Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Cluster Oriented Protocols Report

GORDA Deliverable D3.2

Due date of deliverable: 2006/03/31
Actual submission date: 2006/10/01
Revision 1.1 date: 2007/04/15

Start date of project: 1 October 2004 Duration: 42 Months

UNISI

Revision 1.1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contributors

Fernando Pedone, U. Lugano
Vaide Zuikeviciute, U. Lugano

Abstract

This document reports on the work that has been performed on the evaluation, selection and development of strong consistent, group based, database replication protocols suited for cluster settings. We chose to select protocols representative of three different categories: the Database State Machine protocol and a novel extension, the NOn-Disjoint conflict classes and Optimistic Multicast, and the Sequoia (previously known as C-JDBC) protocols. In this report we describe each protocol in detail, with particular attention to the vDBSM extension to DBSM which has been designed in the context of the project.

Together with the description of the vDBSM and load balancing techniques that boost the performance of the DBSM-like protocols, we present a preliminary performance evaluation of these techniques. Current work is considering a more detailed protocol implementation of the vDBSM as well as the development of a novel hybrid protocol aimed at gracefully handling specific subsets of real-world workloads. These results will soon be presented as an addendum to this report.

Contents

Chapter 1

Introduction

This document describes GORDA strong consistent database replication protocols for databases connected within a cluster. The main goal of this deliverable is to present and discuss state-of-the-art solutions that aim at both performance and high-availability. We focus, in the following sections, on group communication based cluster protocols representative of three different categories of replication techniques. First we consider certification-based protocols which execute transactions optimistically and follow the passive replication approach (each transactions is executed by one replica and the state updates then propagated to the others), then we describe a conservative execution protocol also embodying the passive replication approach, and finally a conservative execution protocol where all replicas are called to execute the transactions (active replication).

Two new developments of GORDA applicable to certification-based protocols are presented:

- *Multiversion Database State Machine (vDBSM).* The vDBSM is an extension of the original DBSM. Its design was driven by simplicity, requiring minimal modifications to the internals of a database. Any database complying to the GORDA API can be used to implement the vDBSM.
- *Conflict-aware load balancing.* While load balancing in general is not a replication protocol in itself, the techniques developed here are important because they show how the performance of a DBSM-like protocol—the main model adopted in GORDA—can be boosted by taking concurrency control issues into account when scheduling transactions for execution.

The remainder of this document is structured as follows: Chapter 2 presents the Database State Machine and its extension, the Multiversion Database State Machine, Chapter 3 introduces conflict-aware load balancing. Chapter 4 reviews the NODO protocol and Chapter 4 the Sequoia protocol.

1.1 Objectives

The goals of the work reported in this document are as follows:

- Evaluate and select synchronous database replication protocols suited for cluster settings;
- Design appropriate protocols targeting specific workloads or addressing existing protocols drawbacks;
- Predict the performance of the new protocols.

1.2 Relationship With Other Deliverables

This deliverable departs from the work on D1.1 - State of the Art Report and, to some extent, the choices herein are influenced by D1.2 - User Requirements Report. It is instrumental on shaping the work to be delivered with D3.3 - Replication Modules Reference Implementation. The current deliverable has two companion reports: D3.1 - Wide-Area Protocols Report and D3.5 - Group Communication Protocols.[1]

[1] Albeit not present in the actual DoW of the project, the role of group communication in supporting reliable, ordered and secure communication to all the developed protocols and the large effort devoted to the APPIA toolkit, justifies a report on its own.

Chapter 2

Database State Machine

This section focuses on the Database State Machine (DBSM) approach [9] and introduce the Multiversion Database State Machine (vDBSM) [13]. In the following, we first describe the system and database model considered, then we introduce the original DBSM and afterwards we present our approach.

2.1 System and database considerations

We consider an asynchronous distributed system composed of database clients, $c_1, c_2, ..., c_m$, and servers, $S_1, S_2, ..., S_n$. Communication is by message passing. Servers can also interact by means of a total-order broadcast, described below. Servers can fail by crashing and subsequently recover. If a server crashes and never recovers, then operational servers eventually detect the crash.

Total-order broadcast is defined by the primitives broadcast(m) and deliver(m), and guarantees that (a) if a server delivers a message m then every server delivers m; (b) no two servers deliver any two messages in different orders; and (c) if a server broadcasts message m and does not fail, then every server eventually delivers m.

Each server has a full copy of the database. Servers execute *transactions* according to strict two-phase locking (2PL)[3]. Transactions are sequences of read and write operations followed by a commit or an abort operation. A transaction is called *read-only* if it does not contain any write operation; otherwise it is called an *update* transaction.

The database *workload* is composed of a set of transactions $\mathcal{T} = \{T_1, T_2, ...\}$. To account for the computational resources needed to execute different transactions, each transaction T_i in the workload can be assigned a *weight* w_i. For example, simple transactions could have less weight than complex transactions.

2.2 DBSM

The state-machine approach is a non-centralized replication technique [11]. Its key concept is that all replicas receive and process the same sequence of requests in the same order. Consistency is guaranteed if replicas behave deterministically, that is, when provided with the same input (e.g., a request) each replica will produce the same output (e.g., state change).

The Database State Machine uses the state-machine approach to implement deferred update replication. Each transaction is executed locally on some server and during the execution there is no interaction between replicas. Read-only transactions are committed locally. Update transactions are broadcast to all replicas for certification. If the transaction passes certification, it is committed; otherwise it is aborted. Certification ensures that the execution is *one-copy serializable (1SR)*, that is, every concurrent execution is equivalent to some serial execution of the same transactions using a single copy of the database.

At certification, the transaction's readsets, writesets, and updates are broadcast to all replicas. The *readsets* and the *writesets* identify the data items read and written by the transactions; they do not contain the values read and written. The transaction's *updates* can be its redo logs or the rows it modified and created. All servers deliver the same transactions in the same order and certify the transactions deterministically. Notice that the DBSM does not require the execution of transactions to be deterministic; only the certification test and the application of the transaction updates to the database are implemented as a state machine.

Transactions pass through well-defined states while being processed. Transactions start in the *executing* state, during which their read and write operations are performed. When the commit operation is requested, the transaction passes to the *committing* state and remains in it until its fate is decided by the database server; depending on the decision, the transaction passes to the *committed* or the *aborted* state. The committed and the aborted states are final.

The DBSM has several advantages when compared to existing replication schemes. In contrast to lazy replication techniques, the DBSM provides strong consistency (i.e., serializability) and fault tolerance. When compared with primary-backup replication, it allows transaction execution to be done in parallel on several replicas, which is ideal for workloads populated by a large number of non-conflicting update transactions. By avoiding distributed locking used in synchronous replication, the DBSM scales to a larger number of nodes. Finally, when compared to active replication, it allows better usage of resources because each transaction is executed by a single node.

