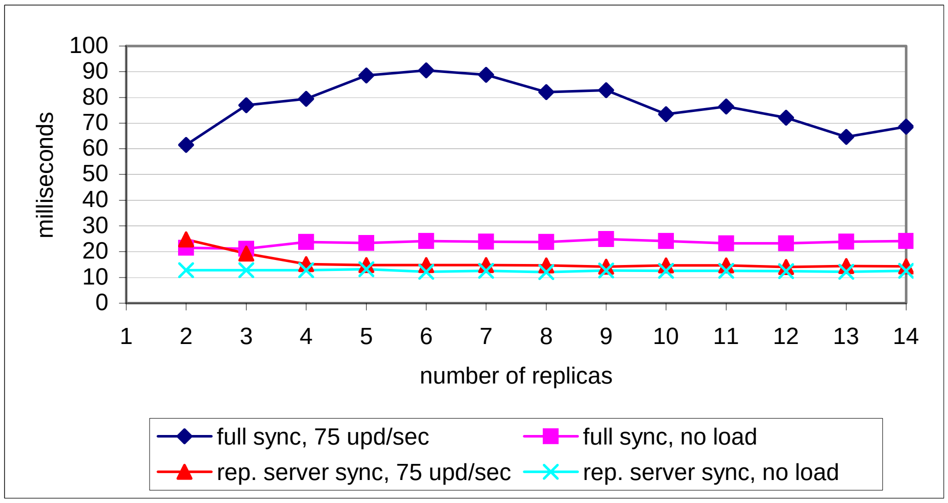

Figure 6: Latency of Updates (LAN)

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 updates per second, 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 specified in Figure 3. 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.

In addition to system throughput, we evaluated the impact of replication on the latency of each update transaction both under no load and medium load of 75 updates per second. Figure 6 shows that the latency of update transactions does not increase as the number of replicas increases. When the system becomes more loaded, the latency per transaction caused by Postgres increases from around 23ms to an average of 77ms. It is interesting to note that the latency does not increase under higher load when Postgres sync is disabled, but the Replication Server does sync.

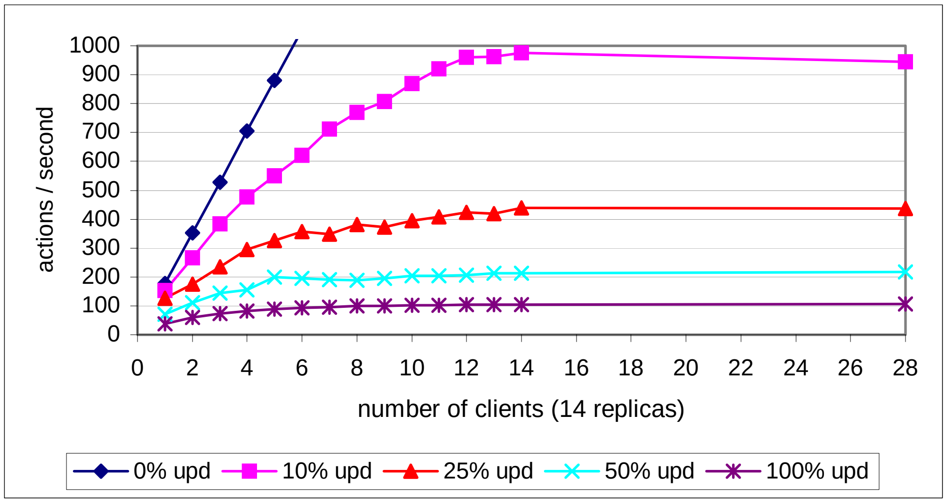

Figure 7: Throughput under Varied Client Set and Action Mix (LAN)

