Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

State of the Art in Database Replication

GORDA Deliverable D1.1

Due date of deliverable: 2005/01/31
Actual submission date: 2005/09/15
Revision date: 2006/12/30

Start date of project: 1 October 2004 Duration: 36 Months

Universidade do Minho

Revision 1.2

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

Alfrânio Correia Júnior
António Sousa
Emmanuel Cecchet
Fernando Pedone
José Pereira
Luís Rodrigues
Nuno Miguel Carvalho
Ricardo Vilaça
Rui Oliveira
Sara Bouchenak
Vaide Zuikeviciute

Contents

Chapter 1

Introduction

The goal of the GORDA project is to foster research, development and deployment of database replication solutions. This is to be achieved by standardizing architecture and interfaces, and by sparking their usage with a comprehensive set of components ready to be deployed. In detail, the project aims at:

- Promoting the interoperability of DBMSs and replication protocols by defining generic architecture and interfaces that can be standardized. Reference implement ions of the resulting implementations are provided in existing open-source DBMSs.
- Providing general-purpose and widely applicable database replication protocols for large-scale systems.
- Providing uniform techniques and tools for managing secure and heterogeneous replicated database systems. This means supporting at architectural and protocol levels complex environments by addressing partial replication, distributed execution and cross-border security issues.
- Providing an interim middleware-based solution that allows immediate integration of current DBMSs where replication interfaces cannot be natively implemented, thus supporting the standardization effort.

In this context, this document surveys existing proposals for database replication, both from industry and from academic research. A companion document provides a complementary view of the problem: Describing user requirements as identified by industrial partners [Con04].

We distinguish a replicated database management system as storing and updating multiple copies of a database at independent sites. Replication is useful mainly for improving performance, by leveraging parallel computation on different sites, and for availability, by redirecting clients to operational replicas. Database replication protocols are specific data replication protocols that support the transactional and concurrent manipulation offering ACID guarantees [BHG87]. In detail, requests to execute data manipulation operations are issued by clients, usually using SQL. A server executes the request and returns results to clients. When a request

to commit the transaction is issued, the server ensures that ACID properties, usually by waiting that updates have been written to stable storage.

Many database replication protocols have been proposed and implemented and we do not attempt to present a full listing of all existing replication approaches. We also avoid to reproduce a taxonomy of existing protocols as found in [WPS+00]. Instead, we adopt the criteria of relevance to the GORDA project and make only a coarse grained classification as follows.

First, in Chapter 2, we present *lazy* replication replication protocols, also known as asynchronous or deferred update protocols. Although these protocols provide only a limited form of fault-tolerance, they are available for the vast majority of databases and account for most installations of replicated databases. The sheer number of users of such protocols makes them relevant for GORDA. We briefly describe implementations for mainstream database management systems and survey existing proposals of lazy replication algorithms.

On the other hand, Chapter 3 presents *eager* replication in databases, also known as synchronous replication. These are present in a number of commercial offers and range from conventional distributed database protocols to the novel and flexible RAIDb approach. Although not as widely used as lazy protocols, these are usable in high-availability scenarios and are therefore relevant for GORDA.

A particularly interesting approach to eager database replication deserves a special treatment. These are group-based replication protocols which take advantage of group communication and are discussed in Chapter 4. A large part of the complexity of group-based replication protocols is off-loaded to group communication. In addition, supporting database replication has shown to pose new challenges to group communication protocols. In Chapter 5 we present an overview of group communication, addressing the services provided and existing implementations.

Chapter 6 concludes this document by identifying common concerns and solutions in existing protocols. We also highlight how replication interfaces with database management systems, end-users and management tools.

Chapter 2

Lazy Protocols

Lazy replication is a standard feature of all mainstream database management systems [Mic05, Urb03, Syb03, Dat 7, Gro]. It works by executing and committing each transaction at a single replica without synchronization. Other replicas are updated later by capturing, distributing and applying updates. Impact in user visible performance, specially on transaction latency is therefore reduced. Lazy replication is not suitable for fault-tolerance by fail-over while ensuring strong consistency because updates can be lost after the failure of a replica.

Implementations differ on which replicas can process and publish updates to different data objects, how updates are captured, distributed, filtered and applied, and finally on how the whole system is managed.

In the following the general structure of lazy replication protocols and how it is realized in existing database management systems is presented. A brief survey of lazy replication algorithms for databases concludes this chapter.

2.1 Model

Lazy replication protocols can be divided in three phases: (*i*) capture, (*ii*) distribution, and (*iii*) apply.

2.1.1 Capture

The capture phase involves obtaining updates performed to replicated objects in a format suitable for publication. Depending on the implementation, capture is triggered by events such as the number of changes in a published object, the number of committed transactions in a publisher, and time intervals.

This can be easily implemented without modifications to the database management system by setting up triggers on update operations, although with some impact on performance. The overhead can be reduced by using the transaction log. By retrieving information from stable parts of the log, there is no interference with running transactions.

Table 2.1 classifies implementations according to the following criteria:

Log reader It means that the capture process catches the updates by reading the database transaction log.

Trigger implementation It means that the capture phase catches the updates using triggers applied to the replicated object.

Capture full object It means that the capture phase make a copy of the entire object at a specific time.

DDL statements capture It denotes if it is possible to replicate modifications (DDL commands) applied to an object.

DBMS	Capture process				
	Type/Product	Log reader	Trigger implementation	Capture full object (i.e. relation)	DDL statements capture
MS SQL 2000	Snapshot	No	No	Yes	Yes
	Merge	No	Yes	No	No
	Transaction	Yes	No	No	No
Oracle	Stream	Yes	Opt.	No	Yes
	Advanced	No	Yes	No	Yes
Sybase	Replication Server	Yes	No	No	Yes
IBM DB2	DataPropagator	Yes	No	No	No
PostgreSQL	Slony-1	No	Yes	No	No
MySQL	Built-in	Yes	No	No	Yes

Table 2.1: Capture phase summary.

2.1.2 Distribution

The distribution phase propagates changes in published objects to relevant replicas. Complex distribution scenarios, involving multiple hops, filtering, and staging areas, are better addressed by implementations that decouple distribution from capture and apply. Namely:

- Distribution must deal with failures in replicas and in the network, possibly including long periods of disconnected operation. Failures must not cause lost or duplicated updates.
- When propagating updates between different database management systems it might be necessary to perform data conversions.
- Filtering is often required due to visibility and performance reasons, when coping with very large amounts of data and different access control policies.
- Filtering can also be required to enforce that updates to each replicate object are performed only at a designated primary replica, thus avoiding serializability conflicts.

Table 2.2 classifies implementations according to the following criteria:

Relay distribution If it is necessary, the database administrator may use more than one distribution mechanism to intermediate the propagation. This feature is interesting for a better use of the network resources.

Proprietary interface Some databases use proprietary interface replica. This feature may increase the performance, but may restrict the connection to databases of the same vendor.

Client interface In a replication environment, the connection between two distinct databases may, optionally, be done by using the common database drivers. This option may be useful for a heterogeneous environment but may introduce some overhead at the communication.

Integrated with Capture If the distribution mechanism is the same or part of the capture mechanism, we say that the distribution mechanism is tightly and for that reason the capture and the distribution occur at the same database.

Integrated with Apply The same logic is applied when the distribution mechanism is tightly coupled with the Apply phase.

Staging area for capture It means that exists a staging area between the capture process and the distribution process.

DBMS	Distribution process							
	Type/Product	Relay distrib.	Proprietary interface	Client interface	Integrated coupled with Capture process	Integrated coupled with Apply process	Staging area for capture	Staging area for apply
MS SQL 2000	Snapshot	No	Yes	Yes	No	No	Yes	No
	Merge	No	Yes	Yes	Yes	Yes	No	No
	Transaction	No	Yes	Yes	No	No	Yes	No
Oracle	Stream	Opt.	Yes	No	No	No	Opt.	Opt.
	Advanced	No	Yes	No	No	No	Yes	Yes
Sybase	Replication Server	Yes	No	Yes	No	Yes	Yes	No
IBM DB2	DataPropagator	No	No	Yes	No	Yes	Yes	No
PostgreSQL	Slony-1	Yes	No	Yes	No	No	Yes	Yes
MySQL	built-in	No	Yes	No	Yes	Yes	No	No

Table 2.2: Distribution process summary

2.1.3 Apply

The main concern when applying updates is that no serializability conflicts arise with previously committed transactions. This can be easily ensured by restricting modification of each data item to a designated master copy. Otherwise, an application specific procedure for reconciliation must be initiated, which is supported by sorting conflicting updates and triggering the necessary events.