2.3 vDBSM

The DBSM depends on transaction readsets and writesets, needed for certification. Extracting readsets usually implies changing the database internals or parsing SQL statements outside the database; extracting writesets is easier: writesets tend to be much smaller than readsets and can be obtained during transaction processing (e.g., using triggers). In this work we explore alternative ways to implement the DBSM model without requiring readsets and writesets, and present the Multiversion Database State Machine, a replication protocol that can be implemented on top of any database that complies with the GORDA API, but without requiring readsets and writesets.

The vDBSM assumes pre-defined, parameterized transactions. The particular data items accessed by the transaction depend on the transactions type and the parameters provided by the application program, when the transaction is instantiated. Predefined transactions are common in many current database applications. By estimating the data items accessed by transactions before their execution, even if conservatively, the replication protocol is spared from extracting readsets and writesets during the execution. In the case of vDBSM, this has also resulted in a certification test simpler than the one used by the original DBSM.

We denote the replica where T_i executes, its readset, and its writeset by $server(T_i)$, $readset(T_i)$, and $writeset(T_i)$, respectively. The vDBSM protocol works as follows:

1. We assign to each data item in the database a version number. Thus, besides storing a full copy of the database, each replica S_k also has a vector V_k of version numbers. The current version of data item d_x at S_k is denoted by $V_k[x]$.

2. Read-only or update transactions can execute on any replica.

3. During the execution, the versions of the data items read by an update transaction are collected. We denote by $V(T_i)[x]$ the version of each data item d_x read by T_i. The versions of the data items read by T_i are broadcast to all replicas together with its readset, writeset, and updates at commit time. T_i's updates are its SQL update statements.

4. Upon delivery, update transactions are certified. Transaction T_i passes certification if all data items it read during its execution are still up-to-date at certification time. More formally, T_i passes certification if the following condition holds:

$$\forall d_x \in readset(T_i) : \; V_k[x] = V(T_i)[x]$$

5. If T_i passes certification, its update SQL statements are submitted to the database, and the version numbers of the data items it wrote are incremented. Replicas must ensure that transactions commit in the same order.

We say that two transactions T_i and T_j *conflict*, denoted $T_i \sim T_j$, if they access some common data item, and one transaction reads the item and the other writes it. If T_i and T_j conflict and are executed concurrently on different servers, certification may abort one of them. If they execute on the same replica, however, the replica's scheduler will order T_i and T_j appropriately, and thus, both can commit.

Therefore, if transactions with similar access patterns execute on the same server, the local replica's scheduler will serialize conflicting transactions and decrease the number of aborts. Based on the transaction types, their parameters and the conflict relation, we assign transactions to *preferred servers*.

The vDBSM ensures consistency (i.e., one-copy serializability) regardless of the server chosen for the execution of a transaction. However, executing update transactions on their preferred servers can reduce the number of certification aborts.

Chapter 3

Conflict-aware load balancing

Assigning transactions to preferred servers is an optimization problem. It consists in distributing the transactions over the replicas $S_1, S_2, ..., S_n$. When assigning transactions to database servers, we aim to (a) minimize the number of conflicting transactions in distinct replicas, and (b) maximize the parallelism between transactions. These are opposite requirements. While the first can be satisfied by concentrating transactions on few database servers, the second is fulfilled by spreading transactions on multiple replicas. In Sections 3.1 and 3.2, we present two greedy algorithms that assign transactions to preferred servers. Each one prioritizes a different requirement.

Our load-balancing algorithms can be executed *statically*, before transactions are submitted to the system, or *dynamically*, during transaction processing, for each transaction when it is submitted. Static load balancing requires knowledge of the transaction types, the conflict relation, and the weight of transactions. Dynamic load balancing further requires information about which transactions are in execution on the servers.

3.1 Minimizing Conflicts First (MCF)

MCF attempts to minimize the number of conflicting transactions assigned to different replicas. The algorithm initially tries to assign each transaction T_i in the workload to the replica containing conflicting transactions with T_i. If more than one option exists, the algorithm strives to distribute the load among the replicas equitably, maximizing parallelism.

1. Consider replicas S_1, S_2, ..., S_n. With an abuse of notation, we say that transaction T_i belongs to S_k^t at time t, $T_i \in S_k^t$, if at time t T_i is assigned to execute on server S_k.

2. For each transaction T_i in the workload, to assign T_i to some server at time t execute step 3, if T_i is an update transaction, or step 4, if T_i is a read-only transaction.

3. Let $C(T_i, t)$ be the set of replicas containing transactions conflicting with T_i at time t, defined as $C(T_i, t) = \{S_k \mid \exists T_j \in S_k^t \text{ such that } T_i \sim T_j\}$.

   (a) If $|C(T_i, t)| = 0$ then assign T_i to the replica S_k with the lowest aggregated weight $w(S_k, t)$ at time t, where $w(S_k, t) = \sum_{T_j \in S_k^t} w_j$.

   (b) If $|C(T_i, t)| = 1$, assign T_i to the replica in $C(T_i, t)$.

   (c) If $|C(T_i, t)| > 1$, then assign T_i to the replica in $C(T_i, t)$ with the highest aggregated weight of transactions conflicting with T_i; if several replicas in $C(T_i, t)$ satisfy this condition, assign T_i to any one of these.

More formally, let $C_{T_i}(S_k^t)$ be the subset of S_k^t containing conflicting transactions with T_i only: $C_{T_i}(S_k^t) = \{T_j \mid T_j \in S_k^t \wedge T_j \sim T_i\}$. Assign T_i to the replica S_k in $C(T_i, t)$ with the greatest aggregated weight $w(C_{T_i}(S_k^t)) = \sum_{T_j \in C_{T_i}(S_k^t)} w_j$.

4. Assign read-only transaction T_i to the replica S_k with the lowest aggregated weight $w(S_k, t)$ at time t, where $w(S_k, t)$ is defined as in step 3(a).

3.2 Maximizing Parallelism First (MPF)

MPF prioritizes parallelism between transactions. Consequently, it initially tries to assign transactions in order to keep the servers' load even. If more than one option exists, the algorithm attempts to minimize conflicts. The load of a server is given by the aggregated weight of the transactions assigned to it at some given time. To compare the load of two servers, we use factor $f, 0 < f \leq 1$. We denote MPF with a factor f as MPF f. Servers S_i and S_j have similar load at time t if the following condition holds: $f \leq w(S_i, t)/w(S_j, t) \leq 1$ **or** $f \leq w(S_j, t)/w(S_i, t) \leq 1$.