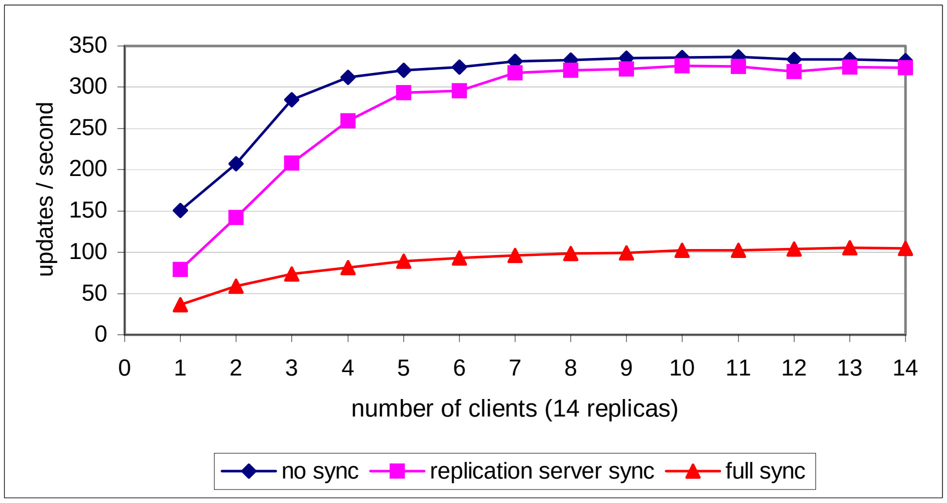

Figure 8: Throughput under Varied Client Set and Disk Sync Options (LAN)

The next two experiments fixed the number of replicas at 14, one replica on each Fog machine. The number of clients connected to the system was 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 distributed when there are more clients. 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.

To test the capacity of our system for updates, we reran the same experiment as Figure 7, but only with 100% updates, under the three sync variations mentioned at the beginning of this section. This is depicted in Figure 8. Under full-sync the maximum throughput of 107 updates per second is reached, while with sync disabled for both Postgres and the Replication Server, we reach a maximum throughput of 336 updates per second. The interesting point is that when the Replication Server does updates with full-sync, the system still achieves 326 updates per second. This shows that as the number of replicas and clients increase, the Replication Server overhead decreases considerably. The cost of disk syncs when only a few clients are used is still noticeable.

We next evaluated our system on the CAIRN wide-area network depicted in Figure 4. The diameter of the network as measured by *ping*, is approximately 45ms involving seven sites and a tree of six links. These experiments validated that the system works as expected on a real operational network. The system was able to achieve the potential performance the hardware allowed, as presented in Figure 3.

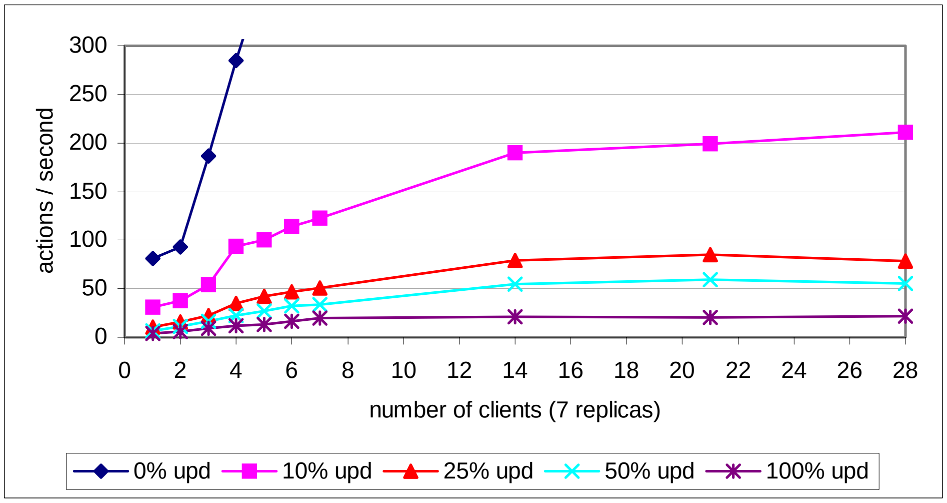

Figure 9: Throughput under Varied Client Set and Action Mix (CAIRN)

We ran the same basic experiment as done for Figure 7. In this experiment, however, the number of replicas is seven, because only seven sites were available. Figure 9 shows the throughput of the system as the number of clients connected to the system increases. Clients are evenly spread among the replicas. Because of the high network latency, more clients are required to achieve the potential maximum throughput when updates are involved. Also, the lines are not as smooth because the machines are very heterogeneous in their capabilities (ranging from an old Pentium 133 up to a Pentium II 450).