Table 2.3 classifies implementations according to the following criteria:

Staging area for apply The same logic is used between the Distribution process and the Apply process.

DML statements To apply the changes at the subscriber, one of the options is to use the usual DML statements. This approach may increase the compatibility with heterogeneous database.

Procedures The changes may also be applied using specific procedures for distinct DML statements.

Auto create It allows to automatically create the object schema in order to start the replication.

Advanced conflict resolver It means that the replication tool may detect the conflict between the new data and the old data and resolve the problem with configurable and advanced features (i.e, update order). A higher priority assigned to the master database is not considered an advanced conflict resolver.

Transactional consistency The replication must preserve ACID [GMUW02] properties.

DBMS	Apply process					
	Type/Product	DML statements	Procedures	Auto create	Advanced conflict resolver	Transactional consistency
MS SQL 2000	Snapshot	No	No	Yes	No	Yes
	Merge	No	Yes	Yes	Opt.	Yes
	Transaction	Yes	No	No	No	Yes
Oracle	Stream	No	Yes	No	Opt.	Yes
	Advanced	No	Yes	Yes	Yes	Yes
Sybase	Replication Server	Yes	No	No	No	Yes
IBM DB2	DataPropagator	Yes	No	Yes	Opt.	Yes
PostgreSQL	Slony-1	Yes	Yes	No	No	Yes
MySQL	built-in	Yes	No	Yes	No	Yes

Table 2.3: Apply process summary

2.2 Management

Management is concerned with two different aspects: changes to database meta-information and changes to the replication configuration. Management can be restricted by requiring that operations are performed at a master replica and by forbidding concurrent access by other management operations or even by user transactions.

The main issue when reconfiguring the system is the initial synchronization of new replicas. In general, the state of the subscriber's object is compared with the publisher's object and the differences are applied to the subscriber. However, in order to avoid an unfeasible large amount of data being propagated between the replicas during the state transfer, the following procedure could be adopted. A previous taken snapshot from the publisher's object is off-line applied to the future subscriber's object before the subscription. Then, during the subscription just the changes that occurred after the snapshot being taken is applied.

Finally, it is important to notice that the snapshot could be periodically taken, for instance, and it could be also used in an lazy replication. However, it is not recommended because it could involve a huge amount of data.

Table 2.4 classifies implementations according to the following criteria:

Management in runtime It means that the data replication is carried without interruption when management activities are executed. Basically, this feature is translated into transparency to the clients.

Centralized management If there is just one management point through which the user does the management replication activities.

Distributed management data The replication management data may be distributed among the replicas without replication. In this case, we say that the management data is distributed. Otherwise, we say that it is centralized.

Additional DB resources It means that the database requires additional modifications to its architecture to support the replication.

Multi-master It means that the replication architecture allows update-everywhere and the execution is symmetric between replicas.

DBMS	Replication management					
	Type/Product	Management in runtime	Centralized management	Distributed management data	Additional DB resources	Multi-master
	Snapshot	Yes	No	Yes	Yes	No
MS SQL 2000	Merge	Yes	No	No	Yes	Yes
	Transaction	Yes	No	Yes	Yes	No
Oracle	Stream	Yes	No	Yes	No	Yes
	Advanced	No	Yes	No	Yes	Yes
Sybase	Replication Server	Yes	No	Yes	No	No
IBM DB2	DataPropagator	Yes	No	Yes	No	Yes
PostGreSQL	Slony-1	Yes	No	Yes	No	No
MySQL	built-in	Yes	Yes	No	No	Yes

Table 2.4: Management process summary

2.3 Implementations

2.3.1 MS SQL Server 2000

The MS SQL Server 2000 provides three replication solutions: (*i*) transactional replication, (*ii*) snapshot replication and (*iii*) merge replication.

The transactional replication is a single-master replication where the changes are later replicate per transaction. The snapshot replication is used during state transfers when a replica subscribes a published object. The merge replication allows multi-master replication with conflict detection mechanisms which are built according to the application semantic.

In what follows, we introduce the ideas behind the MS SQL Server 2000 replication architecture.

Snapshot Replication

In a snapshot replication, the entire published object is captured. This solution does not monitor the updates made against an object and it is normally used as a state transfer for the transactional and merge replication.

Roughly, the capture process is implemented by a "Snapshot Agent". Periodically, it copies the replicated object's schema[1] and data from the publisher to a snapshot folder for future use by a "Distribution Agent", which also acts as an apply process.

Transactional Replication

The transactional replication uses the log as source to capture incremental changes that were made to published objects.

The capture process is implemented by a "Log Reader Agent", which copies transactions marked for replication from the log to the "Distribution Agent". Basically, the capture process reads the transaction log and queues the committed transactions marked for replication. The capture process runs at the distribution's site and there is one process (i.e., log reader agent) for each database that has an object configured to be published. Then, the "Distribution Agent" reads the queued transactions and apply them to the subscribers.

Merge Replication

The merge replication allows the publisher and the subscribers to make updates while connected or disconnected. When both are connected, it merges the updates. Whenever a conflict arises, a user defined conflict resolution is started.

The capture process tracks the changes at the publisher and at the subscribers. It is implemented using triggers.

The distribution process is made by a "Merge Agent", which also acts as an apply process.

2.3.2 Oracle 10g

Oracle 10g offers three replication solutions: snapshot replication, stream replication and advanced replication [Urb03, Ora97].

Stream Replication

Oracle stream enables the propagation and management of data, transactions and events in a data stream either within a database, or from one database to another. The stream routes published information to subscribed destinations. Based on this feature, it is possible to use it for replication.

The Oracle stream uses the log as input to a capture process, which formats the changes into modifier events and enqueues them. It captures data manipulation and definition commands.

[1] The schema is the object's structure and it is used to rebuild it.

A distribution process reads the previous queue, which is located at the same database, and enqueues the events in a remote database. The queue from which the events are propagated is called the source queue, and the queue that receives the events is called the destination queue. There can be a one-to-many, many-to-one, or many-to-many relationship between source and destination queues.

At a destination database, an apply process consumes the events, applying them directly; or the apply process may dequeue the events and send them to an apply handler. Basically, the apply handler performs customized processing of the event and then applies it to the replicated object.

It is important to note that the management does not stop the replication and all the process are configurable.

Advanced Replication

When the replication is enabled in a database, it is created a set of support triggers and procedures at the new replica. Basically, for each replicated object (i.e., relation) it is created a trigger (see [Ora97] for further explanation about triggers and the special triggers create for replication.), which enqueues a specific remote procedure call according to the command executed. This queue is consumed by the Oracle implementation of the distribution process.

The distribution process push and pull the deferred calls, propagating the changes from sources to destination databases. Then, the apply process dequeues these information and updates the subscriber. Since it allows update-everywhere, a conflict detection mechanism compares the old value at the source object with the actual value of the destination. If a conflict is detected, it is called a conflict resolution procedure.

The management process performs operations such as the addition of replicas or modifications to replicated objects. Management is allowed only from a specific site know as "master definition site" and when the replication is in a suspended mode, which means that the environment does not accept transactions during this phase.

2.3.3 Sybase

Sybase Replication Server

The "Sybase Replication Server" is a separate product designed specifically for replication. "Replication Server" components are clearly separated from the activities of a source database. The Sybase "Replication Server" architecture, which does not use triggers or rules, does not burden the source database. One of the reasons organizations choose to replicate data is to offload the work that is performed on the primary server. Consequently a replication system that imposes a significant performance and administration burden on the source database would in part defeat the purpose for which it is introduced.