1. Consider replicas S_1, S_2, ..., S_n. To assign each transaction T_i in the workload to some server at time t execute steps 2–4, if T_i is an update transaction, or step 5, if T_i is a read-only transaction.

2. Let $W(t) = \{S_k \mid w(S_k, t) * f \leq \min_{l \in 1..n} w(S_l, t)\}$ be the set of replicas with minimal load at time t, where $w(S_l, t)$ has been defined in step 3(a) in Section 3.1.

3. If $|W(t)| = 1$ then assign T_i to the replica in $W(t)$.

4. If $|W(t)| > 1$ then let $C_W(T_i, t)$ be the set of replicas containing conflicting transactions with T_i in $W(t)$: $C_W(T_i, t) = \{S_k \mid S_k \in W(t)$ and $\exists T_j \in S_k$ such that $T_i \sim T_j\}$.

   (a) If $|C_W(T_i, t)| = 0$, assign T_i to the S_k in $W(t)$ with the lowest aggregated weight $w(S_k, t)$.

   (b) If $|C_W(T_i, t)| = 1$, assign T_i to the replica in $C_W(T_i, t)$.

   (c) If $|C_W(T_i, t)| > 1$, assign T_i to the replica S_k in $C_W(T_i, t)$ with the highest aggregated weight $w(C_{T_i}(S_k^t))$ counting only transactions conflicting with T_i, $w(C_{T_i}(S_k^t))$ is formally defined in step 3 in Section 3.1; if several replicas in $C_W(T_i, t)$ satisfy this condition, assign T_i to any one of these.

5. Assign read-only transaction T_i to the replica S_k with the lowest aggregated weight $w(S_k, t)$ at time t.

Notice that the MCF algorithm is a special case of MPF with a factor $f = 0$.

3.2.1 Static load balancing

A static load balancer executes MCF and MPF offline, considering each transaction in the workload at a time in some order—for example, transactions can be considered in decreasing order of weight, or according to some time distribution, if available. Since the assignments are pre-computed, during the execution there is no need for the replicas to send feedback information to the load balancer. The main drawback of this approach is that it can potentially make poor assignment decisions.

We now illustrate static load balancing of MCF and MPF. Consider a workload with 10 transactions, $T_1, T_2, ..., T_{10}$, running in a system with 4 replicas. Transactions with odd index conflict with transactions with odd index; transactions with even index conflict with transactions with even index. Each transaction T_i has weight $w(T_i) = i$.

By considering transactions in decreasing order of weight, MCF will assign transactions T_{10}, T_8, T_6, T_4, and T_2 to S_1; T_9, T_7, T_5, T_3, and T_1 to S_2; and no transactions to S_3 and S_4. MPF 1 will assign T_{10}, T_3, and T_2 to S_1; T_9, T_4, and T_1 to S_2; T_8 and T_5 to S_3; and T_7 and T_6 to S_4. MPF 0.8 will assign T_{10}, T_4, and T_2 to S_1; T_9 and T_3 to S_2; T_8 and T_6 to S_3; and T_7, T_5, and T_1 to S_4.

MPF 1 creates a balanced assignment of transactions. The resulting scheme is such that $w(S_1) = 15, w(S_2) = 14, w(S_3) = 13$, and $w(S_4) = 13$. Conflicting transactions are assigned to all servers however. MCF completely concentrates conflicting transactions on distinct servers, S_1 and S_2, but the aggregated weight distribution is poor: $w(S_1) = 30, w(S_2) = 25, w(S_3) = 0$, and $w(S_4) = 0$, that is, two replicas would be idle. MPF 0.8 is a compromise between the previous schemes. Even transactions are assigned to S_1 and S_3, and odd transactions to S_2 and S_4. The aggregated weight is fairly balanced: $w(S_1) = 16, w(S_2) = 12, w(S_3) = 14$, and $w(S_4) = 13$.

Dynamic load balancing

Dynamic load balancing can potentially outperform static load balancing by taking into account information about the execution of transactions when making assignment choices. Moreover, the approach does not require any pre-processing since transactions are assigned to replicas on-the-fly, as they are submitted. As a disadvantage, a dynamic scheme requires feedback from the replicas with information about the execution of transactions. Receiving and analyzing this information may introduce overheads. MCF and MPF can be implemented in a dynamic load balancer as follows: The load balancer keeps a local data structure $S[1..n]$ with information about the current assignment of transactions to each server. Each transaction in the workload is considered at a time, when it is submitted by the client, and assigned to a server according to MCF or MPF. When a replica S_k finishes the execution of a transaction T_i, committing or aborting it, S_k notifies the load balancer. Upon receiving the notification of termination from S_k, the load balancer removes T_i from $S[k]$.

3.3 Static vs. dynamic load balancing

A key difference between static and dynamic load balancing is that the former will only be effective if transactions are pre-processed in a way that resembles the real execution. For example, assume that a static assignment considers that all transactions are uniformly distributed over a period of time, but in reality some transaction types only occur in the first half of the period and the other types in the second half. Obviously, this is not an issue with dynamic load balancing.

Another aspect that distinguishes static and dynamic load balancing is membership changes, that is, a new replica joins the system or an existent one leaves the system (e.g., due to a crash). Membership changes invalidate the assignments of transactions to servers. Until MCF and MPF are updated with the current membership, no transaction will be assigned to a new replica joining the system, for example. Therefore, with static load balancing, the assignment of preferred servers has to be recalculated whenever the membership changes. Notice that adapting to a new membership is done for performance, and not consistency since the certification test of the vDBSM does not rely on transaction assignment information to ensure one-copy serializability; the consistency of the system is always guaranteed, even though out of date transaction assignment information is used.

Adjusting MCF and MPF to a new system membership using a dynamic load balancer is straightforward: as soon as the the new membership is known by the load balancer, it can update the number of replicas in either MCF or MPF and start assigning transactions correctly. With static load balancing, a new membership requires executing MCF or MPF again for the complete workload, which may take some time. To speed up the calculation, transaction assignments for configurations with different number of "virtual replicas" can be done offline. Therefore, if one replica fails, the system switches to a pre-calculated assignment with one replica less. Only the mapping between virtual replicas to real ones has to be done online.

3.4 Preliminary performance evaluation

3.4.1 Analysis of the TPC-C benchmark

In this section we overview the TPC-C benchmark, show how it can be mapped to our transactional model, and provide some preliminary results of vDBSM using MCF and MPF.