The latency each update experiences in the CAIRN network is 267ms under no load (measured at TISWPC in San Jose) and reaches 359ms at the point the system reaches maximum throughput of 20 updates per second. Once the throughput limit of the system is reached, the latency experienced by each client increases because more clients are sharing the same throughput.

As explained in the beginning of this section, the CAIRN machines were not adequate for our database performance tests because of their hardware limitations. We extended our wide-area tests by utilizing the Emulab facility. As accurately as possible, we emulated the CAIRN network on Emulab.

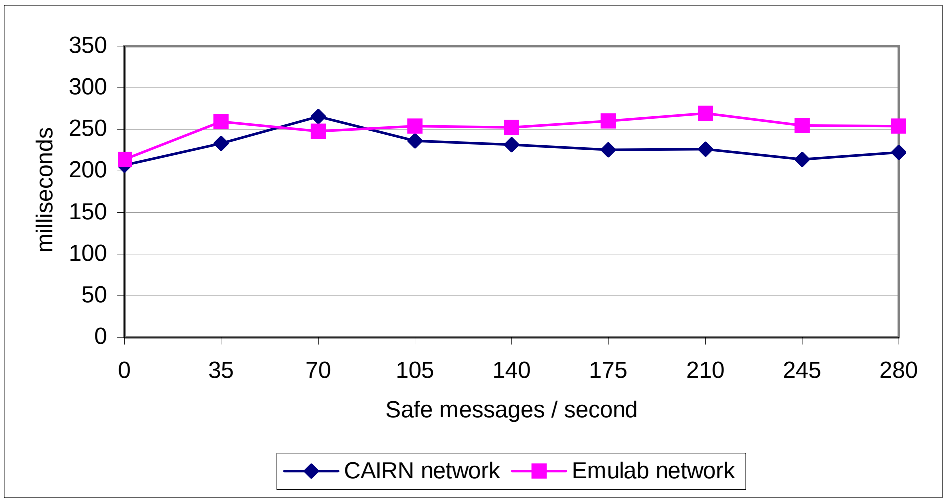

Figure 10: Latency of Safe Delivery on CAIRN and Emulab

Figure 10 shows the validation of Emulab against CAIRN by presenting the latency of Safe Delivery in the system under different Safe message load that resembles updates and queries in size. Under all loads both networks provided very similar message latency. Therefore, we believe that Emulab results are comparable to equivalent real-world networks running on the same machines.

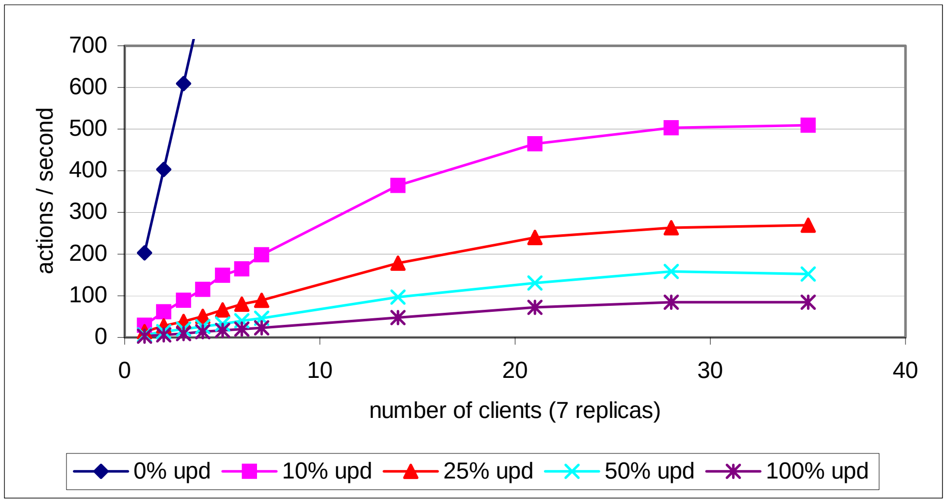

Figure 11: Throughput under Varying Client Set and Action Mix (Emulab)

The first experiment conducted on Emulab duplicated the experiment of Figure 9. In this case, the system was able to achieve a throughput close to that achieved on a local area network with similar number of replicas (seven), but with more clients as depicted in Figure 11. For updates, the maximum throughput achieved on Emulab is 85 updates per second (with about 28 clients in the system), compared with 98 updates per second on

LAN for the same number of replicas (with about 10 clients in the system) as can be seen in Figure 5. Although these results show very useful performance on a wide area network, we are not entirely happy with them since the algorithm predicts that the Emulab system should be able to reach similar maximum throughput for updates (as long as enough clients inject updates). We attribute the 14% difference to the fact that the Fog machines used in the LAN test are dual processor machines with SCSI disks, while the Emulab machines are single (though somewhat stronger) processor machines with IDE disks.

