ETH zürich

ETH Library

Ganymed

Scalable Replication for Transactional Web Applications

Report

Author(s):
Plattner, Christian; Alonso, Gustavo

Publication date:
2004

Permanent link:
https://doi.org/10.3929/ethz-a-006741951

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
Technical Report / ETH Zurich, Department of Computer Science 439

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Ganymed: Scalable Replication for Transactional Web Applications

Christian Plattner and Gustavo Alonso

Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, CH-8092 Zürich, Switzerland
{plattner,alonso}@inf.ethz.ch

Abstract. Data grids, large scale web applications generating dynamic content and database service providing pose significant scalability challenges to existing database architectures. Replication is the most common solution, but, in all cases, existing systems still have drawbacks and impose difficult trade-offs. The most difficult one of all is the choice between scalability and consistency. Commercial systems give up consistency, while research solutions typically either offer a compromise (limited scalability in exchange for consistency) or they impose severe limitations on the data schema and the accepted transactional workload. In this paper we introduce Ganymed, a database replication middleware intended to provide scalability without sacrificing consistency and avoiding the limitations of existing approaches. The main idea is to use a novel transaction scheduling algorithm that separates update and read-only transactions. Transactions can be submitted to Ganymed through a special JDBC driver. Ganymed then routes updates to a main server and queries to a potentially unlimited number of read-only copies. The system guarantees that all transactions see a consistent data state (snapshot isolation). In the paper we describe the scheduling algorithm, the architecture of Ganymed, and present an extensive performance evaluation that proves the potential of the system.

1 Introduction

Traditionally used in predictable and static environments like ERP (Enterprise Resource Planning), data-base management systems had to face new challenges in the last few years. Web services, application service providing and grid computing require higher scalability and availability. Database replication is widely used to achieve those goals. The problem with replication is that existing commercial solutions do not behave like single instance databases. Often, a choice must be made between scalability and consistency. If full consistency is required, the price is a loss in scalability [1].

Recent research in the area of Transactional Web Applications has led to many alternative proposals, but they all suffer from a variety of problems. Systems that put data on the edge of the network, [22, 23, 24], are able to reduce

response times but give up consistency. Other approaches, [2, 3, 4], work on the middleware layer and need a predeclaration of the access pattern of all transactions to enable efficient scheduling. [5] offers both scalability and consistency, however the database designer is forced to partition the data statically. This forbids certain queries and restricts the free evolution of the application. In these systems, a reasonable scale-out is possible only if the load conforms exactly to the chosen data partition.

This paper describes Ganymed, a middleware based system designed to address the problems above. The system is able to offer both scalability and consistency without having to partition the data or knowing the transaction access pattern. Ganymed does not impose any restrictions on the queries submitted and does not duplicate work of the underlying database engines. The main component of Ganymed is a lightweight scheduler that routes transactions to a set of snapshot isolation [6, 7] based replicas by using RSI-PC, a novel scheduling algorithm. The key idea behind RSI-PC is the separation of update and read-only transactions. Updates will always be routed to a main replica, whereas the remaining transactions are handled by any of the remaining replicas, which act as read-only copies. The RSI-PC approach is tailored to dynamic content generation (as modelled in the TPC-W benchmark) and database service providing, where a vast amount of complex reads is conducted together with a small number of short update transactions. RSI-PC makes sure that all transactions see a consistent data state, inconsistencies between the replicas are hidden from the clients. A JDBC driver enables client applications to be connected to the Ganymed scheduler, thereby offering an accepted interface to the system.

The evaluation presented in the paper shows that Ganymed offers almost linear scalability. The scalability manifests itself both as increased throughput as well as reduced response times. In addition, Ganymed significantly increases the availability of the system. In the experiments we show the graceful degradation properties of Ganymed as well as the ability to migrate the main replica as needed.

The paper is structured as follows: first, we describe the RSI-PC scheduling algorithm and the middleware architecture of Ganymed. The system overview is followed by an evaluation of our working Java based prototype. The paper concludes with a discussion of related and future work.

2 Motivation

Database replication is the process of maintaining multiple copies of data items on different locations called replicas. As already noted, the motivation for doing this is twofold: on the one side, the availability can be increased since the database system can better tolerate failures; on the other hand, the systems's throughput can be increased and response times can be lowered by distributing the transaction load across the replicas.

Traditionally, there are two types of replication system: *eager* and *lazy* [1]. Eager (or *synchronous*) systems keep the replicas synchronized within transac-

tion boundaries. They conform to *1-copy-serializability* [8]: the resulting schedules are equivalent to a serial schedule on a single database. Although eager systems offer the same correctness guarantees as single-database installations, the concept is rarely used in commercial replication products. This stems from the fact that conventional eager protocols have significant drawbacks regarding performance and scalability [1, 8, 9, 10]: first, the communication overhead between the replicas is very high, leading to long response times, and second, the probability of deadlocks is proportional to the third power of number of replicas. Such traditional eager implementations are not able to scale beyond a few replicas. To circumvent these problems, database designers started creating lazy systems, in which replicas can be updated outside the transaction boundaries. As already noted, lazy replication scales very well, but leads to new problems: transactions can read stale data and conflicts between updating transactions are possibly detected very late, introducing the need for conflict resolution. Unfortunately, these problems cannot easily be hidden from the user, and are often even left to be resolved to the client applications.

The unsatisfactory properties of lazy protocols have led to a continuation in the research on eager protocols [2, 3, 11, 12]. An important result [11] is the insight that the distribution of full SQL update statements, as often done in eager update-everywhere approaches, is not optimal. Performance can be significantly improved by executing SQL statements only once and then propagating the resulting database changes (so called *writesets*) to other replicas.

The approach in Ganymed tries to combine the advantages of lazy and eager strategies. Ganymed uses a set of fully replicated databases. Internally, Ganymed works in a lazy way, not all replicas are updated inside transaction boundaries. For efficiency, the replication process is writeset based. Nevertheless, by working at the middleware layer the system is able to offer an eager service. In addition, Ganymed transparently takes care of configuration changes, failing replicas, migration of nodes, etc.