TPC-C is an industry standard benchmark for online transaction processing (OLTP) [12]. It represents a generic wholesale supplier workload. The benchmark's database consists of a number of warehouses, each one composed of ten districts and maintaining a stock of 100000 items; each district serves 3000 customers. All the data is stored in a set of 9 relations: *Warehouse, District, Customer, Item, Stock, Orders, Order Line, New Order,* and *History*.

TPC-C defines five transaction types: *New Order*, *Payment*, *Delivery*, *Order Status* and *Stock Level*. *Order Status* and *Stock Level* are read-only transactions; the others are update transactions. Since only update transactions count for conflicts—read-only transactions execute at preferred servers just to balance the load—there are only three update transaction types to consider: *Delivery* (D), *Payment* (P), and *New Order* (NO). These three transaction types compose 92% of TPC-C workload.

In the following we define the workload of update transactions as:

$$\begin{aligned}\mathcal{T} \;=\; \{D_i, P_{ijkm}, NO_{ijS} \mid\; & i,k \in 1..\#\text{WH};\\ & j,m \in 1..10;\\ & S \subseteq \{1,...,\#WH\}\}\end{aligned}$$

where #WH is the number of warehouses considered. D_i stands for a *Delivery* transaction accessing districts in warehouse i. P_{ijkm} relates to a *Payment* transaction which reflects the payment and sales statistics on the district j and warehouse i and updates the customer's balance. In 15% of the cases, the customer is chosen from a remote warehouse k and district m. Thus, for 85% of transactions of type P_{ijkm}: $(k = i) \wedge (m = j)$. NO_{ijS} is a *New Order* transaction referring to a customer assigned to warehouse i and district j. For an order to complete, some items must be chosen: 99% of the time the item chosen is from the home warehouse i and 1% of the time from a remote warehouse. S is a set of remote warehouses.

To assign a particular update transaction to a replica, we have to analyze the conflicts between transaction types. Our analysis is based on the warehouse and district numbers only. For example, *New Order* and *Payment* transactions might conflict if they operate on the same warehouse. We define the conflict relation $\sim$ between transaction types as follows:

$$\begin{aligned}\sim \;=\; & \{(D_i, D_x) \mid (x = i)\} \cup\\ & \{(D_i, P_{xykm}) \mid (k = i)\} \cup\\ & \{(D_i, NO_{xyS}) \mid (x = i) \vee (i \in S)\} \cup\\ & \{(P_{ijkm}, P_{xyzq}) \mid (x = i) \vee ((z = k) \wedge (q = m))\} \cup\\ & \{(NO_{ijS}, NO_{xyZ}) \mid\\ & ((x = i) \wedge (y = j)) \vee (S \cap Z \neq \emptyset)\} \cup\\ & \{(NO_{ijS}, P_{xyzq}) \mid (x = i) \vee ((z = i) \wedge (q = j))\}\end{aligned}$$

For example, two *Delivery* transactions conflict if they access the same warehouse.

Notice that we do not have to consider every transaction that may happen in the workload in order to define the conflict relation between transactions. Only the transaction types and how they relate to each other should be taken into account. To keep our characterization simple, we will assume that the weights associated with the workload represent the frequency in which transactions of some type may occur in a run of the benchmark.

We are interested in the system's load distribution and the number of conflicting transactions executing on different replicas. To measure the load, we use the aggregated weight of all transactions assigned to each replica. To measure the conflicts, we use the *overlapping ratio* $O_R(S_i, S_j)$ between database servers S_i and S_j, defined as the ratio between the aggregated weight of update transactions assigned to

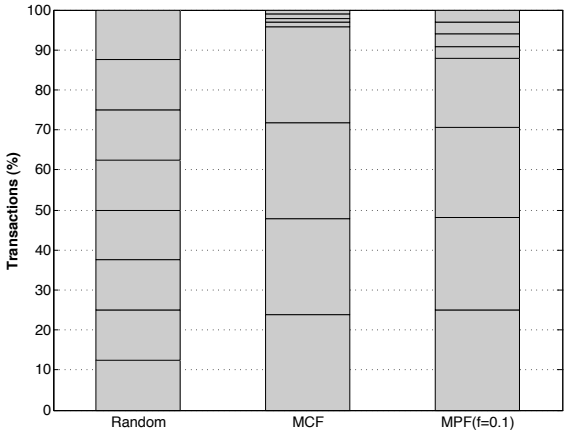


Figure 3.1: Load distribution over 8 replicas

S_i that conflict with update transactions assigned to S_j, and the aggregated weight of all update transactions assigned to S_i. For example, consider that T_1, T_2, and T_3 are assigned to S_i, and T_4, T_5, T_6, and T_7 are assigned to S_j. T_1 conflicts with T_4, and T_2 conflicts with T_6. Then the overlapping ratio for these replicas is calculated as $O_R(S_i, S_j) = \frac{w(T_1)+w(T_2)}{w(T_1)+w(T_2)+w(T_3)}$ and $O_R(S_j, S_i) = \frac{w(T_4)+w(T_6)}{w(T_4)+w(T_5)+w(T_6)+w(T_7)}$. Notice that since our analysis here is static, the overlapping ratio gives a measure of "potential aborts"; real aborts will only happen if conflicting transactions are executed concurrently on different servers. Clearly, a high risk of abort translates into more real aborts during the execution.

We have considered 4 warehouses (i.e., #WH = 4) and 8 database replicas in our static analysis. We compared the results of MCF, MPF 1 and MPF 0.1 with a random assignment of transactions to replicas (dubbed Random).

Random results in a fair load distribution (see Figure 3.1), but has very high overlapping ratio (see Figure 3.2). MPF 1 (not shown in the graphs) behaves similarly to Random: it distributes the load equitably over the replicas, but has high overlapping ratio.

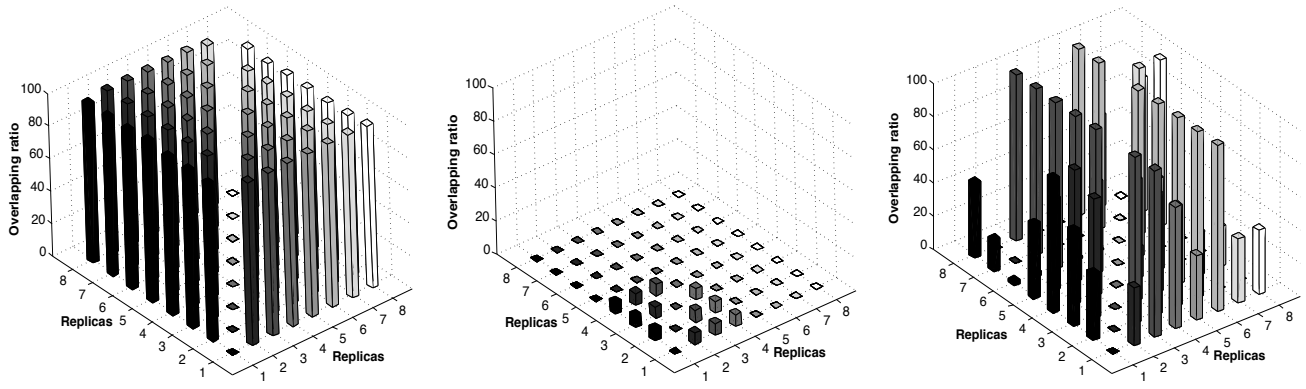


Figure 3.2: Overlapping ratio, (a) Random (b) MCF (c) MPF 0.1

MCF minimizes significantly the number of conflicts, but update transactions are distributed over 4 replicas only; the other 4 replicas execute just read-only transactions (see Figure 3.1). This is a consequence of TPC-C and the 4 warehouses considered. Even if more replicas were available, MCF would