In the "Sybase Replication Server", the capture process runs as a thread ("Replication Agent") inside the database that reads the transaction log to detect changes to the objects. When it de-

tects a committed transaction, a modification to a table or a stored procedure that has been marked for replication, it passes it to the "Replication Server". During this propagation, it translates the logged information into data source independent commands understood by the "Replication Server".

Briefly, the distribution process is available as a Service into the "Replication Server". It propagates the changes and allows to use multiple "Replication Servers" to create routes from the source to the destination. For instance, if the publisher and the subscriber are on separate LANs, the distribution propagates the changes from the "Replication Server" connected with the publisher database to the "Replication Server" connected with the subscriber database.

The direct or indirect routes that replicated changes take are pre-configured by systems administrators. Routes allow administrators to make the most efficient use of corporate networks in accordance with the constraints of their networks and the requirements of their applications.

To apply the changes, the "Replication Server" connects to the subscriber and then delivers the replicated information to its destination. There is no special process running at the destination. Furthermore, the "Replication Server" does provide data translation vehicles called function strings and user-defined data types that the customer can use to change the data within a command or even change the actual command. This feature allows simple connections with non Sybase database systems.

The management tool ("Replication Management Server") allows creation of connections, routes, replication definitions, publications, and subscriptions. Furthermore, the configuration and status information on direct and indirect routes can be monitored from a central location with the Replication Server Manager.

2.3.4 IBM DB2

IBM DB2 DataPropagator

The DB2 DataPropagator is a replication product for relational data. It is useful to replicate changes between any DB2 relational database. As changes made to a publisher are not made immediately available to subscribers. However, it is possible to control how frequently the changes are applied to the subscriber by specifying time intervals, events, or both. For environments that have occasionally connected clients, it is possible to replicate information on demand. The DB2 DataPropagator consists of three main components: administration interfaces, change-capture mechanisms, and an apply program. In what follows, we attempt to describe the DB2 DataPropagator.

The capture process runs at the subscriber. It uses the log to read the changes and stores them in a local table. Each published table (i.e., object) has a corresponding "change data table" where the captured changes are stored temporally. When the changes are applied into the subscriber, they are automatically deleted.

An apply program reads the "change data table" and applies these changes to subscribers. It may run as a batch process or as a task that runs all the time at any server.

The management is made using control tables. The replication components use control tables to communicate with each other and to manage replication tasks (such as managing replication sources and targets, capturing changes, replicating changes, and tracking how many changes are replicated and how many remain to be done).

2.3.5 PostgreSQL

The Slony-1 is the “replication product” available for the PostgreSQL. It implements lazy replication with a "master to multiple slaves" replication.

The capture process is implemented using triggers that log the changes made against the published objects.

The changes are periodically distributed using a replication daemon that connects directly to the publisher, reads the logged changes and forwards it to the subscribers. It allows to connect several subscribers in cascade. The management process is not integrated or centralized. For that reason, the maintenance tasks must be done in each replica.

2.3.6 MySQL

MySQL includes a built-in lazy replication protocol that can be configured to propagate updates between replicas. Although the system provides master-slave replication, each slave can be configured to relay updates to other replicas, or even, to behave as master for local updates. There is however no conflict resolution and thus it is the responsibility of the users to avoid inconsistency. Schema updates are propagated and thus administration can be centralized. Replica management is eased by the possibility of bulk transfer of database state while the primary is on-line.

2.4 Algorithms

A number of research groups have worked database replication based on the lazy propagation of updates. We survey some of the most representative proposals [ABKW98, BKR+99, ACZ03, PA04, ATS+05, GPN05].

In [ABKW98] Anderson et al. present a solution targeted towards large scale replication. The authors assume a star-topology where one central site is used to handle global concurrency control. Two variations of the protocol are considered, one with optimistic and the other with pessimistic transaction execution. In general, the optimistic approach outperforms the pessimistic one. In that case, a transaction is first executed at the site receiving the request, subject to local concurrency control there. After local execution the read-set and write-set of the transaction is submitted to the central site for global validation. Using the read- and write-sets, this validator site maintains a replication graph, which essentially represents a compressed version of the conflict graph for currently running transactions.

If the replication graph remains acyclic after tentatively applying the changes required by the transaction being validated, the execution is successful. Thus, the write-set is multicast to

all replicating sites which can commit the updates as soon as they are received. The approach is regarded lazy since a positive reply to the client can be sent as soon as the execution is validated by the certifying site, i.e. some replicas may not yet have received the updates at the time of commit. Serializability is ensured by aborting transactions failing the validation procedure. The protocol is shown to provide good performance compared to a traditional, eager replication protocol based on two-phase locking, but we are not aware of any other studies comparing it to more modern replication protocols.

Breitbart et al. [BKR+99] present two lazy protocols that release the central certification and ordering but instead rely on a precise placement of the replicas.

This work assumes that all replicas implement strict two-phase locking (2PL) and that communication is through FIFO reliable multicast. A copy graph is defined as a directed graph in which a set of vertices corresponds to the set of sites and there is an edge between two vertices only if one of them has a primary copy of an item and the other one stores a secondary copy. A set of edges in the copy graph are called backedges if their deletion eliminates all the cycles in the copy graph.

A forest is constructed out of the copy graph and used to propagate replica updates along the edges of the forest. Transactions execute at a single site and when they commit, the transaction's updates are forwarded to the children of the site in the given tree. Updates are applied in the order in which they are received at the site. Since updates might need to be routed through a number of intermediate sites, the protocol might result in significant message overload in the network and unnecessary propagation delays. Thus, the second protocol proposed propagates updates along the edges of the copy graph itself instead of the constructed forest. To guarantee correct order of updates at commit time transactions are assigned a system wide unique timestamp.

Conflict-aware scheduling was introduced as a way to implement lazy read-one write-all replication that guarantees one-copy serializability [ACZ03]. The key to this approach is the extension of the scheduler to include conflict awareness: transactions are directed to the replicas in such a way that the number of conflicts is reduced. Moreover, instead of resolving conflicts by aborting conflicting transactions, the scheduler is also augmented with a sequence numbering scheme to provide strong consistency is introduced.

The authors consider cluster architecture for a dynamic content site, where a scheduler distributes incoming requests to a cluster of databases and delivers responses to the application servers. The tables accessed by the transaction are required to be known in advance. The scheduler forwards write operations to all the replicas and replies to the client as soon as the first reply from the replicas is received. Read operations are executed only at a single replica. Each incoming transaction is assigned a unique sequence number. Transactions lock requests are sent to all the replicas and executed in order of their assigned sequence numbers, thus forcing all conflicting operations to execute in a total order at all database servers. A single read transaction holds locks only on a replica it executes. Some database servers might have fallen behind with execution of updates; some others might keep conflicting locks acquired by the transaction. For read operations, other than reads in a single query transaction, the scheduler

first determines the set of database replicas where the locks for its enclosing transaction have been granted and where all previous operations have executed already. It then selects the least loaded replica from this set as the replica to execute the particular read operation. The load over the replica is specified according to shortest execution length first load balancing algorithm.

The conflict-aware scheduler approach maintains one-copy serializability by assigning a unique sequence number to each incoming transaction. Lock requests are sent to all replicas and executed in assigned sequence order, thus forcing conflicting transactions execute in the same total order at all replicas and enforcing strong consistency. The scheduler itself can be replicated for availability and fault tolerance reasons. Each scheduler keeps a persistent log of all write queries executed together with their sequence numbers.

Ganymed [PA04] is a recent middleware-based replication approach based on the following two key aspects: the first is the separation between update and read-only transactions: update transactions are handled by a master replica and lazily propagated to the slaves, whereas queries are processed by any replica; the second is the use of snapshot isolation concurrency control. All updates on the master replica result in write sets that must be applied on the slave replicas. The use of snapshot isolation allows to install the write sets on the slave replicas without having conflicts with the read-only transactions executing on the slaves.

The main component of the Ganymed system is a lightweight scheduler that implements the replication protocol and balances transactional load over a set of database replicas. Update transactions are executed at the master replica; read-only transactions are scheduled to the slaves according to least pending request first balancing algorithm. The slave replicas must implement snapshot isolation concurrency control. All update transactions are directly forwarded to the master replica without any delay. The scheduler keeps track of the order in which update transactions are committed at the master and ensures the same commit order at the slave replicas. Read-only transactions can be processed by any slave replica. If the latest database version is not yet available at the chosen slave node, the execution of the transaction is delayed. Furthermore client application can choose to execute read-only transaction at the master replica or set a staleness threshold.