3 The RSI-PC Scheduling Algorithm

In this chapter we present our novel RSI-PC scheduling algorithm. RSI-PC stands for *Replicated Snapshot Isolation with Primary Copy*. It can be used to schedule transactions over a set of *Snapshot Isolation* based database replicas. RSI-PC works by separating read-only and update transactions and sending them to different replicas: updates are sent to a master replica (the primary copy), reads are sent to any of the other replicas, which act as caches. To the clients, the scheduler looks like a snapshot isolation database. Temporary inconsistencies between the replicas are hidden from the client by the scheduler, all synchronization is done transparently. The RSI-PC approach fits exactly the needs of typical dynamic content generation, where a vast amount of complex reads is conducted by a small number of short update transactions.

3.1 Snapshot Isolation

Snapshot Isolation (SI) [6, 7] is a multiversion concurrency control mechanism used in databases. Popular database engines that use SI include Oracle [13] and PostgreSQL [14]. One of the most important properties of SI is the fact that readers are never blocked by writers, an idea which was already used in the multiversion mixed protocol described in [8]. This property is a big win in comparison to systems that use two *two phase locking* (2PL), where many non-conflicting updaters may be blocked by as simply as one reader. SI completely avoids the four extended ANSI SQL phenomenas P0-P3 described in [6], nevertheless it does not guarantee serializability. As shown in [15, 16] this is not a problem in real applications, since transaction programs can be arranged in ways so that any concurrent execution of the resulting transactions is equivalent to a serialized execution.

For the purposes of this paper, we will work with the following definition of SI (slightly more formalized than the description in [6]):

> **SI**: A transaction T_i that is executed under snapshot isolation gets assigned a start timestamp $start(T_i)$ which reflects the starting time. This timestamp is used to define a snapshot S_i for transaction T_i. The snapshot S_i consists of the latest committed values of all objects of the database at the time $start(T_i)$. Every read operation issued by transaction T_i on a database object x is mapped to a read of the version of x which is included in the snapshot S_i. Updated values by write operations of T_i (which make up the *writeset* WS_i of T_i) are also integrated into the snapshot S_i, so that they can be read again if the transaction accesses updated data. Updates issued by transactions that did not commit before $start(T_i)$ are invisible to the transaction T_i. When transaction T_i tries to commit, it gets assigned a commit timestamp $commit(T_i)$, which has to be larger than any other existing start timestamp or commit timestamp. Transaction T_i can only successfully commit if there exists no other committed transaction T_k having a commit timestamp $commit(T_k)$ in the interval $\{start(T_i), commit(T_i)\}$ and $WS_k \bigcap WS_i \neq \{\}$. If such a committed transaction T_k exists, then T_i has to be aborted (this is called the $first$-$commiter$-$wins$ rule, which is used to prevent lost updates). If no such transaction exists, then T_i can commit (WS_i gets applied to the database) and its updates are visible to transactions which have a start timestamp which is larger than $commit(T_i)$.

A sample execution of transactions running on a database offering SI is given in 1. The symbols B, C and A refer to the *begin*, *commit* and *abort* of a transaction. The long running transaction T_1 is of type $read - only$, i.e., its writeset is empty: $WS_1 = \{\}$. T_1 will never be blocked by any other other transaction, nor will it block other transactions. Updates from concurrent updaters (like T_2, T_3, T_4 and T_6) are invisible to T_1. T_2 will update the database element X, it does not conflict with any other transaction. T_3 updates Y, it does not see the changes made by T_2, since it started while T_2 was still running. T_4 updates X

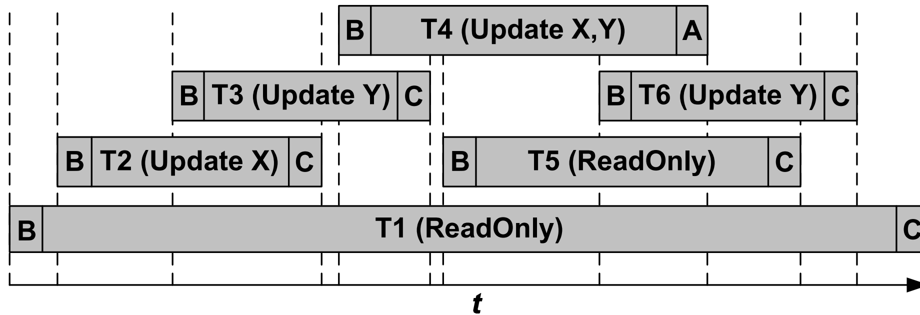

Fig. 1. An example of transactions running under SI.

and Y. Conforming to the first-committer-wins rule it cannot commit, since its writeset overlaps with that from T_3 and T_3 committed while T_4 was running. The transaction manager has therefore to abort T_4. T_5 is read-only and sees the changes made by T_2 and T_3. T_6 can successfully update Y. Due to the fact that T_4 did not commit, the overlapping writesets of T_6 and T_4 do not impose a conflict.

As will be shown in the next section, practical systems handle the comparison of writesets and the first-committer-wins rule in a different, more efficient way. Both, Oracle and PostgreSQL, offer two different ANSI SQL isolation levels for transactions: SERIALIZABLE and READ COMMITTED. An extended discussion regarding ANSI SQL isolation levels is given in [6].

3.2 The SERIALIZABLE Isolation Level

Oracle and PostgreSQL implement a variant of the SI algorithm for transactions that run in the isolation level SERIALIZABLE. Writesets are not compared at the commit time of transactions, instead this process is done progressively by using row level write locks. When a transaction T_i running in isolation level SERIALIZABLE tries to modify a row in a table that was modified by a concurrent transaction T_k which has already committed, then the current update operation of T_i gets aborted immediately. Unlike PostgreSQL, which then aborts the whole transaction T_i, Oracle is a little bit more flexible: it allows the user to decide if he wants to commit the work done so far or if he wants to proceed with other operations in T_i. If T_k is concurrent but not committed yet, then both products behave the same: they block transaction T_i until T_k commits or aborts. If T_k commits, then the same procedure gets involved as described before, if however T_k aborts, then the update operation of T_i can proceed. The blocking of a transaction due to a potential update conflict is of course not unproblematic since it can lead to deadlocks, which must be resolved by the database by aborting transactions.