still strive to minimize the overlapping ratio, assigning update transactions to only 4 replicas.

A compromise between maximizing parallelism and minimizing conflicts can be achieved by varying the f factor of the MPF algorithm. With $f = 0.1$ the overlap ratio is much lower than Random (and MPF 1), for example.

3.4.2 Prototype overview

We have implemented a preliminary prototype of the vDBSM in Java v.1.5.0 using both static and dynamic load balancing. Our intent at this point is to better understand the tradeoffs and bottlenecks of the proposed protocols. Ongoing work is refining this prototype.

Client applications interact with the replicated compound by submitting SQL statements through a customized JDBC-like interface. Application requests are sent directly to a database server, in case of static load balancing, or first to the load balancer and then re-directed to a server. A *replication module* in each server is responsible for executing transactions against the local database, and certifying and applying them in case of commit. Every transaction received by the replication module is submitted to the database through the standard JDBC interface.

On delivery the transaction is enqueued for certification. While transactions execute concurrently in the database, their certification and possible commitment are sequential. The current versions of the data items are kept in main memory to speed up the certification process; however, for persistency, every row in the database is extended with a version number. If a transaction passes the certification test, its updates are applied to the database and the versions of the data items written are incremented both in the database, as part of the committing transaction, and in main memory.

To ensure that all replicas commit transactions in the same order, before applying T_i's updates, the server aborts every locally executing conflicting transaction T_j. To see why this is done, assume that T_i and T_j write the same data item d_x, each one executes on a different server, T_i is delivered first, and both pass certification test. T_j already has a lock on d_x at $server(T_j)$, but T_i should update d_x first. We ensure correct commit order by aborting T_j on $server(T_j)$ and re-executing its updates later. If T_j keeps a read lock on d_x, it is a doomed transaction, and in any case it would be aborted by the certification test later.

In the case of static load balancing, the assignment of transactions to replicas is done by the replication modules and sent to the customized JDBC interface upon the first application connect. Therefore when the application submits a transaction, it sends it directly to the replica responsible for that transaction type. The dynamic load balancer is interposed between the client applications and the replication modules. The assignment of submitted transactions is computed on-the-fly based on currently executing transactions. The load balancer keeps track of each transaction's execution and completion status at the replicas. Since all application requests are routed through the load balancer, no additional information exchange is needed between replication modules and the load balancer. The load balancer does not need to know when a transaction commits at each replica, but only at the replica where the transaction was executed.

We perform worst case analysis concentrating only on update transactions for load balancing, i.e. *Stock Level* and *Order Status* transactions are assigned randomly to the replicas for execution. We evaluated the algorithms varying the number of servers from 2 to 8. Each server stores a TPC-C database, populated with data for 4 warehouses ($\approx$ 400MB database). The workload is created by a full-fledged implementation of TPC-C. According to TPC-C, each warehouse must support 10 emulated clients, thus throughout the experiments the workload is submitted by 40 concurrent clients. TPC-C specifies that between transactions, each client should have a mean think time between 5 and 12 seconds.

Experiments have two phases: the *warm-up phase* when the load is injected but no measurements are taken, and the *measurement phase* when the data is collected.

3.4.3 Static vs. dynamic load balancing

Figure 3.3 shows the number of update transactions assigned to each server during executions of the benchmark with a static load balancer and different scheduling techniques.

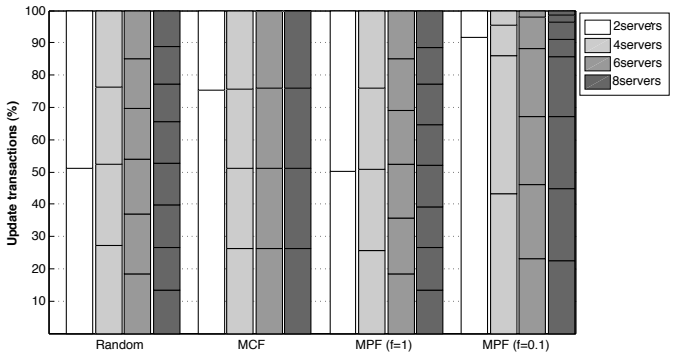


Figure 3.3: Real load distribution (static)

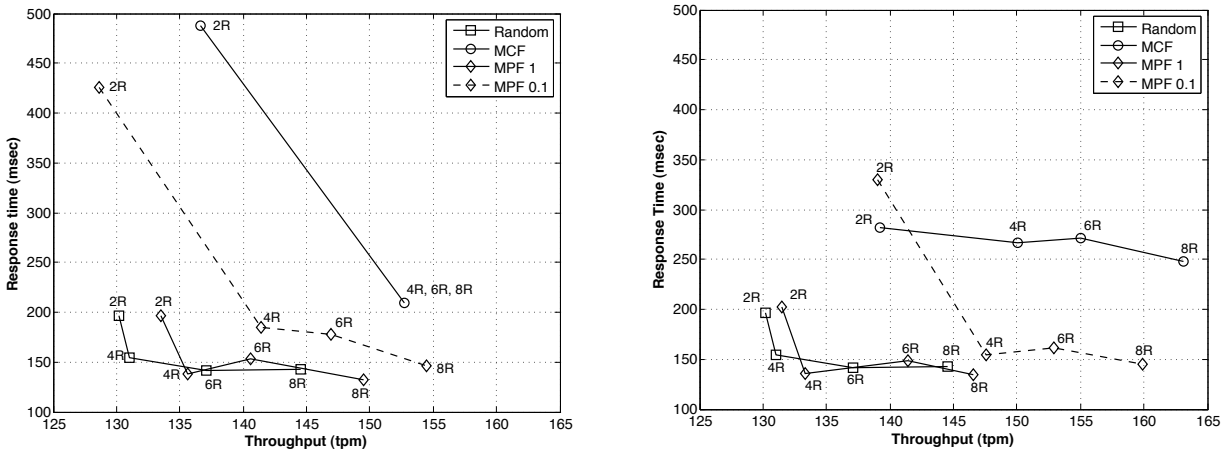
MCF and MPF 0.1 implemented with a static load balancer suffer from poor load distribution over the replicas. MCF distributes transactions over four replicas only, even when more replicas are available. MPF 0.1 achieves better load balancing than MCF with 6 and 8 replicas. Random and MPF 1 result in a fair load distribution.