To ensure one copy snapshot isolation, the scheduler must make sure that write sets of update transactions get applied on all replicas in the same order. The distribution of write sets is handled by a FIFO update queue for every replica.

Conventional lazy replication techniques do not necessarily provide up-to-date data; recent work in [ATS+05, GPN05] addresses the issue of data freshness. The main idea of the proposed protocols is to exploit distributed versioning together with freshness locking to guarantee efficient replica maintenance that provides consistent execution of read-only transactions.

Both works assume a replicated database that contains objects which are both distributed and replicated among the server nodes. Each object has a primary site; updates of an object can occur only at its primary site, thus write operations on the same object are ordered according to the order given at the primary site. All database servers are partitioned into read-only sites and update sites. Read-only transactions can run at any read-only site. Read operations of the same transaction can run at different read-only sites. Local changes are propagated to secondary

copies by propagation transactions. Refresh transactions bring the secondary copies at read-only sites to the freshness level specified by a read-only transaction.

Read-only transactions place freshness-locks on the objects at the read-only sites to ensure that concurrent updates do not overwrite the required versions by ongoing read-only transaction. Freshness locks keep the objects accessed at a certain freshness level during the execution of the transaction. If a read-only transaction is scheduled to the site that does not yet fulfill the freshness requirements, a refresh transaction updates the objects at that site.

Regardless of the number of read-only sites, read-only transactions always see a consistent database state.

Chapter 3

Eager Protocols

Eager replication protocols provide strong consistency and fault-tolerance by ensuring that updates are stable at multiple replicas before replying to clients. This can however have a noticeable impact in user visible performance. Also in contrast with lazy protocols, approaches to eager replication differ in fundamental issues and provide very different trade offs between performance and flexibility.

3.1 Volume Replication Protocols

Replication of disk volumes performed at the block I/O level is a straightforward and general purpose approach to replication: By intercepting each block written by the application designated volumes and shipping it over to network, a remote copy is maintained ready for fail-over. Reads are performed on the local copy. The replication process is thus completely transparent to the application.

As with local mirroring, waiting for confirmation that both copies have been written before notifying the application of completion ensures consistency upon failure. This is specially relevant for database management systems which rely on write ordering to be able to recover by replaying logs.

The downside of the approach is that remote updates are performed with a block granularity, which depending on the application might represent a large network overhead. The approach is also restricted to fail-over, as the backup copy cannot usually be used even for read-only access, due to lack of cache synchronization at the application level.

Examples of the approach can be found in the Veritas Volume Replicator [Cor02], available for a number of different operating systems, and in the open source DRBD for Linux [Rei02]. Performance evaluation of both [TKS, e02] shows that with sufficient network resources available and with a typical OLTP load, the approach results in overhead of within 5% of a single local volume and is thus viable.

3.2 Distributed Database Protocols

In a distributed database, a transaction operates on items stored at different sites by shipping operations to the appropriate destinations, where local concurrency control and recovery mechanisms are used. Atomicity is ensured by running a two phase commit protocol to agree on the outcome of the transaction. In addition, a global deadlock detection protocol is required to ensure liveness. In this context, a replicated database can be obtained by storing multiple copies of items at different sites and scheduling mirrored operations to them.

A major issue is the strategy used to update replicas while ensuring that read operations return current values and a number of proposals exist [TGGL82, GSC+83, BG84, BHG87, Tho79, Gif79, DS83, BL82, AA92, ES83]. Recent results [JPPMAK03] indicate that under realistic conditions the best performance is offered by protocols that eagerly update replicas.

The simplest of these is the Read One/Write All (ROWA) protocol [TGGL82], allows read operations to be executed by reading a single replica and ensuring that write operations are done by every replica. In such environment, the failure of a single replica is sufficient to disallow write operations in the system.

A practical alternative is the Read One/Write All Available (ROWAA) protocol [GSC+83], in which the system continues to operate as long as there are replicas that do not fail. Each replica is in one of two possible states online or off-line. A replica is online when its state is up to date and it is ready to execute transactions. Whenever it fails, it is excluded by the others from the set of available replicas, and its state changes to off-line. To became online again it has to go through an incarnation process, which will bring it up to date. The available replicas management protocol is a separate protocol from the replication protocol and only interacts with the replication protocol in the definition of the currently available replicas.

Although the resulting approach allows update operations to be issued at multiple sites while ensuring one copy serilizability [BHG87], a well known result [GHOS96] points out that it does not scale as the workload, number of nodes, or network latency increase due to the resulting high probability of deadlocks.

3.3 RAIDb Protocols

A Redundant Array of Inexpensive Databases (RAIDb) aims at providing better performance and fault tolerance than a single database, at low cost, by combining multiple database instances into an array of databases. Like RAID to disks, different RAIDb levels provide various cost/performance/fault tolerance tradeoffs. RAIDb-0 features full partitioning, RAIDb-1 offers full replication and RAIDb-2 introduces an intermediate solution called partial replication, in which the user can define the degree of replication of each database table.

RAIDb primarily targets low-cost commodity hardware and software such as clusters of workstations and open source databases. On such platforms, RAIDb will be mostly implemented as a software solution like the C-JDBC middleware prototype [CMZ04]. However, like for RAID systems, hardware solutions could be provided to enhance RAIDb performance while

still being cost effective.

RAIDb hides the distribution complexity and provide the database clients with the view of a single database like in a centralized architecture. As for RAID, a controller sits in front of the underlying resources. The clients send their requests directly to the RAIDb controller that distributes them among the set of RDBMS backends. The RAIDb controller gives the illusion of a single RDBMS to the clients. RAIDb controllers may provide various additional services such as load balancing, dynamic back-end addition and removal, caching, connection pooling, monitoring, debugging, logging or security management services.

3.3.1 Basic RAIDb levels

RAIDb-0: full partitioning

RAIDb level 0 is similar to striping provided by RAID-0. It consists in partitioning the database tables among the nodes. RAIDb-0 allows large databases to be distributed, which could be a solution if no node has enough storage capacity to store the whole database. Also, each database engine processes a smaller working set and can possibly have better cache usage, since the requests are always hitting a reduced number of tables. As RAID-0, RAIDb-0 gives the best storage efficiency since no information is duplicated.

RAIDb-0 requires the RAIDb controller to know which tables are available on each node in order to direct the requests to the right node. This knowledge can be configured statically in configuration files or build dynamically by fetching the schema from each database.

Like for RAID systems, the Mean Time Between Failures (MTBF) of the array is equal to the MTBF of an individual database backend, divided by the number of backends in the array. Because of this, the MTBF of a RAIDb-0 system is too low for mission-critical systems.

RAIDb-1: full replication

RAIDb level 1 is similar to disk mirroring in RAID-1: Databases are fully replicated. RAIDb-1 requires each backend node to have enough storage capacity to hold all database data. RAIDb-1 needs at least 2 database backends, but there is (theoretically) no limit to the number of RDBMS backends.

The performance scalability will be limited by the capacity of the RAIDb controller to efficiently broadcast the updates to all backends. In case of a large number of backend databases, a hierarchical structure like those discussed in section 4 would give better scalability.

Unlike RAIDb-0, the RAIDb-1 controller does not need to know the database schema, since all nodes are capable of treating any request. However, if the RAIDb controller provides a cache, it will need the database schema to maintain the cache coherence.

RAIDb-1 provides speedup for read queries because they can be balanced over the backends. Write queries are performed in parallel by all nodes, therefore they execute at the same speed as the one of a single node. However, RAIDb-1 provides good fault tolerance, since it can continue to operate with a single backend node.

RAIDb-1ec

To ensure further data integrity, we define the *RAIDb-1ec* level that adds error checking to RAIDb-1. Error checking aims at detecting Byzantine failures that may occur in highly stressed clusters of PCs. RAIDb-1ec detects and tolerates failures as long as a majority of nodes does not fail. RAIDb-1 requires at least 3 nodes to operate. A read request is always sent to a majority of nodes and the replies are compared. If a consensus is reached, the reply is sent to the client. Otherwise the request is sent to all nodes to reach a quorum. If a quorum cannot be reached, an error is returned to the client.