3.3 The READ COMMITTED Isolation Level

Both databases offer also a slightly less strict isolation level called READ COMMITTED, which is based on a variant of snapshot isolation. READ COMMITTED is the default isolation level for both products. The main difference to SERIALIZABLE is the implementation of the snapshot: a transaction running in this isolation mode gets a new snapshot for every issued SQL statement. The handling of conflicting operations is also different than in the SERIALIZABLE isolation level. If a transaction T_i running in READ COMMITTED mode tries to update a row which was already updated by a concurrent transaction T_k, then T_i gets blocked until T_k has either committed or aborted. If T_k commits, then T_i's update statement gets reevaluated again, since the updated row possibly does not match a used selection predicate anymore. READ COMMITTED avoids phenomena P0 and P1, but is vulnerable to P2 and P3 (fuzzy read and phantom).

3.4 RSI-PC Definition

Schedulers implementing the RSI-PC algorithm can be used with SI based replicas that offer the transaction isolation levels SERIALIZABLE and READ COMMITTED as defined above. For incoming update transactions, RSI-PC is able to support the SERIALIZABLE and READ COMMITTED transaction isolation levels. Read-only transactions always have to be run in isolation mode SERIALIZABLE, therefore they the see the same snapshot during the whole transaction.

> **RSI-PC:** A RSI-PC scheduler is responsible for a set of n SI-based replicas. One of the replicas is used as *master*, the other $n-1$ replicas are the *slaves*. The scheduler makes a clear distinction between *read-only* and *update* transactions, which have to be marked by the client application in advance.
> **Update transactions:** b_i, $r_j(x)$, $w_j(x)$ and c_i statements of any arriving update transaction are directly forwarded to the master (including their mode, SERIALIZABLE or READ COMMITTED), they are never delayed. The scheduler takes notice of the order in which update transactions commit on the master. After a successful commit of an update transaction, the scheduler makes sure that the corresponding write set is sent to the $n-1$ slaves and that every slave applies the write sets of different transactions in the same order as the corresponding commit occurred on the master replica. The scheduler uses also the notation of a global database version number. Whenever an update transaction commits on the master, the global database version number is increased by one and the client gets notified about the commit. Write sets get tagged by the version number which was created by the corresponding update transaction.
> **Read-Only transactions:** read-only transactions can be processed by any replica in SERIALIZABLE mode. The scheduler is free to decide on

which replica to execute such a transaction. If on a chosen replica the latest produced global database version is not yet available, the scheduler must delay the creation of the read-only snapshot until all needed write sets have been applied to the replica. For clients that are not willing to accept any delays for read-only transactions there are two choices: either their queries are sent to the master replica, therefore reducing the available capacity for updates, or the client can set a staleness threshold. A staleness threshold is for example a maximum age of the requested snapshot in seconds or the condition that the client sees its own updates. The scheduler can then use this threshold to choose a replica. Once the scheduler has chosen a replica and the replica created a snapshot, all consecutive operations of the read-only transaction will be performed using that snapshot. Due to the nature of SI, the application of further write sets on this replica will not conflict with this or any other running read-only transaction.

Due to its simplicity, there is no risk of a RSI-PC scheduler becoming the bottleneck in the system. In contrast to other middleware based schedulers, like the ones used in [2,5], this scheduling algorithm does not involve any SQL statement parsing or concurrency control operations. Also, no row or table level locking is done at the scheduler level. The detection of conflicts, which by definition of SI can only happen during updates, is left to the SI database running on the master replica. Moreover, (unlike [4,5]), RSI-PC does not make any assumptions about the transactional load, the data partition, organization of the schema, or answerable and unanswerable queries.

Since only a small amount of state information must be kept by a RSI-PC scheduler, it is even possible to construct parallel working schedulers. This helps to improve the overall fault tolerance. In contrast to traditional eager systems, where every replica has its own scheduler that is aware of the global state, the exchange of status information between a small number of RSI-PC schedulers can be done very efficiently. Even in the case that all schedulers fail, it is possible to reconstruct the overall database state: a replacement scheduler can be used and its state initialized by inspecting all available replicas.

In the case of failing slave replicas, the scheduler simply ignores them until they have been repaired by an administrator. However, in the case of a failing master, things are a little bit more complicated. By just electing a new master the problem is only halfway solved. The scheduler must also make sure that no updates from committed transactions get lost, thereby guaranteeing ACID durability. This goal can be achieved by only sending commit notifications to clients after the writesets of update transactions have successfully been applied on a certain, user defined amount of replicas.

4 Ganymed Middleware Architecture

The main component of Ganymed is a lightweight middleware scheduler which balances transactions from clients over a set of SI based database replicas. Clients

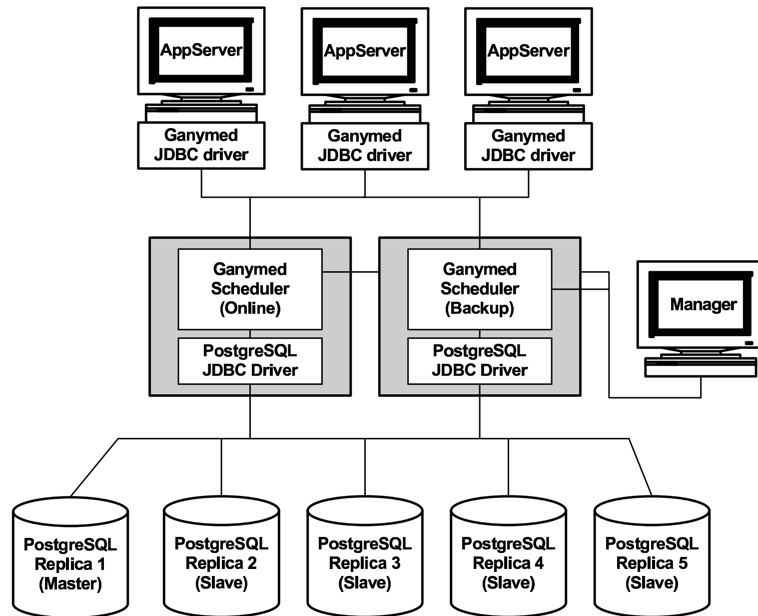

Fig. 2. Ganymed Prototype Architecture

are typically application servers in e-commerce environments. From the viewpoint of such clients, the Ganymed scheduler behaves like a single SI based database. We have implemented a working prototype of the system using Java.