Figure 3.4 shows the achieved throughput of committed transactions versus the response time for both static and dynamic load balancers. For each curve in both static and dynamic schemes the increased throughput is obtained by adding replicas to the system. From both graphs, scheduling transactions based mainly on replica load, MPF 1, results in slightly better throughput than Random, while keeping the response time constant. Prioritizing conflicts has a more noticeable effect on load balancing. With static and dynamic load balancing, MCF, which primarily takes conflicts into consideration, achieves higher throughput, but at the expense of increased response times. A hybrid load-balancing technique, such as MPF 0.1, which considers both conflicts between transactions and the load over the replicas, improves transaction throughput and only slightly increases response times with respect to Random and MPF 1.

Since in static load balancing MCF uses the same transaction assignment for 4, 6 and 8 replicas (see Figure 3.3), the throughput does not increase by adding replicas, and that is why MCF in Figure 3.4 (a) only contains two points, one for 2 replicas (2R) and another one for 4, 6 and 8 replicas (4R, 6R, 8R). Static MCF strives to minimize conflicts between replicas and assigns transactions to the same 4 servers. In this case, dynamic load balancing clearly outperforms the static one, since all available replicas are used. Finally, the results also show that except for dynamic MPF 0.1, the system is overloaded with 2 servers.

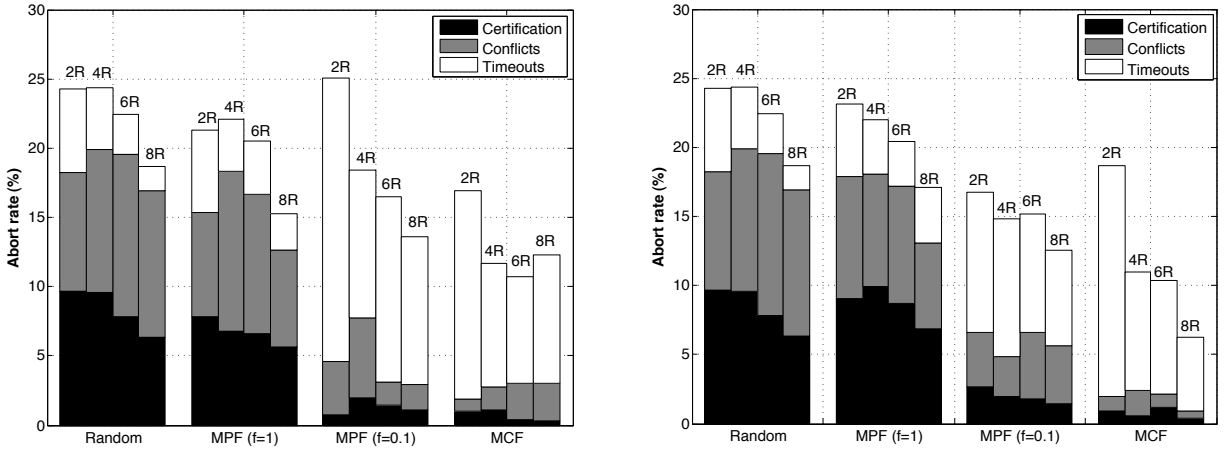
3.4.4 Abort rate breakdown

In this section we consider the abort rate breakdown for both dynamic and static load balancing (see Figure 3.5). There are three main reasons for a transaction to abort: (i) it fails the certification test, (ii) it holds locks that conflict with a committing transaction (see Section 3.4.2), and (iii) it times out after waiting for too long. Notice that aborts due to conflicts are similar in nature to certification aborts, in that they both happen due to the lack of synchronization between transactions during the execution. Thus, a transaction will never be involved in aborts of type (i) or (ii) due to another transaction executing on the same replica.



(a) (b)

Figure 3.4: Throughput vs. response time (a) static load balancing (b) dynamic load balancing



(a) (b)

Figure 3.5: Abort rates (a) static load balancing (b) dynamic load balancing

In both static and dynamic strategies with more than 2 replicas, Random and MPF 1 result in more aborts than MCF and MPF 0.1. Random and MPF 1 lead to aborts due to conflicts and certification, whereas aborts in MCF and MPF 0.1 are primarily caused by timeouts.

MCF reduces certification aborts from $\approx 20\%$ to $\approx 3\%$. However, MCF, especially with a static load balancer, results in many timeouts caused by conflicting transactions waiting for execution. MPF 0.1 with a static load balancer suffers mostly from unfair load distribution over the servers, while with a dynamic load balancing MPF 0.1 is between MPF 1 and MCF: reduced certification aborts, if compared to the former, and reduced timeouts, if compared to the latter. In the end, MCF and MPF 0.1 win because local aborts introduce lower overhead in the system than certification aborts.

3.4.5 The impact of failures and reconfigurations

We now consider the impact of membership changes due to failures in the dynamic load-balancing scheme. The scenario is that of a system with 4 replicas and one of them fails. Until the failure is detected, the load balancer continues to schedule transactions (using MPF 1 in this case) to all replicas, including the failed one. After 20 seconds, the time that it takes for the load balancer to detect the failure, only the operational three replicas receive transactions. Figure 3.6 shows the replica's failure im-

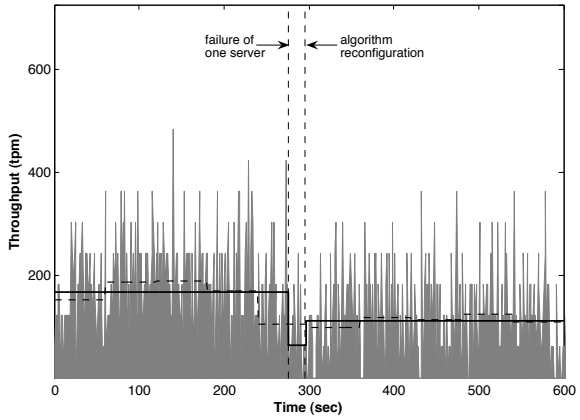


Figure 3.6: Algorithm reconfiguration

pact on committed transactions throughput with MPF 1. The solid horizontal line represents the average throughput when 4 replicas are processing transactions before the failure, during the failure, and after the failure is detected. The horizontal dashed line shows the average number of committed transactions per one-minute intervals. The throughput decreases significantly during the failure, but after the failure is detected, the load-balancing algorithm adapts to the system reconfiguration and the throughput improves. However, only 3 replicas continue functioning, so the throughput is lower than the one with 4 replicas. Our detection time of 20 seconds has been chosen to highlight the effects of failures. In practice smaller timeout values would be more adequate.