The RAIDb controller is responsible for choosing a set of nodes for each request. Note that the algorithm can be user defined or tuned if the controller supports it. The number of nodes always ranges from the majority (half of the nodes plus 1) to all nodes. If all nodes are chosen, it results in the most secure configuration but the performance will be the one of the slowest backend. This setting is a tradeoff between performance and data integrity.

RAIDb-2: partial replication

RAIDb level 2 features partial replication which is an intermediate solution between RAIDb-0 and RAIDb-1. Unlike RAIDb-1, RAIDb-2 does not require any single node to host a full copy of the database. This is essential when the full database is too large to be hosted on a node's disks. Each database table must be replicated at least once to survive a single node failure. RAIDb-2 uses at least 3 database backends (2 nodes would be a RAIDb-1 solution). Like for RAIDb-0, RAIDb-2 requires the RAIDb controller to be aware of the underlying database schemas to route the request to the appropriate set of nodes.

Typically, RAIDb-2 is used in a configuration where no or few nodes host a full copy of the database and a set of nodes host partitions of the database to offload the full databases. RAIDb-2 can be useful with heterogeneous databases. An existing enterprise database using a commercial RDBMS could be too expensive to fully duplicate both in term of storage and additional licenses cost. Therefore, a RAIDb-2 configuration can add a number of smaller open-source RDBMS hosting smaller partitions of the database to offload the full database and offer better fault tolerance.

As RAID-5, RAIDb-2 is a good tradeoff between cost, performance and data protection.

RAIDb-2ec

Like for RAIDb-1ec, RAIDb-2ec adds error checking to RAIDb-2. Three copies of each table are needed in order to achieve a quorum. RAIDb-2ec requires at least 4 RDBMS backends to operate. The choice of the nodes that will perform a read request is more complex than in RAIDb-1ec due to the data partitioning. However, nodes hosting a partition of the database may perform the request faster than nodes hosting the whole database. Therefore RAIDb-2ec might perform better than RAIDb-1ec.

3.3.2 Composing RAIDb levels

Vertical scalability

It is possible to compose several RAIDb levels to build large-scale configurations. As a RAIDb controller may scale only to a limited number of backend databases, it is possible to cascade RAIDb controller to support a larger number of RDBMS.

There is potentially no limit to the depth of RAIDb compositions. It can also make sense to cascade several RAIDb controllers using the same RAIDb levels. For example, a RAIDb-1-1 solution could be envisioned with a large number of mirrored databases. The tree architecture offered by RAIDb composition offers a more scalable solution for large database clusters especially if the RAIDb controller has no network support to broadcast the writes.

As each RAIDb controller can provide its own cache, a RAIDb composition can help specialize the caches and improve the hit rate.

Horizontal scalability

The RAIDb controller can quickly become a single point of failure. It is possible to have two or more controllers that synchronize the incoming requests to agree on a common serializable order. In the case of shared backends, only one controller sends the request to the backend and notifies the other controllers upon completion. Backends do not necessarily have to be shared between controllers but nodes that are attached to a single controller will no longer be accessible if the controller fails.

Chapter 4

Group-Based Protocols

Replication is an area of interest to both distributed systems and databases: in database systems replication is done mainly for performance reasons while in distributed systems - mainly for fault tolerance purposes. The synergy between these two disciplines offers an opportunity for emergence of database replication protocols based on group communication primitives.

These protocols take advantage of the specific properties of the group communication primitives, such as ordering and atomicity of messages, to eliminate the possibility of deadlocks, reduce message overhead and increase performance.

4.1 Active replication

In the active replication protocol, client requests are sent to every server, get ordered, executed by the servers and replies are sent to the client after execution. The principle governing active replication, also called state machine approach [Sch93], is that if all replicas start in the same initial state and execute the same requests in the same order, then each of them will do the same work and produce the same output. Read-only operations can be executed by any replica independently, thus providing parallelism.

This principle implies that requests are processed in the same deterministic way which is hard to achieve with complex interactive transactions and with concurrent execution. Group communication is used to totally order messages. Unlike typical two-phase commit mechanism for synchronous replication, this approach does not require per-action-basis acknowledgments from remote databases, thus keeping the communication costs to a minimum.

Active replication of databases has been implemented in EMIC Application Cluster (EAC). With its multi-master architecture, the client connections are intercepted distributed to any active server. In addition, Emic's Cluster Manager incorporates fault management technology, which allows automated, fast fail-over where remaining cluster nodes take over the load of the failed node(s). The Cluster Manager also allows a shutdown of any node for maintenance purposes, without service interruption.

4.2 Partitionable networks

By taking advantage of the *Spread Group Communication Toolkit* that provides atomic broadcast and deals with network partitions, Amir et al. [AT02, ADA+02, ADA+03] propose an active replication architecture that tolerates network partitions. Hereafter, we briefly describe the replication model, the replication architecture [AT02]. Interesting performance measurements are reported in [ADA+03] for different settings, varying the number of replicas, the number of clients as well as the network configuration.

4.2.1 Replication Model

The replication implements a symmetric distributed algorithm to determine the order of actions to be applied to the database. Each server builds its own knowledge about the order of actions in the system using the following algorithm: The knowledge about each action is indicated by its *color*, i.e., a *red action* is an action for which the global order is unknown, a *green action* is an action for which the server has determined the global order, a *white action* is an action for which the server knows that all servers marked it green.

In the presence of network partitions, a single component of the server group is elected *primary component*, other servers will belong to *non-primary component*. Actions (queries and writes) are handled depending on the server state, i.e., whether the server belongs to the primary component or not and whether there is a change in the membership (transiting from primary component to non primary component and vice versa). Only servers in the primary component are allowed to apply actions to the database. Indeed, actions delivered to the primary component are marked green, while actions delivered to a the non-primary component are marked red. A server in the *prim state* or the *non-prim state*, switches to the *exchange state*, upon delivery of a view change notification (i.e., connectivity change, join/leave). Servers in the new view exchange information allowing them to define the set of actions that are known by some of them but not by all. If the new view can form the next primary component the server will move to the *construct state*, otherwise it will return to the *non-prim state*. In the *construct state*, servers attempt to install the next primary component, by sending *create primary component (CPC) messages*. When a server receives all CPC messages from the members of the current component, it transforms all red actions into green, apply them to the database, and switch to the *prim state*. Using this model, we don't know which messages were received by which servers just before a network partition occurs or server crashes, more explicitly a server within the primary component cannot know if the last actions it delivered before the view change, were delivered to all the members of the primary component or not. The messages in question can not so be marked green. The model is extended to cope with view changes and consistent action knowledge. Thus, a view change message is split into two messages and defines so two configurations; a *regular configuration* and a *transitional configuration*. The latter contains members of the next regular configuration coming directly from the same regular configuration. The *prim state* is replaced by the *regular state* and the *transitional state*. Consequently,

actions delivered during the regular configuration are marked green, while actions delivered during the transitional configuration are marked yellow. A *yellow action* becomes green, if the server becomes part of the primary component.

4.2.2 Replication Architecture

The architecture is structured into two layers: a *Replication Server* and *Spread Toolkit* as Group Communication Toolkit.

- Replication Server: the prototype is an extension of PostgreSQL, that consists of three components, (i) a *Postgres Interceptor* (ii) a *Semantics Optimizer* and (iii) a *Replication Engine*.
  - Postgres Interceptor: it interfaces the Replication Engine with the DBMS client-server protocol. The interceptor listens for client connections, and once a client message is received, it is passed to the Semantics Optimizer.
  - Semantics Optimizer: it consists of (i) a basic SQL parser that identifies queries, which are executed locally without any replication overhead; and (ii) a primary component checker.
  - Replication Engine: It includes the recovery module dealing with both node and network failures, and the synchronizer algorithm [AT02] handling delivered actions.
- Spread Group Communication Toolkit: The Spread Toolkit provide basic services for reliable, FIFO, total order, safe delivery as well as it overcomes message omission faults and notifies the replication server of changes in the membership of the currently connected servers.