Figure 2 shows the main components of the architecture. Applications servers connect to the Ganymed scheduler through a custom JDBC driver. The Ganymed scheduler, implementing the RSI-PC algorithm, then distributes incoming transactions over the master and slave replicas. Replicas can be added and removed from the system at runtime. The master role can be assigned dynamically, for example when the master replica fails. The current prototype does not support parallel working schedulers, yet it is not vulnerable to failures of the scheduler. If a Ganymed scheduler fails, it will immediately be replaced by a standby scheduler. The decision for a scheduler to be replaced by a backup has to be made by the manager component. The manager component, running on a dedicated machine, constantly monitors the system. The manager component is also responsible for reconfigurations. It is used, e.g., by the database administrator to add and remove replicas. Interaction with the manager component takes place through a graphical interface.

In the following sections we will describe each component of the system in more detail.

4.1 Client interface

Clients connect to the scheduler through the Ganymed JDBC 3.0 compliant database driver. The availability of such a standard database interface makes it straightforward to connect Java based application servers (like BEA Weblogic[1] or JBoss[2]) to Ganymed. The migration from a centralized database to a Ganymed environment is very simple, only the JDBC driver component in the application server has to be reconfigured, there is no change in the application code. Our driver also supports a certain level of fault tolerance. If a configured Ganymed scheduler is not reachable, the driver automatically tries to connect to an alternate scheduler.

Since the scheduler needs to know if a transaction is an update or read-only, the application code has to communicate this to the Ganymed JDBC driver. This mechanism is already included in the JDBC standard. Application code that wants to start a read-only transaction simply calls the *Connection.setReadonly()* method.

4.2 Replica Support and Writeset Handling

On the replica side, Ganymed currently supports PostgreSQL and Oracle database engines. This allows to build heterogenous setups, where replicas run different engines. Also, our approach is not limited to a specific operating system. As in the client side, the communication with the replicas is done through JDBC drivers. In a heterogenous configuration, the Ganymed scheduler has therefore to load for every different type of replica the corresponding JDBC driver. Since this can be done dynamically at run time, on startup the scheduler does not need to be aware of the type of replicas added at runtime.

Unfortunately, the JDBC interface has no support for writeset extraction, which is needed on the master replica. In the case of PostgreSQL, we implemented an extension of the database software. PostgreSQL is very flexible, it supports the loading of additional functionality during runtime. Our extension consists of a shared library which holds the necessary logic to collect changes of update transactions in the database. Internally, the tracking of updates is done using triggers. To avoid another interface especially for the writeset handling, the extension was designed to be controlled over the normal JDBC interface. For instance, the extraction of a writeset can be performed with a "*SELECT writeset()*" SQL query. The extracted writesets are table row based, they do not contain full disk blocks. This ensures that they can be applied on a replica which uses another low level disk block layout than the master replica.

In the case of Oracle, such an extension can be built in a similar manner.

4.3 RSI-PC Implementation

The Ganymed scheduler implements the RSI-PC algorithm. Although feasible, loose consistency models are not supported in the current version. The scheduler

[1] http://www.bea.com
[2] http://www.jboss.org

always provides strong consistency. Read-only transactions will always see the latest snapshot of the database. The scheduler also makes a strict distinction between the master and the slave replicas. Even if there is free capacity on the master, read-only transactions are always assigned to a slave replica. This ensures that the master is not loaded by complex read-only transactions and that there is always enough capacity on the master for sudden bursts of updates. If there is no slave replica present, the scheduler is forced to assign all transactions to the master replica, acting as a relay.

As already noted, implementing RSI-PC does not involve any SQL parsing and table locking at the middleware layer. In some regards, Ganymed resembles more a router than a scheduler. Incoming transactions result in a decisions step: to which replica they must be sent. In the case of updates, this is always the master. Read-only transactions are assigned to a valid replica according to the LPRF (*least pending requests first*) rule. Valid means in this context, that the replica must contain the latest produced writeset. If no such replica exists, the start of the transaction is delayed.

For every replica, Ganymed keeps a pool of open connections. After an incoming transaction has been assigned to a replica, the scheduler uses a connection from the replica's pool to forward the SQL statements of that transaction. Every connection is used for a single transaction, connections are never shared. Also, once a transaction is assigned to a connection, this assignment will not change. If a pool is exhausted, Ganymed opens new connections until a per replica limit is reached, at that point arriving transaction are blocked. After a transaction commits or aborts, the assigned connection is returned to the pool of the respective replica. Unlike solutions using group communication [4, 11] Ganymed does not require that transactions are submitted as a block, i.e., the entire transaction must be present for it to be scheduled. In Ganymed different SQL statements of the same transaction are progressively scheduled as they arrive, with the scheduler ensuring that the result is always consistent.

To achieve a consistent state between all replicas, the scheduler must make sure that writesets of update transactions get applied on all replicas in the same order. This already imposes a problem when writesets are generated on the master, since the scheduler must be sure about the correct commit order of transactions. Ganymed solves this problem by sending COMMIT operations to the master in a serialized fashion. The distribution of writesets is handled by having a FIFO update queue for every replica. There is also for every replica a thread in the scheduler software that applies constantly the contents of that queue to its assigned replica. Clearly, the distribution of writesets for a large set of replicas in a star configuration is not optimal. A concrete optimization would be to have the replicas directly receiving writesets from the master, e.g. similar to the Oracle STREAMS [17] feature.