The latency each update experiences in this Emulab experiment is 268ms when no other load is present (almost identical to the corresponding CAIRN experiment above) and reaches 331ms at the point the system reaches maximum throughput of 85 updates per second. Again, once the throughput limit of the system is reached, the latency experienced by each client increases because more clients are sharing the same throughput.

For queries, similarly to the LAN tests in Figure 7, the performance increases linearly with the number of clients until the seven servers are saturated with seven clients at 1422 queries per second. The results for in-between mixes are as expected.

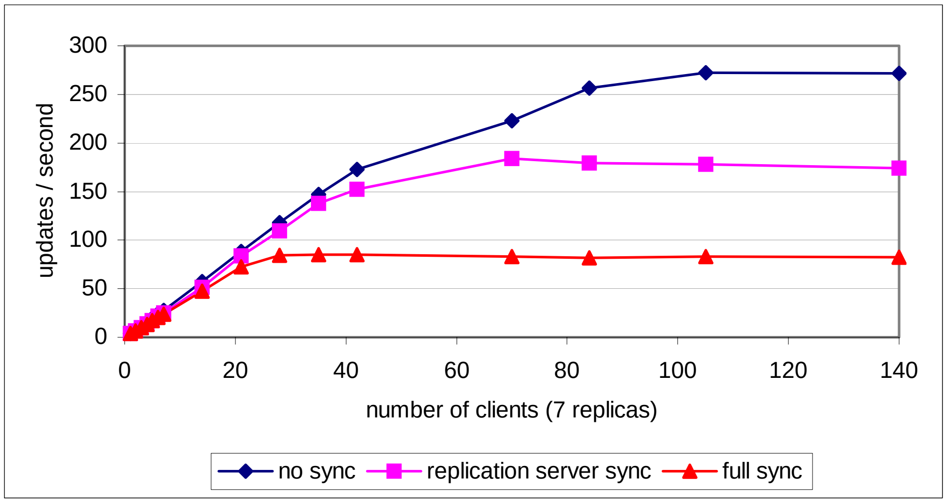

Figure 12: Throughput under Varied Client Set and Disk Sync Options (Emulab)

To test the capacity of our system for updates, we reran the same experiment as Figure 11, but just with 100% updates, under the three sync variations mentioned at the beginning of this section. Figure 12 shows that under full-sync the maximum throughput of 85 updates per second is reached, while with sync disabled for both Postgres and the Replication Server a maximum throughput of 272 updates per second is reached (with 105 clients in the system). When only the Replication Server syncs to disk, the system achieves 183 updates per second (with 70 clients in the system). The system performance does not degrade as clients are added.

To test the impact of network latency, we used Emulab to construct two additional networks, identical in topology to the CAIRN 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, as the original CAIRN network has a diameter of about 45ms, and the additional networks have about 22ms and 90ms diameters respectively. Figure 13 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.