4.3 Optimistic Execution

In practice eager, update everywhere replication protocols often suffer from high deadlock rates, message overhead and poor response time. Several works have proposed implementing eager replication on top of group communication middleware. Among them Kemme and Alonso [KA98, KA00] introduced a suite of eager, update everywhere replication protocols, which use group communication primitives to ensure ordering and atomicity of transactions. They prove that such replication is perfectly feasible in many environments by implementing their approach in Postgres-R, an extension of PostgreSQL.

4.3.1 Model

Kemme and Alonso suggested to bundle all transaction writes to a single writeset message. Furthermore they use shadow copies to apply updates: write operations are executed on a private copy in order to check consistency constraints. These changes are propagated to other sites at transaction commit. Read operations are executed only at one site and no information is exchanged with other replicas. Moreover group communication primitives providing total

order semantics are used to multicast the write sets and to determine the serialization order of the transactions.

The proposed replication protocols execute a transaction in four major phases:

1. Local Read Phase. All read operations are performed locally. Write operations, after acquiring the locks, are executed on shadow copies.
2. Send Phase. Read-only transactions are committed, else all transaction's writes are collected to the write set and multicast to all sites including the sending one (same delivery order at all sites).
3. Lock Phase. Upon transaction's write set delivery, all locks for it are requested in atomic step:
   (a) For each write operation:
      - Conflict test is performed: if another local transaction has a lock on the same data item and that transaction is still in read or send phase, it is aborted. If that transaction is in send phase, then decision message abort is multicasted (no order requirements).
      - If there is no lock on the data item, the lock is granted to the transaction. Otherwise the lock request is queued after all locks from transactions, that are beyond there lock phase.
   (b) If transaction is local, the decision message commit is multicasted (no order requirements).
4. Write Phase. If the required lock is granted, transaction updates are applied. A local transaction can commit and release all the locks once all updates have been applied to the database. A remote transaction must wait until the decision message arrives.

4.3.2 Consistency

Kemme and Alonso take advantage of the different levels of isolation provided by databases to relax consistency guarantees of the protocols and to capture varying requirements of applications. In [KA98] and [KA00] protocols with one copy serializability (achieved implementing strict two phase locking), cursor stability (implemented with short read locks) and snapshot isolation are presented. The same approach is applied in the case of the atomicity requirements of the broadcast primitives, which can be relaxed in order to minimize message overhead. Therefore the authors introduce also non-blocking, blocking and reconciliation based versions of the protocols.

4.4 Scalable replication

The two major drawbacks of existing replication protocols based on group communication primitives are the amount of redundant work performed at all sites and the high abort rates

created when consistency is enforced. Two protocols have been proposed [PMJPKA00, KPAS99, KPA+03, JPPMKA02, PMJPKA05] to avoid these problems, first, the redundant work is minimized if transactions are executed only at one site and the other sites only install final updates, and second, the reordering technique is used to reduce the abort rate of transactions.

4.4.1 Model and proposed algorithms

Communication between replicas uses optimistic total order broadcast, defined by three primitives: to-broadcast (the message is broadcasted to all the sites of the system), opt-deliver (message is delivered without order guarantees), to-deliver (message is delivered to the application in a total order).

Concurrency control is based on conflict classes, each of them representing a partition of the data. Each conflict class has a master site. Read-only transactions can be executed at any site using a snapshot of the data, while update transactions are broadcast to all sites, however they are only executed at the master site of their conflict class.

The Middle-R algorithm

The algorithm is described [JPPMKA02] according to different phases in a transaction execution: it is opt-delivered, to-delivered, completes execution and commits. Each of the phases is done in atomic step. A transaction can only commit when it has been executed and to-delivered, furthermore transaction can be executed only at its master site. Each transaction has two state variables: execution state and delivery state. The execution state can be active or executed, while the delivery state can be pending or committed. When a transaction is opt-delivered its state is set to active or pending, therefore whenever a transaction is local and the first one in any of its queues, its operations are submitted for execution.

The NODO-Reordering Algorithm

This algorithm [PMJPKA00] is similar to the Middle-R. The commit message contains however the identifier of the serializer transaction and follow a FIFO order. The rescheduling together with the FIFO ordering ensure that remote transactions will commit at all sites in the same order in which they did at the local site.

4.4.2 Consistency

In [PMJPKA00] it is proved that both algorithms ensures one copy serializable transaction histories, showing that all produced histories are conflict equivalent.

4.4.3 Failures

In the proposed system, each site acts as a primary for the conflict classes it owns and as a backup for all other conflict classes. In the case of site failures, the available sites have to select

a new master for the conflict classes of the failed site. The new master is also responsible for all pending transactions, for which the failed node was the owner.

Transaction messages must be uniformly multicast because only then it is ensured that the master site will commit transaction only when all the sites receive it. In the NODO algorithm commit messages do not need to be uniform, while in the REORDERING algorithm it must be uniform and the master may not commit a transaction before the commit message is delivered in order to keep the serialization order the same in all replicas.

4.5 Database State Machine approach

The resilience and performance of eager update-everywhere replication can be further improved by independently certifying transactions at each replica. The Database State Machine (DBSM) by Pedone et al. [PGS02, Ped99] is a replication technique, which copes with replication in the cluster of database sites connected by a standard communication network. This approach is based on deferred update replication, implemented as a state machine. All database sites (replicas) receive and process the same sequence of request, assuming deterministic behavior of replicas. Databases local consistency (one copy serializability) is guarantied by strict two phase locking concurrency control mechanism and the global consistency - by atomic broadcast primitives and deterministic certification test.

4.5.1 Model

Read only transactions are processed locally at database site and update transactions do not require any synchronization with other database sites until the commit is requested. On the commit request the transaction's updates, that is its read-sets and write sets, are atomically broadcasted to all replicas for certification, which will result in transaction commit or abort. Each database site behaves as a state machine, where the agreement and order properties required by the state machine are ensured by the atomic broadcast primitives. All the database sites must eventually reach the same state and to accomplish this requirement, delivered transactions are processed with certain care - all of them have to pass deterministic certification test. Every transaction execution is handled by the Transaction Manager, the Lock Manager and the Data Manager. The certification test on incoming transactions, delivered by atomic broadcast module, is carried out by the Certifier. During the certification, the Certifier can ask information about already committed transactions to the Data Manager. If a transaction passes certification test, its write operations are forwarded to the Lock Manager, which is responsible for granting the write locks. Once the locks are received, the updates can be performed. To make sure that each replica converges to the same state, each certifier has to

1. reach the same decision when certifying transactions

2. guarantee that conflicting transactions are applied to the database in the same order

The first constraint is achieved by providing the Certifier with the same set of transactions and using deterministic certification test. The second constraint is ensured if conflicting transactions grant their locks in the same order as they are delivered.

4.5.2 Consistency

Using the multi-version formalism the author has proved, that local strict two phase locking, deterministic certification test and atomic broadcast primitives are sufficient to achieve one copy serializability in the database system.

4.6 Pronto protocol

The Pronto protocol presented in [PF00] by Pedone and Frølund was among the first to consider building database replication without requiring modifications to the database engine, allowing it to be deployed in a heterogeneous environment. The protocol is based on the primary backup replication model and atomic broadcast primitives.

4.6.1 Model

The main idea behind the protocol is a hybrid approach, that has elements of both active replication and primary-backup. The authors deal with database non-determinism by having a single database to execute transactions in non-deterministic manner. But instead of sending only resulting state to backups, they send the transaction itself together with ordering information, that allows the backups to make the same non-deterministic choices as the primary. Like active replication, every database processes all transactions, in contrast to it, the backups process transactions after the primary, what allows the primary to make non-deterministic choices and export them to the backups. By shipping transactions instead of their logs, heterogeneous databases with different log formats are supported.

4.6.2 Consistency

In the Pronto protocol one copy serializability is achieved using strict two phase locking and atomic broadcast primitives.