4.4 Manager Console

The manager console is responsible for monitoring the Ganymed system. On the one hand, it includes a permanently running process which monitors the load of

the system and the failure of components. On the other hand it is used by the administrator to perform configuration changes.

While the failure of replicas can directly be handled by the scheduler (failed replicas are just discarded, failed masters are replaced by a slave replica), the failure of a scheduler is more critical. In the event that a scheduler fails, the monitor process in the manager console will detect this and is responsible for starting a backup scheduler. Client side Ganymed JDBC drivers will also detect the failure and try to find a working scheduler according to their configuration. Assuming fail stop behavior, the connections between the failing scheduler and all replicas will be closed, all running transactions will be aborted by the assigned replicas. The manager will then inspect all replicas, elect a master and configure the new scheduler so that transaction processing can continue. The inspection of a replica involves the detection of the last applied writeset, which can be done by the same software implementing the writeset extraction.

The manager console is also used by administrators that need to change the set of attached replicas to a scheduler, or need to reactivate disable replicas. While the removal of a replica is a relatively simple task, the attachment or re-enabling of a replica is a more challenging task. Syncing-in a replica is actually performed by copying the state of a running replica to the new one. At the scheduler level, the writeset queue of the source replica is also duplicated and assigned to the destination replica. Since the copying process uses a SERIALIZABLE read-only transaction on the source replica, there is no need for shutting it down during the duplicating process. The new replica cannot be used to serve transactions until the whole copying process is over. Its writeset queue, which grows during the copy process, will be applied as soon as the copying has finished. Although from the viewpoint of performance this is not optimal, in the current prototype the whole copying process is done by the scheduler under the control of the manager console.

5 Experimental Evaluation

To verify the validity of our approach we performed several tests. First, we did extensive scalability measurements by comparing the performance of different Ganymed configurations with a single PostgreSQL instance. We used a load generator that simulates the transaction traffic of a TPC-W application server. Second, we tested the behavior of Ganymed in scenarios where replicas are failing. The failure of both, slaves and masters, was investigated.

5.1 TPC-W traces

The TPC benchmark W (TPC-W) is a transactional web benchmark from the Transaction Processing Council [18]. TPC-W defines an internet commerce environment that resembles real world, business oriented, transactional web applications. The benchmark also defines different types of workloads which are intended to stress different components in such applications (namely multiple

on-line browser sessions, dynamic page generation with database access, update of consistent web objects, simultaneous execution of multiple transaction types, a backend database with many tables with a variety of sizes and relationships, transaction integrity (ACID) and contention on data access and update). The workloads are as follows: primarily shopping (WIPS), browsing (WIPSb) and web-based ordering (WIPSo). The difference between the different workloads is the ratio of browse to buy: WIPSb consists of 95% read-only interactions, for WIPS the ratio is 80% and for WIPSo the ratio is 50%. WIPS, being the primary workload, it is considered the most representative one.

Numitem	10000
Customers	288000
# of EB	100

Table 1. TPC-W Parameters used for Trace Generation.

For the evaluation of Ganymed we generated database transaction traces with a running TPC-W installation. We used an open source implementation [19] that had to be changed to support a PostgreSQL backend database. Although the specification allows the use of loose consistency models, we did not make use of that. Our implementation is based on strong consistency. Parameters used to populate the installation and do the trace file generation are given in table 1.

Traces were then generated for the three different TPC-W workloads: *shopping mix* (a trace file based on the WIPS workload), *browsing mix* (based on WIPSb) and *ordering mix* (based on WIPSo). Each trace consists of 50'000 consecutive transactions.

5.2 The Load Generator

The load generator is Java based. Once started, it loads a trace file into memory and starts parallel database connections using the configured JDBC driver. After all connections are established, transactions are read from the in-memory tracefile and then fed into the database. Once a transaction has finished on a connection, the load generator assigns the next available transaction from the trace to the connection.

For the length of the given measurement interval, the number of processed transactions and the average response time of transactions are measured. Also, to enable the creation of histograms, every second the current number of processed transactions and their status (committed/aborted) is recorded.

5.3 Experimental Setup

For the experiments, a pool of machines had to be used to host the different parts of the Ganymed system. For every component (load generator, scheduler,

database replicas) of the system, a dedicated machine was used. All machines had the same configuration (Dual AMD Athlon 1400 MHz CPU, 1 GB RAM, 80 GB IBM Deskstar harddisk, Linux kernel 2.4.20, PostgreSQL 7.4.1, Blackdown-1.4.2-rc1 Java 2 Platform). All machines were connected through a 100 MBit Ethernet LAN.

Before starting any experiment, all databases were always reset to an initial state. Also, the PostgreSQL *VACUUM FULL ANALYZE* command was executed. This ensured that every experiment started from the same state.

In the experiments, we did not use a manager console or a backup Ganymed scheduler. The scheduler was always configured with automatic generated configuration scripts. Also, for update transactions, the Ganymed scheduler was always configured to report a transaction as committed to the client as soon as the commit was successful on the master replica. In theory, this could rise the small possibility of a lost update in case the scheduler fails. The detection of a failed scheduler is outside of the scope of this evaluation.

5.4 Part 1: Performance and Scalability

The first part of the evaluation is concentrated on performance and scalability. The Ganymed prototype was compared with a reference system consisting of a single PostgreSQL instance. We measured the performance of the Ganymed scheduler in different configurations, from 1 up to 7 replicas. This gives a total of 8 experimental setups (called PGSQL and GNY-n, $1 \leq n \leq 7$), each setup was tested with the three different TPC-W traces.