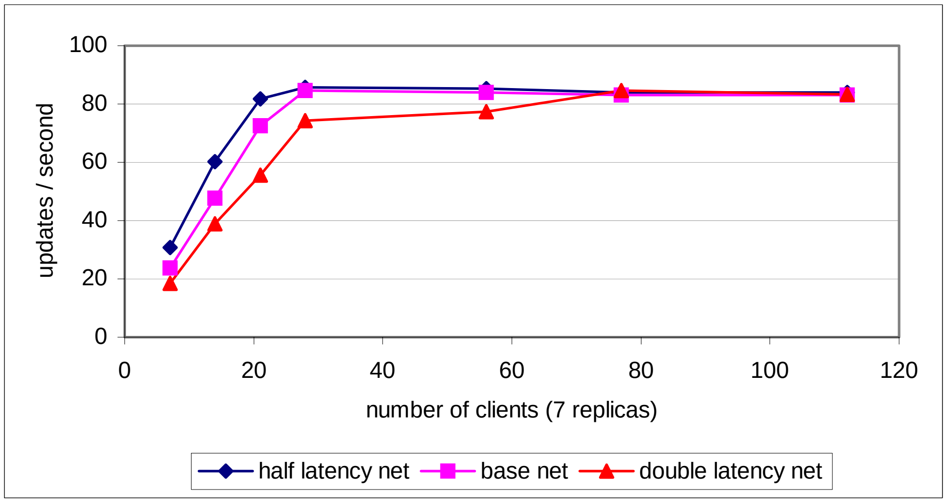

Figure 13: Throughput under Varied Network Latency (Emulab)

We conclude that the above results demonstrate that for many applications, peer synchronous database replication is becoming an attractive option not only for local area networks, but also for wide area networks.

6. Conclusions

One of the difficulties in providing efficient and correct wide area database replication is that it requires integrating different techniques from several research fields including distributed systems, databases, network protocols and operating systems. Not only does each of these techniques have to be efficient by itself, they all have to be efficient in concert with each other.

Some highlights of our results, replicating the Postgres database (that can perform about 120 updates per second without replication): For a local-area cluster with 14 replicas, the latency each update experiences is 27ms under zero throughput and 50ms under a throughput of 80 updates per second. The highest throughput in this setting is 106 updates per second.

For a wide-area network with a 45ms diameter and 7 replicas, the latency each update experiences is 268ms under zero throughput and 281ms under a throughput of 73 updates per second. The highest throughput in this setting is 85 updates per second, which is achieved with 28 clients and a latency of 331ms per update. When the diameter of the network is doubled to 90ms, the 85 updates per second throughput is maintained (although more clients are needed).

In all cases, the throughput for read-only actions approaches the combined power of all the replicas.

These results demonstrate the practicality of local and wide area peer (synchronous) database replication. Using our scheme to replicate a database with better performance could achieve better results since our replication architecture was not the bottleneck.

To achieve these results optimizations had to take place at various levels: First, at the network level, we had to optimize the latency of Safe Delivery on wide area networks. Second, we had to avoid end-to-end acknowledgments for each transaction while not compromising correctness of the system even in partitionable and crash-prone environments, and not delaying transaction execution. Third, we had to minimize the synchronous disk writes the replication server requires in addition to those performed by the database manager. Fourth, we had to obtain some semantic knowledge to correctly avoid replicating transactions that do not require it (e.g. read-only queries).

We show the feasibility of a database replication architecture that is transparent to both client and database manager, correctly handles arbitrary failures, and supports a number of different semantic guarantees for transactions such as one-copy serializability, weak consistency and delayed updates.

References

[A95] Y. Amir. Replication Using Group Communication Over a Partitioned Network. Ph.D. thesis, The Hebrew University of Jerusalem, Israel 1995. www.cs.jhu.edu/~yairamir.

[AADS01] Yair Amir, Baruch Awerbuch, Claudiu Danilov, and Jonathan Stanton. Flow control for many-to-many multicast: A cost-benefit approach. Technical Report CNDS--2001--1, Johns Hopkins University, Center for Networking and Distributed Systems, 2001.

[ADS00] Yair Amir, Claudiu Danilov, and Jonathan Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 327--336. IEEE Computer Society Press, Los Alamitos, CA, June 2000. FTCS 30.

[AMMB98] 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, 16(2):*93--132, May 1998.

[APS98] Yair Amir, Alec Peterson, and David Shaw. Seamlessly Selecting the Best Copy from Internet-Wide Replicated Web Servers. *Proceedings of the International Symposium on Distributed Computing (Disc98), LNCS 1499,* pages 22-33 Andros, Greece, September 1998.

[AS98] Yair Amir and Jonathan Stanton. The Spread wide area group communication system. Technical Report 98-4, Johns Hopkins University, CNDS, 1998.