4.6.3 Failures

The protocol proposed has a mechanism to handle replica failures. Transactions execution evolves as a sequence of epochs. During an epoch there can only exist one primary, which is deterministically computed from the epoch number. When a backup replica suspects the primary to have crashed, it broadcast message to all database sites to change the current epoch, which will result in another database server as the primary. To prevent database inconsistencies, a transaction passes a validation test before committing, which ensures, that a transaction

is only committed if the epoch when the database site delivers the transaction and the epoch when the transaction was executed are the same.

4.7 Partial database replication

Most of the previous works on database replication using group communication [ADA+02, AT02, PMJPKA00, KA98, KA00, Ped99] concentrates on full replication strategies. Alonso in [Alo97] discusses future trends for partial database replication based on atomic broadcast, stating that solutions for full replicated scenarios may not be solutions for partially replicated ones. He proves, that *order preserving serializability* is a sufficient condition to guarantee overall correctness in a partial replication architecture based on the execution of replicated transactions according to the total order established by a group communication protocol.

In [SPOM01] as part of ESCADA project Sousa et al. investigates the use of partial replication in the Database State Machine approach. It builds on the order and atomicity properties of group communication primitives to achieve strong consistency and proposes two new abstractions: Resilient Atomic Commit and Fast Atomic Broadcast. Even with atomic broadcast, partial replication requires a termination protocol such as atomic commit to ensure transaction atomicity. With Resilient Atomic Commit, the termination protocol allows the commit of a transaction despite the failure of some of the participants. Preliminary performance studies suggest that the additional cost of supporting partial replication can be mitigated through the use of Fast Atomic Broadcast, which exposes preliminary message delivery orders to the application before providing the application with a final definitive order.

4.8 Online recovery

While the previous protocols are able to mask site failures effectively, directly or through the virtual syncrony abstraction provided by the group communication substrate, they do not address the addition of new replicas or the recovery of failed sites. The challenge of the online recovery of databases in a replicated system is how to update the state of joining participants efficiently and with minimal disruptions to the normal transaction processing.

In [KBB01] Kemme, Bartoli and Babaoğlu address issues on how failed database sites can rejoin the system after recovery, how partitions can merge after repairs or how new sites can be added to a running system. The authors use view change events for the coordination and synchronization and propose several data transfer protocols, which depict the main alternatives for a database supported data transfer and the most important issues to consider: determining the data that must be transferred, exploiting information available within the database (e.g. log, version numbers), maintaining additional information to speed up recovery and allowing high concurrency.

Holliday [Hol01] also addresses the recovery of group-based replicated databases. The contribution of this work focus on determining the required amount of data to be transferred.

It does however overlook the overall system performance by proposing blocking protocols that suspend transaction processing while a replica is recovering.

In [JPPMA02], Jimenez-Peris et al. extend the work in [KBB01] so that recovery can be performed in parallel by partitioning the database among a set of donors. The partition scheme further allows a recovering site to start processing transactions as soon as they are confined to the already recovered partitions. The protocol still minimizes the time the system is unavailable to the view change periods.

4.9 Performance evaluation

Several independent research efforts have shown the performance of group-based database replication. Holliday et al. [HAA99] address the issue of implementing a fully replicated database with update-anywhere. The authors use simulation to evaluate a set of four abstract replication protocols based on atomic broadcast:

- The first protocol termed *broadcast all*, requires that a replica broadcasts all the operations and even the read' ones. Lock requests can be delayed, implying transaction deadlocks, which replicas must resolve in the same way.

- The second protocol termed *broadcast writes*, broadcasts only write operations and performs a broadcast per update operation. After obtaining read locks, read operations are executed locally for better response times. The local transaction processing is based on usual locks granting, detection and resolution of deadlocks between conflicting transactions, and finally the commit is broadcast.

- The third protocol termed *delayed broadcast writes*, obtains all write locks for a transaction atomically by deferring update operations. The update operations are broadcast to replicas, that obtain the locks grants atomically and execute the transaction. Once, all replicas execute the updates, the commit is broadcast to them.

- The fourth protocol termed *single broadcast transactions* uses a single broadcast per transaction, and combines versioning and locks. Indeed, after deferring write operations, the site broadcasts the set of reads with their version numbers and the set of writes of the transaction. Each replica tests if the version numbers are obsolete or not, if not tries to obtain atomically all write locks and resolves conflicting transactions.

Results show that all protocols perform and scale up surprisingly well in face of traditional distributed database protocols. The *broadcast writes* protocol requires little or no changes to the database and is therefore easier to implement and deploy. The approach underlies the EMIC Application cluster and the partitionable protocol of Amir et al. The *single broadcast transactions protocol* is the one that allows for better performance by avoiding duplicate execution and blocking. This protocol abstracts the Database State Machine approach.

An implementation of the Database State Machine is evaluated in detail in the ESCADA project [SPS+04] with a model of a replicated database server that combines simulation of the environment with early real implementations of key components. In particular, a real implementation of the certification and communication protocols is used, as these are the components responsible for the database replication based on the DBSM, and both the database engine and the network are simulated. This allows the assessment of the validity of the design and implementation decisions by experimenting with different configuration parameters. In detail, this includes:

- A centralized simulation runtime based on the standard Scalable Simulation Framework (SSF) [rac04], which allows the controlled execution of real implementations. By using the SSF, the existing SSFNet [CNO99] network simulator is reused as a key component of a realistic environment.

- A transaction processing model and a traffic generator based on the industry standard benchmark TPC-C [(TP01]. This provides a realistic load to prototype implementations of replication components.

- A configuration of the simulated components that closely matches a real system, namely, obtained by instrumenting and profiling PostgreSQL [pos]. This also validates the model.

- Finally, we apply the model to

The paper compares a centralized database with a replicated one, with various replication degrees and fault scenarios. The results confirm the usefulness of the approach, by illustrating the scalability to 3 and 6 sites with close to linear scalability. The results obtained with fault-injection are also able to pinpoint the requirements on the implementation of the group communication protocol and its impact in the quality of service provided to end-users of the replicated database.

Chapter 5

Group Communication

Group communication toolkits support message passing within groups of processes by offering membership management and reliable multicast services. Examples of group communication toolkits are Isis and Horus [BvR94], Ensemble [Hay98], xAMp [RV92], Transis [ADKM92], Newtop [EMS95], Phoenix [Mal96], Spread [ADS00] and Appia [MPR01].

Membership management keeps track of which processes are operational and mutually reachable, taking into account both voluntary requests to join and leave the group as well as process failures and network partitions. By ensuring that a common membership is observed by all participants, many distributed algorithms are simplified. Reliable multicast can informally be described as ensuring delivery of all messages to all destination processes that do not fail. Group membership serves as an implicit destination set for reliable multicast. Several definitions are possible and differ subtly in their guarantees in the presence of process failures. Several message ordering criteria in addition to reliable multicast can be provided by group communication. Message ordering simplifies application programming by ensuring that each message is handled in a predictable context resulting from previous messages.

5.1 Overview

5.1.1 Group Communication and other services

One way of building fault-tolerant protocols consists in using a modular approach based on a Generic Consensus Service [GS01a]. Fault-tolerant agreement protocols are built using a client-server interaction, where the clients are the processes that must solve the agreement problem and the servers implement the consensus service. This type of service provides a simple way to solve agreement problems. In the consensus problem, a collection of processes called acceptors cooperate to choose a value. Each acceptor runs on a different node. The basic safety requirement is that only a single value be chosen. To rule out trivial solutions, there is an additional requirement that the chosen value must be one proposed by a client. *Paxos* [Lam98] is an example of asynchronous consensus algorithms. It assumes some method of choosing a coordinator process, called the leader. Clients send proposed values to the leader. In normal operation,

there is a unique leader. A new leader is selected only when the current one fails. However, a unique leader is needed only to ensure progress and safety is guaranteed even if there is no leader or there are multiple leaders. These protocols can be used to decide on several things such as message ordering delivery or commit of distributed transactions.