The load generator was then attached to the database (either the single instance database or the scheduler, depending on the experiment). During a measurement interval of 100 seconds, a trace was then fed into the system over 100 parallel client connections and at the same time average throughput and response times were measured. All transactions, read-only and updates, were executed in SERIALIZABLE mode. Every experiment was repeated until a sufficient, small standard deviation was reached (included in the graphs, however, the attained standard deviation was very low, it is only visible for the TPC-W ordering mix in the GNY-6 and GNY-7 setups.).

Figure 3 shows the results for the achieved throughput (transactions per second) and average transaction response times, respectively. The rate of aborted transactions was below 0.5 percent for all experiments. Figure 4 shows two example histograms for the TPC-W ordering mix workload: on the left side the reference system, on the right side GNY-7. The short drop in performance in the GNY-7 histogram is due to multiple PostgreSQL replicas that did checkpointing of the WAL (*write ahead log*) at the same time. The replicas were configured to perform this process at least every 300 seconds, this is the default for PostgreSQL.

Based on the graphs, we can prove the lightweight structure of the Ganymed prototype. In a relay configuration, where only one replica is attached to the Ganymed scheduler, the achieved performance is close to identical to the PostgreSQL reference system. The performance of the setup with two replicas, where

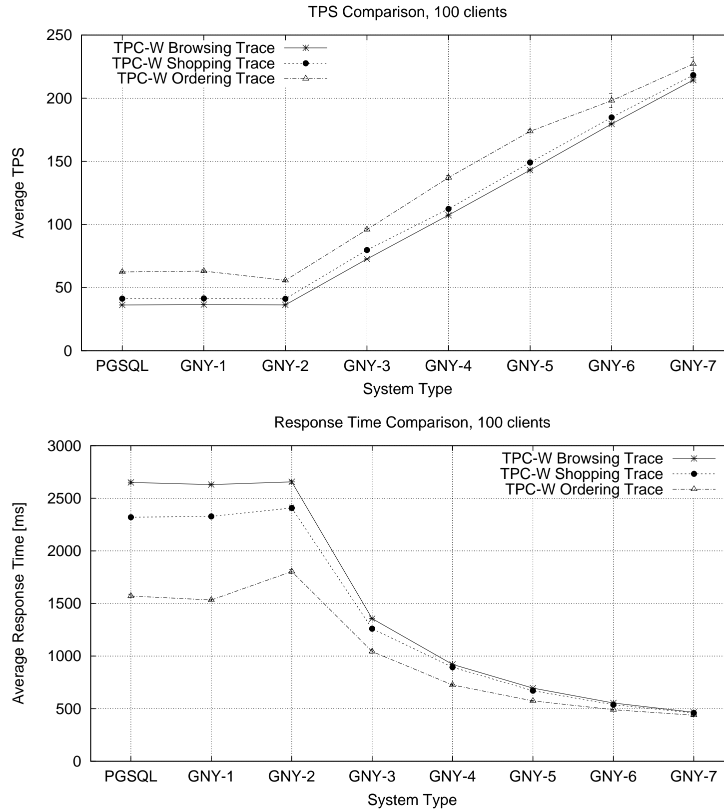

Fig. 3. Ganymed Performance for TPC-W Mixes.

one replica is used for updates and the other for read-only transactions, is comparable to the single replica setup. This clearly reflects the fact that the heavy part of the TPC-W loads consists of complex read-only queries. In the case of the write intensive TPC-W ordering mix, a two replica setup is slightly slower than the single replica setup. In the setups where more than two replicas are used, the performance compared to the reference system could be significantly improved. A close look at the response times chart shows that they converge. This is due to the RSI-PC algorithm which uses parallelism for different transactions, but no intra-parallelism for single transactions. A GNY-7 system, for example, would have the same performance as a GNY-1 system when used only by a single client.

One can summarize that in almost all cases a nearly linear scale-out was achieved. These experiments show that the Ganymed scheduler was able to attain an impressive increase in throughput and reduction of transaction latency while maintaining the strongest possible consistency level.

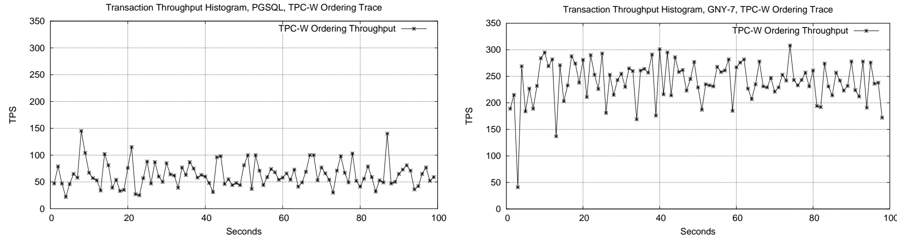

Fig. 4. Example histograms for the TPC-W Ordering Mix.

It must be noted that in our setup all replicas were identical. By having more specialized index structures on the non-master replicas the execution of read-only transactions could be optimized even more. We are exploiting this option as part of future work.

5.5 Part 2: Reaction to a failing slave replica

In this experiment, the schedulers reaction to a failing slave replica was investigated. A GNY-4 system was configured and the load generator was attached with the TPC-W shopping mix trace.

After the experiment run for a while, one of the the slave replicas was stopped, by killing the PostgreSQL process with a kill (*SIGKILL*) signal. It must be emphasized that this is different from the usage of a *SIGTERM* signal, since in that case the PostgreSQL software would have had a chance to catch the signal and shutdown gracefully.

Figure 5 shows the generated histogram for this experiment. In second 56, a slave replica was killed as described above. The failure of the slave replica led to an abort rate of 39 read-only transactions in second 56, otherwise no transaction was aborted in this run. The arrows in the graph show the change of the average transaction throughput per second. Clearly, the system's performance degraded to that of a GNY-3 setup. As can be seen from the graph, the system recovered immediately. Transactions running on the failing replica were aborted, but otherwise the system continued working normally. This is a consequence of the lightweight structure of the Ganymed scheduler approach: if a replica fails, no costly consensus protocols have to be executed. The system just continues working with the remaining replicas.