[AT01] Yair Amir and Ciprian Tutu. From total order to database replication. *Proceedings of the International Conference on Distributed Computing Systems,* July 2002, to appear. Also, Technical Report CNDS-2001-6, Johns Hopkins University, November 2001, www.cnds.jhu.edu./publications.

[AW96] Y.Amir and A.Wool. Evaluating quorum systems over the internet. *Symposium on Fault-Tolerant Computing,* pages 26--35, 1996.

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

[Demers87] A. Demers et al. Epidemic algorithms for replicated database maintenance. Fred B. Schneider, editor, *Proceedings of the 6th Annual {ACM} Symposium on Principles of Distributed Computing*, pages 1--12, Vancouver, BC, Canada, August 1987. ACM Press.

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

[EMULAB] http://www.emulab.net

[GR93] J. N. Gray and A. Reuter. Transaction Processing: concepts and techniques. Data Management Systems. Morgan Kaufmann Publishers, Inc., 1993.

[H98] Mark Hayden. The Ensemble System. PhD thesis, Cornell University, 1998.

[HAE00] J. Holliday, D. Agrawal, and A. El Abbadi. Database replication using epidemic update.Technical Report TRCS00-01, University of California Santa-Barbara, 19, 2000.

[Informix] www.informix.com

[JM90] S. Jajodia and D. Mutchler. Dynamic Voting Algorithms for Maintaining the Consistency of Replicated Database. *ACM Trans. on Database Systems,* 15(2):230-280, June 1990.

[JPKA00] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters, Technical Report 2000

[KA00] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication. *Proceedings of the 26th International Conference on Very Large Databases (VLDB),* Cairo, Egypt, September 2000.

[KBB01] B. Kemme, A. Bartoli and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. *In Proceedings of the International Conference on Dependable Systems and Networks (IC-DSN),* pages 117-126, Sweden, July 2001.

[KK00] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. *In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems,* p. 344--355, Taipei, Taiwan, April 2000.

[KSMD00] Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. A client-server oriented algorithm for virtually synchronous group membership in {WAN}s. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems,* pages 356--365, Taipei, Taiwan, April 2000

[L78] L. Lamport. Time, Clocks, and The Ordering of Events in a Distributed System. *Comm. ACM*, 21(7), pages 558-565. 1978.

[MAMA94] L. E. Moser, Y. Amir, P. M. Melliar-Smith and D. A. Agarwal. Extended Virtual Synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems,* pages 56-65, June 1994.

[Mon00] Alberto Montresor. System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems.PhD thesis, Dept. of Computer Science, University of Bologna, Februrary 2000.

[MySQL] http://www.mysql.com/doc/R/e/Replication.html

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

[PJKA00] M. Patino-Martinez and R. Jimenez-Peris and B. Kemme and G. Alonso. Scalable replication in database clusters. *Proceedings of 14th International Symposium on DIStributed Computing (DISC2000),* 2000

[PL88] J. F. Paris and D. D. E. Long. Efficient Dynamic Voting Algorithms. In *Proceedings of the 4th International Conference on Data Engineering*, pages 268-275, February 1988.

[Pgrep] pgreplicator.sourceforge.net

[Postgres] www.postgresql.com.

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

[RBM96] Robbert~Van Renesse, Kenneth Birman, and S.~Maffeis. Horus: A flexible group communication system. *Communications of the ACM, 39(4):*76--83, April 1996

[Spread] www.spread.org.

[Sybase] www.sybase.com.

Appendix 1. Example of Safe Delivery in Action

Let's consider a network of five daemons N_1 – N_5 where each of the daemons represents a site. Figure A1 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 (Figure A1.a).

Upon receiving the Safe message from N_1, the daemon N_4 updates its *Site_lts* (Figure A1.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.

seq	lts	aru
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Global_Aru: 0

(a)

seq	lts	aru
0	0	0
0	0	0
0	0	0
0	1	0
0	0	0

Global_Aru: 0

(b)

seq	lts	aru
0	1	0
0	1	0
0	1	0
0	1	1
0	1	0

Global_Aru: 0

(c)

seq	lts	aru
0	1	1
0	1	1
0	1	1
0	1	1
0	1	1

Global_Aru: 1

(d)

Figure A1: Scalable Safe Delivery for Wide Area Networks

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 (Figure A1.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 (Figure 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.