Chapter 4

NOn-Disjoint conflict classes and Optimistic multicast

This section presents a protocol introduced in [8]. The protocol's goal is to achieve the generality of a replication engine external to the database while still being able to exploit certain database specific optimizations. That way, the authors combine the best of both approaches, kernel- and middleware-based, to achieve generality and a wider flexibility in terms of performance and applications.

4.1 System model

The protocol assumes an asynchronous system extended with (possible unreliable) failure detectors in which reliable multicast with strong virtual synchrony can be implemented. The system consists of a group of sites $\mathcal{N} = \{N_1, N_2, ...N_n\}$, also called nodes, which communicate by exchanging messages. Sites only fail by crashing (Byzantine failures are excluded). There is at least one site that never crashes. Each such site is denoted as an available site. Each site contains a middleware layer and a database layer. The client submits its requests to the middleware layer which performs according operations on the database. The middleware layer instances on the different sites communicate with each other for replica control purposes. The database systems do not perform any communication. Sites are provided with a group communication system supporting strong virtual synchrony. Strong virtual synchrony ensures that messages are delivered in the same view (current connected and active sites) they were sent and that two sites transiting to a new view have delivered the same set of messages in the previous view.

Additionally, the replication protocol requires an optimistic total order multicast protocol specially tailored for transaction processing [6, 7, 10]. This optimistic multicast is defined by three primitives:

1. *TO-Multicast(m)* multicasts the message m to all the sites in the system;
2. *OPT-deliver(m)* delivers message m optimistically to the application with the same semantics as a simple reliable multicast (no ordering guarantees);
3. *TO-deliver(m)* delivers m definitively to the application with the same semantics as a total order uniform reliable multicast. That is, messages can be OPT-delivered in a different order at each available site, but are TO-delivered in the same total order at all available sites. Furthermore, this optimistic multicast primitive ensures that no site TO-delivers a message before OPT-delivering it.

A sequence of OPT-delivered messages is a tentative order. A sequence of TO-delivered messages is the definitive order or total order.

Clients interact with the system middleware by issuing application-oriented functions. Each of these functions is a pre-implemented application program consisting of several database operations. At the

time the request is submitted with a given set of parameters, both the set of operations to be executed, and the data items to be accessed within these operations is known. Such execution model reflects quite well the current use of application servers. Each application program is executed within the context of a transaction.

The authors consider one-copy serializability as the system's correctness criterion. To achieve global serializability the concurrency control for update transactions is based on conflict classes. A basic conflict class represents a partition of the data. How to partition the data is application dependent. In a simple case, there could be a class per table. If the application is well structured, other granularities are possible. Transactions accessing the same conflict class have a high probability of conflicts, as they can access the same data, while transactions in different partitions do not conflict and can be executed concurrently. Transaction can access a single basic conflict class or a compound conflict class. The authors denote as C_T T's conflict class (either basic or compound), and assume that C_T is known in advance (as mentioned above).

The middleware layer implements the above concurrency control. At each site there is a queue associated to each basic conflict class. When a transaction is delivered to a site, it is added to the queue(s) of the basic conflict class(es) it accesses. Each conflict class (basic or compound) has a master site. Transaction T is local to the master site of its conflict class, and is remote to the rest of the sites. Conflict classes are statically assigned to sites, but in case of failures, they are reassigned to different sites.

4.2 NODO protocol

The client submits a transaction T to any site using uniform reliable multicast. Then optimistic multicast is used to immediately forward the message to all the other sites. The idea is to start transaction execution upon optimistic delivery, but to use the total order established by the total order delivery as a guideline to serialize transactions. All sites see the same total order for update transactions. Thus, to guarantee correctness, it suffices for a site to ensure that conflicting transactions are ordered according to the definitive order. Since the execution order is not important for non-conflicting transactions, they can be executed in different orders (or in parallel) at different sites.

When a transaction T is optimistic delivered at site N, it is added to the queues of all basic conflict classes accessed by C_T. Only the master site of C_T executes T : whenever T is at the head of all of its queues the transaction is submitted for execution. When a transaction T is total order delivered at N, N checks whether the definitive and tentative orders agree. If they agree, T can be committed after its execution has completed. If they do not agree, there are several cases to consider. The first one is when the lack of agreement is on non-conflicting transactions. In that case, the ordering mismatch can be ignored. If the mismatch is on conflicting transactions, there are two possible scenarios. If no local transactions are involved, T can be simply rescheduled in the queues before the transactions that are only optimistic delivered but not yet total order delivered (that is the queue is reordered to reflect the total order determined by total order delivery). If local transactions are involved, the procedure is similar but a local pending transaction T' that might already have started execution (it is the first in its queue), must be aborted. Note that the algorithm does not wait to reschedule until the abort is complete but schedules T immediately before T'. Hence, T might be submitted to the database before T' has completed the abort. However, since the database has its own concurrency control, we have the guarantee that T cannot access any data item T' has written before T' undoes the change. An aborted transaction can only be resubmitted for execution once the abort is complete. Once a transaction is total order delivered and completely executed the local site multicasts the commit message including the writeset using simple, reliable multicast.

Hence, the commit message can arrive to other sites before the transaction has been total order

delivered at that site. In that case, the definitive order is not yet known, and hence, the transaction cannot commit at that site to prevent conflicting serialization orders. For this reason the processing of the commit message at a remote site is delayed until the corresponding transaction has been total order delivered at that site. Later, when the transaction has been total order delivered and it is at the head of its queues, the writeset is applied to the database and the transaction committed. Read-only transactions are only executed at the site they are submitted to.

In order for the middleware to send and apply writesets, the authors assume that the underlying database provides two services. One to obtain the writeset of a transaction and another one to apply the writeset. Executing local transactions and applying the updates of remote transactions must be controlled such as to guarantee one-copy-serializability. In particular, transactions can execute concurrently if they do not have any common basic conflict class, however, as soon as they share one basic conflict class the execution of the two transactions will be serial according to their order in the corresponding queue. Note that deadlocks cannot occur since a transaction is appended to all basic conflict classes it accesses at transaction start.