5.6 Part 3: Reaction to a failing master replica

In the last experiment, the schedulers behavior in case of a failing master replica was investigated. As in the previous experiment, the basic configuration was a GNY-4 system fed with a TPC-W shopping mix trace. Again, a SIGKILL signal was used to kill the PostgreSQL database system, this time on the master replica.

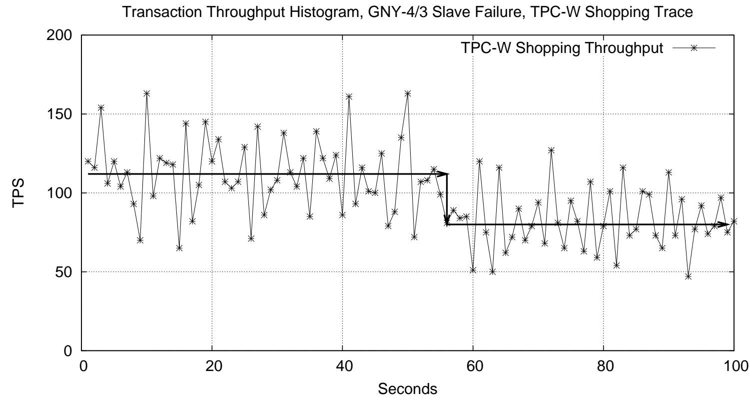

Fig. 5. Ganymed reacting to a slave replica failure.

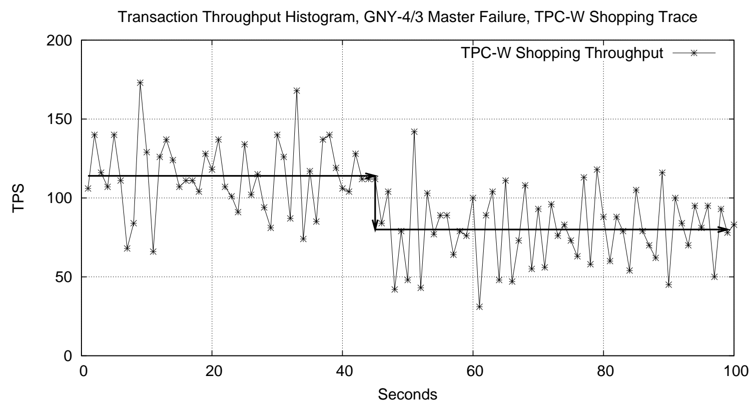

Fig. 6. Ganymed reacting to a master replica failure.

Figure 6 shows the resulting histogram for this experiment. Transaction processing is normal until in second 45 the master replica stops working. The immediately move of the master role to a slave replica leaves a GNY-3 configuration with one master and two slave replicas. The failure of the master replica led to an abort of 2 update transactions, no other transactions were aborted during the experiment. As before, the arrows in the graph show the change of the average transaction throughput per second.

This experiment shows that Ganymed is also capable of handling failing master replicas. The system reacts by reassigning the master role to a different, still working slave replica. It is important to note that the reaction to failing replicas can be done by the scheduler without intervention from the manager console. Even with a failed or otherwise unavailable manager console the scheduler can still disable failed replicas and, if needed, move the master role autonomously.

6 Related work

Work in Ganymed has been mainly influenced by C-JDBC [5], an open source database cluster middleware that can be accessed by any Java application that uses the supplied JDBC driver. C-JDBC is a general replication tool, the only assumption it makes about the database replicas is accessibility through a vendor supplied JDBC driver. The downside of this approach is the need for duplicating logic from the backend databases into the middleware, since JDBC does not supply mechanisms to achieve a fine grained control over a replica. An example for this is *locking*, which has to be done at the middleware level by parsing the incoming statements and then doing table-level locking. Another example are writesets, since these are not supported by JDBC, the middleware has to broadcast SQL update statements to all replicas. When encountering peaks of updates, this leads to a situation where every replica has to evaluate the same update statements. Ganymed behaves differently under such loads: all update statements go to the master, freeing capacity on the read-only replicas since these install only the resulting writesets. To circumvent these scalability problems, C-JDBC offers also the partition of the data on the backend replicas in various ways (called *RAIDb-levels*, in analogy to the RAID concept). As already noted, static partitions of data have the disadvantage of restricting the queries that can be executed.

Distributed versioning is an approach introduced in [2]. Again, the key idea is to use a middleware based scheduler which accepts transactions from clients and routes them to a set of replicas. The main idea is the bookkeeping of *versions* of tables in all the replicas. Every transaction that updates a table increases the corresponding version number. At the beginning of every transaction, clients have to inform the scheduler about the tables they are going to access. The scheduler then uses this information to assign versions of tables to the transactions. Similar to the C-JDBC approach, SQL statements have to be parsed to be able to do locking at the scheduler level. Also, replicas are kept in sync by sending the full SQL update statements. Since table level locking reduces concurrency, distributed versioning also introduces the concept of *early version releases*. This allows clients to notify the scheduler when they have used a table for the last time in a transaction.

Group communication has been proposed for use in replicated database systems [12, 20, 21], however only a few working prototypes are available. Postgres-R [11] is an implementation of such a system, based on a modified version of PostgreSQL (v. 6.4.2). The clear advantage of these approaches is the avoidance of centralized components. Unfortunately, in case of bursts of update traffic, this becomes a disadvantage, since the system is busy resolving conflicts between the replicas. In the worst case, such systems behave slower than a single instance database. A solution to the problem of high conflict rates in group communication systems is the partition of the load [4]. In this approach, although all replicas hold the complete data set, update transaction cannot be executed on every replica. For every transaction it has to be predeclared which elements in the database it is going to update (so called *conflict classes*). Depending on this

set of conflict classes, a so called *compound conflict class* can be deduced. Every possible compound conflict class is statically assigned to a replica, replicas are said to act as *master site* for assigned compound conflict classes. Incoming update transactions are broadcasted to all replicas using group communication, leading to a total order. Each replica decides then if it is the master site for a given transaction. Master sites execute transactions, other sites just install the resulting writesets, using the derived total order.

IBM and Microsoft [22, 23, 24] have recently proposed solutions for systems where web applications are distributed over the network. The main idea is to do caching at the edge of the network, which leads to improved response times. Of course, full consistency has to be given up. To the application servers, the caches look like an ordinary database system. Some of these approaches are not limited to static cache structures, they can react to changes in the load and adapt the amount of data kept in the cache.

7 Conclusions and Further Work

This paper has presented a novel algorithm for the replication of databases at the middleware level (RSI-PC) as well as the the design, implementation and evaluation of a replication platform (Ganymed) based on this algorithm. Although the algorithm may appear to be conceptually simple, it combines subtle insights in how real databases work and several optimizations that make it extremely efficient. The resulting system is light weight and avoids the limitations of existing solutions. For instance, the fact that Ganymed does not use group communication helps both with scalability and fast reaction to failures. It also reduces the footprint of the system. Ganymed does not duplicate database functionality either. It performs neither high level concurrency control (which typically implies a significant reduction in concurrency since it is done at the level of table locking) nor SQL parsing (which is a very expensive operation for real loads). This is an issue that needs to be emphasized as the redundancy is not just a matter of footprint or efficiency. We are not aware of any proposed solution that duplicates database functionality (be it locking, concurrency control, or SQL parsing) that can support real database engines. The problem of these designs is that they assume the middleware layer can control everything which happens on a database engine. This is, however, not a correct assumption as concurrency control affects more than just tables (e.g., recovery procedures, indexes) and, to work correctly, important database functionality such as triggers would have to be disabled or the concurrency control at the middleware level had to work at an extremely conservative level, thereby slowing down the system. In the same spirit, Ganymed imposes no data organization, structuring of the load, or particular arrangements of the schema. Finally, applications that want to make use of Ganymed do not have to be modified, the JDBC driver approach guarantees the generality of the interface that Ganymed offers. Since all existing solution suffer from one or more of these problems, Ganymed represents a significant step forward in database replication.

Thanks to the minimal infrastructure needed, Ganymed provides excellent scalability and reliability. We have shown that for typical benchmarks Ganymed scales almost linearly. Even if replicas fail, Ganymed is able to continue working with proper performance levels thanks to the simple mechanisms involved in recovery. It is also important to note that the results provided are a lower bound in terms of scalability. There are many optimizations possible that will increase scalability even further. As part of future work, we are exploring the use of specialized indexes in the read-only replicas to speed up query processing. We are also studying the possibility of autonomic behavior in the creation of such indexes. For instance, the manager console could adapt these indexes dynamically as a result of inspecting the current load. Given the low overhead of the infrastructure, we can invest in such optimizations without worrying about the impact of the extra computations on performance. In the medium term, the implementation of complete autonomous behavior is an important goal. This would affect the automatic creation of read-only copies as needed, supporting several database instances in a cluster setting, etc. One can think of scenarios with multiple, independent schedulers responsible for different logical databases and large sets of replicas. The replicas could then be automatically assigned to schedulers, e.g., based on load observations, charging schemas (e.g., *pay per TPS*) or business rules. Such a system would enable the creation of large scale database provisioning services. Again, the fact that the infrastructure has a very low overhead makes many of these approaches feasible.

References

1. Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.
2. Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-End Databases of Dynamic Content Web Sites. In *Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003, Proceedings*, 2003.
3. Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.
4. R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *IEEE 22nd Int. Conf. on Distributed Computing Systems, ICDCS'02, Vienna, Austria*, pages 477–484, July 2002.
5. Emmanuel Cecchet, Julie Marguerite, Mathieu Peltier, and Nicolas Modrzyk. C-JDBC: Clustered JDBC, http://c-jdbc.objectweb.org.
6. Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1–10, May 1995.
7. Ralf Schenkel and Gerhard Weikum. Integrating Snapshot Isolation into Transactional Federation. In Opher Etzion and Peter Scheuermann, editors, *Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel,*

September 6-8, 2000, Proceedings, volume 1901 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2000.
8. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
9. Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems*. Morgan Kaufmann Publishers, 2002.
10. R. Jiménez-Peris and M. Patiño-Martínez and G. Alonso and B. Kemme. Are Quorums an Alternative for Data Replication? *ACM Transactions on Database Systems*, 2003.
11. Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication. In *Proceedings of the 26th International Conference on Very Large Databases, 2000.*
12. Bettina Kemme. Implementing Database Replication based on Group Communication. In *Proc. of the International Workshop on Future Directions in Distributed Computing (FuDiCo 2002), Bertinoro, Italy*, June 2002.
13. Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7. Oracle White Paper, July 1995.
14. PostgreSQL Global Development Group. PostgreSQL: The most advanced Open Source Database System in the World. http://www.postgresql.org.
15. Alan Fekete, Dimitros Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Sasha. Making Snapshot Isolation Serializable, http://www.cs.umb.edu/ isotest/snaptest/snaptest.pdf.
16. Alan D. Fekete. Serialisability and Snapshot Isolation. In *Proceedings of the Australian Database Conference*, pages 210–210, January 1999.
17. Oracle Streams Concepts and Administration, Oracle Database Advanced Replication, 10g Release 1 (10.1). Oracle Database Documentation Library, 2003.
18. The Transaction Processing Performance Council. TPC-W, a Transactional Web E-Commerce Benchmark. http://www.tpc.org/tpcw/.
19. Mikko H. Lipasti. Java TPC-W Implementation Distribution of Prof. Lipasti's Fall 1999 ECE 902 Course, http://www.ece.wisc.edu/ pharm/.
20. Y. Amir and C. Tutu. From Total Order to Database Replication. Technical report, CNDS, 2002.
21. Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.
22. Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In *Proceedings of the 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany.*
23. K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A Dynamic Data Cache for Web Applications. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India.*
24. Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. Transparent Mid-tier Database Caching in SQL Server. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 661–661. ACM Press, 2003.