Chapter 5

Sequoia

Sequoia [5] (former C-JDBC [4]) is an open-source solution for database clustering on a shared-nothing architecture built with commodity hardware. Sequoia hides the complexity of the cluster and offers a single database view to the application. The client application does not need to be modified and transparently accesses a database cluster as if it was a centralized database. Sequoia works with any database that provides a JDBC driver.

5.1 Functionality and key components

Sequoia provides a generic JDBC driver to be used by the clients. This driver forwards the SQL requests to the Sequoia controller that balances them on a cluster of databases (reads are load balanced and writes are broadcast).

The Sequoia controller is a regular Java application made of several components that implement the logic of a RAIDb (Redundant Array of Inexpensive Databases). The controller exposes a single database view, called a virtual database, to the JDBC driver and thus to the application. Multiple virtual databases can be hosted by a controller. Each virtual database has its own request manager that defines its requests scheduling, caching and load balancing. The database backends are accessed through their native JDBC drivers.

The virtual database contains the following components:

- authentication manager: it matches the virtual database login/password (provided by the application to the Sequoia driver) with the real login/password to use on each backend. The authentication manager is only involved at connection establishment time.
- backup manager: manages a list of generic or database specific Backupers that are in charge of performing database dump and restore operation. Backupers should also take care of transferring dumps from one controller to another.
- request manager: it contains the core functionality of Sequoia controller, it handles the requests coming from a connection with a Sequoia driver. The request manager is composed of several components:
  - scheduler: it is responsible for scheduling the requests. Each RAIDb level has its own scheduler.
  - request caches: these are optional components that can cache query parsing, the result set and result metadata of queries.
  - load balancer: it balances the load on the underlying backends according to the chosen RAIDb level configuration.

  - recovery log: it handles checkpoints and allows backends to dynamically recover from a failure or to be dynamically added to a running cluster.

- database backend: it represents the real database backend running the RDBMS engine. A connection manager mainly provides connection pooling on top of the database JDBC native driver.

When a request arrives from a Sequoia driver, it is routed to the request manager associated with the virtual database. Begin transaction, commit and abort operations are sent to all backends. Reads are sent to a single backend. Updates are sent to all backends where the affected tables reside. Depending on whether full or partial replication is used this may be one, several or all database backends. All operations are synchronous with respect to the client.

The request manager waits until it has received responses from all backends involved in the operation before it returns a response to the client. If a backend executing an update, a commit or an abort fails, it is disabled. In particular, Sequoia does not use a two-phase commit protocol. Instead, it provides tools to automatically re-integrate failed backends into a virtual database (see 5.2). At any given time only a single update, commit or abort is in progress on a particular virtual database. Multiple reads from different transactions can be going on at the same time. Updates, commits and aborts are sent to all backends in the same order.

Sequoia offers various load balancers according to the degree of replication the user wants. Full replication is easy to handle. It does not require request parsing since every database backend can handle any query. Database updates, however, need to be sent to all nodes, and performance suffers from the need to broadcast updates when the number of backends increases. To address this problem, Sequoia provides partial replication in which the user can define database replication on a per-table basis. Load balancers supporting partial replication must parse the incoming queries and need to know the database schema of each backend. The schema information is dynamically gathered. When a backend is enabled, the appropriate methods are called on the JDBC database Metadata information of the backend native driver. Database schemas can also be specified statically by using a configuration file. The schema is updated dynamically on each create or drop SQL statement to accurately reflect each backend. Among the backends that can treat a read request (all of them with full replication), one is selected according to the load balancing algorithm. Currently implemented algorithms are round robin, weighted round robin and least pending requests first (the request is sent to the node that has the least pending queries).

5.2 Fault tolerance and scalability

To allow a database backend to recover after a failure or bring new backends into the system Sequoia uses checkpoints and recovery logs. A checkpoint of a virtual database can be performed at any point in time. Checkpointing can be manually triggered by the administrator or automated based on temporal rules. Taking a snapshot of a backend while the system is online requires disabling this backend so that no updates occur on it during the backup. The other backends remain enabled to answer client requests. The checkpoint procedure starts by inserting a checkpoint marker in the recovery log. Next, the database content is dumped. Then, the updates that occurred during the dump are replayed from the recovery log to the backend, starting at the checkpoint marker. Once all updates have been replayed, the backend is enabled again.
A recovery log records a log entry for each begin transaction, commit, abort and update statement. A log entry consists of the user identification, the transaction identifier and the SQL statement. The log can be stored in a flat file, but also in a database using JDBC. A fault-tolerant log can then be created by sending the log updates to a virtual Sequoia database with fault tolerance enabled.

To prevent the Sequoia controller from being a single point of failure, Sequoia provides controller replication also called horizontal scalability. A virtual database can be replicated in several controllers

that can be added dynamically at runtime. Controllers need group communication middleware to synchronize updates in a distributed way. By default, controller communication is implemented using the Appia group communication library [1], although any jGCS compliant group communication protocol [2] can be used. When a virtual database is loaded in a controller, a group name can be assigned to the virtual database. This group name is used to communicate with other controllers hosting the same virtual database. At initialization time, the controllers exchange their respective backend configurations. If a controller fails, only the backend attached to it has to be resynchronized.

To support a large number of database backends, Sequoia also provides vertical scalability that enables building a hierarchy of database backends. Furthermore, combinations of vertical and horizontal scalability are also possible.

Bibliography

[1] Appia. http://appia.di.fc.ul.pt.

[2] jGCS. http://jgcs.sourceforge.net.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proceedings of USENIX Annual Technical Conference, Freenix track*, 2004.

[5] Continuent.org. Sequoia project. http://sequoia.continuent.org.

[6] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1018–1032, July 2003.

[7] J. Mocito, A. Respicio, and L. Rodrigues. On statistically estimated optimistic delivery in wide-area total order protocols. In *Proceedings of the 12th IEEE International Symposium Pacific Rim Dependable Computing*, 2006.

[8] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems (TOCS)*, 2005.

[9] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14:71–98, 2002.

[10] L. Rodrigues, J. Mocito, and N. Carvalho. From spontaneous total order to uniform total order: different degrees of optimistic delivery. In *Proceedings of the ACM Symposium on Applied Computing*, 2006.

[11] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[12] Transaction Proccesing Performance Council (TPC). TPC benchmark C. Standard Specification, 2005. http://www.tpc.org/tpcc/spec/.

[13] V. Zuikeviciute and F. Pedone. Conflict-Aware Load-Balancing Techniques for Database Replication. Technical Report 2006/01, University of Lugano, 2006. Available at http://www.inf.unisi.ch/publications/pub.php?id=10.