Group communication differs from other message passing middleware in the consistency guarantees enforced in face of process and network faults by coordinating membership changes with message delivery. This is known as view synchrony and can be superficially described as ordering message delivery with view changes, thus enabling processes to handle each message in a common membership context. This reduces the complexity of coping with process failure when programming applications. Notice that services such as group membership and view synchronous reliable multicast encapsulate solutions to fundamental problems arising in the development of fault tolerant distributed systems such as database replication. This means that, although apparently similar to best-effort multicast protocols, the resulting programming paradigm is in fact as powerful as distributed atomic transactions. This makes group communication suitable for fault tolerant applications such as strong consistent replication of services while maintaining a simple and general purpose programming interface.

5.1.2 Advantages of group communication

Group communication eases the development of distributed fault tolerant applications by encapsulating solutions to complex problems in simple abstractions. In fact, it can be observed that by using group communication, the intuitive solution to replication problems is also the correct solution. This is best captured by presenting examples of distributed fault tolerant applications and overview their solutions and correctness arguments.

Information dissemination

An interesting application of group communication is the problem of fault tolerant state dissemination. A server process keeps a set of data items that change frequently. Such changes have to be propagated to a set of observer processes that are interested in current values of items. The goal is to ensure that:

1. If the server ceases to modify its state, all observers eventually have the same values for all the items as the server.

2. If the server fails, all observers eventually have the same values for all the items and that the combined state is the state of the server at some previous instant.

A solution using group communication is easily achieved. Each time the server changes the value of an item, it multicasts the identification of the item together with the new value to all observers. Upon delivery of a message, each observer updates the referred item with the new value.

Intuitively, the usefulness of reliable multicast can be illustrated by situations in which violation of any property of the specification leads to inconsistency. Such situations would have to be handled by the application when using unreliable multicast.

Primary-backup replication

A common strategy for implementing strong consistent replication is the primary-backup approach to replication [BMST93, GS97]: A single server - the primary - handles requests from clients. Upon executing each request the primary broadcasts a state update to backup replicas. A reply can be sent to client after acknowledgment messages are collected from backup replicas. Should the primary fail, a backup replica takes-over as the primary.

Implementing primary-backup replication involves determining the primary and updating backup replicas. The primary can easily be selected using the group membership service, for instance, by deterministically selecting one member of the group, that is ensured to be the same for all participants. When the primary fails and a backup needs to take over, it must be determined whether a consistent state has already been reached, as it is possible that late messages are still in transit. By using view synchrony the application can use membership notifications as indications that exactly the same messages have been delivered and thus that processes installing the new view are consistent.

Replicated state machine

An alternative strategy for replicating services is known as the replicated state machine [Sch93, GS97]. In contrast to primary-backup replication, all replicas handle requests directly from clients, executing them in parallel, updating their state and replying to clients. A client uses any of the replies[1].

Consistency is ensured by the determinism of replicas and by processing the same sequence of client requests. Violation of Validity means that the client might not get any reply. Violation of Agreement and Integrity might leave servers with inconsistent state that results later in inconsistent replies to a request. In addition, implementing a replicated state machine using group communication requires also that requests are totally ordered. Otherwise concurrent requests by multiple clients might be delivered by different orders. Notice that ensuring that the same sequence of messages is delivered to correct processes is not enough. If a process delivers a different sequence, replies to a client, and then crashes, it exposes inconsistent state to a client. This is solved by requiring the multicast protocol to implement Uniform Agreement.

5.2 Definitions and Properties

A number of properties developed over reliable multicast allow to implement the distributed state machine paradigm, which is an interesting paradigm for the support of database replica-

[1] Assuming no byzantine faults.

tion with strong consistency. This section begins by providing brief descriptions of a number of properties that complement reliable multicast and can show to be valuable in the scope of GORDA by contributing to the implementation of the distributed state machine or because they can simplify the middleware implementation. Comprehensive surveys of group membership and view synchrony properties are found in [HS95, CKV01]. Reliable multicast properties are found in [HT94, CKV01]. Algorithms implementing this specification can be found, for example, in [GS01b, SS93]. Reliable multicast is supported by a number of protocol composition frameworks. The chapter concludes with a comparison of these frameworks in respect to the properties provided and their adequacy to be used in the scope of GORDA.

5.2.1 Group Membership

The list of correct processes in a group is usually referred to as the view. A view change is triggered by the addition or removal of processes to the group. Processes install a new view when the membership changes. The history of addition and removal of processes to views is represented by sequentially numbering the views. View v_i succeeds view v_{i-1} and will be succeeded by view v_{i+1} as soon as a membership change is detected. Membership protocols differ in how new views are formed. Those supporting the primary partition model enforce a total order of view installation events, thus ensuring that each view is uniquely preceded and succeeded by a single other view. Primary partition group membership means that:

Primary Partition If process p installs view v_{i_p} and process q installs view v_{i_q}, then $v_{i_p} = v_{i_q}$.

As all processes agree on the i_{th} view, it is possible to refer to v_{i_p} simply as v_i. The alternative is the partitionable model allowing installation of concurrent disjoint views. New views result from merging or splitting previous views. This allows continued operation when a majority of processes is not reachable, at the expense of consistency.

In both situations, the events used to trigger view changes are determinant in the liveness guarantees offered to application programs. Examples of triggering mechanisms used in existing toolkits are unreliable failure detectors, resource availability and network connectivity.

5.2.2 Message Ordering

To ensure consistency on database replication, the messages of a group of replicas must be ordered. This order depends on the type of replication used. In this section we formally describe several types of message ordering, important for the GORDA project.

The simplest criterion for ordering deliveries is First-In-First-Out (FIFO), in which messages from the same sender are delivered in the order they were multicast:

Reliable FIFO Order If a process multicasts a message m before it multicasts a message m', no process delivers m' without previously delivering m.

A step further is given by ordering delivery according to the causal "happens before" relation [Lam78]. A message m causally precedes a message m' if some process sends m before

sending m'; or receives m before sending m'; or some m'' exists such that m precedes m'' and m'' precedes m'. Causal order multicast is [HT94, CKV01]:

Reliable Causal Order If a message m causally precedes message m', no process delivers m' without previously delivering m.

This is useful, for instance, when determining a global snapshot of the system state [CL85]. Most group communication toolkits also allow to totally order the delivery of concurrent messages, resulting in messages being delivered in the same order to all processes:

Total Order or Atomic Broadcast If a process delivers a message m before it delivers a message m', no process delivers m after delivering m'.

Notice that total order is orthogonal to both FIFO and causal ordering and thus can be combined with both. Total order is more costly to implement as it requires coordinating the delivery of concurrent messages. On the other hand, it is useful in implementing replicated services using the active replication approach [Sch93]. Totally ordered reliable multicast is also often called atomic multicast, as the effect of the multicast operation appears to occur instantaneously as it logically does not overlap with the delivery of other messages.

Alternative versions of FIFO and causal ordering do not require previous delivery of all predecessors, but prevent only out-of-order delivery. However, reliable versions of ordering properties are more useful and as easy to implement. Alternative formulations of the total order are weak total order [WS95], that is not enforced for processes that are expelled from the group and is thus less costly to implement, and total order to multiple overlapping groups, that enforces total order among messages with different destination sets.

5.2.3 Reliable Multicast

The multicast service is used through a pair of primitives: $multicast(m)$ and $deliver(m)$. A process executing the multicast primitive initiates the transmission of a message and is said to multicast m. By executing the deliver primitive, transmission is completed by handing over a message to the application at a destination process, which is said to deliver m. The definition of reliable multicast is as follows [HT94]:

Validity If a correct process multicasts a message m, then eventually it delivers m.

Agreement If a correct process delivers a message m, then all correct processes eventually deliver m.

Integrity For every message m, every process delivers m at most once and only if m was previously multicast by some process.

5.2.4 Uniformity

A stronger form of reliable multicast imposes consistency between correct and possibly failed processes. Is provided by some group communication toolkits. The resulting strengthened reliability model is known as Uniform Agreement or Safe Delivery [CKV01]:

Uniform Agreement If a process delivers a message m, then all correct processes eventually deliver m.

Although useful in maintaining consistency in distributed applications [GT91], implementation of Uniform Agreement is more costly in the number of messages exchanged.

5.2.5 View Synchrony

The coordination of reliable multicast and group membership is provided by view synchrony. If a process multicasts (resp. delivers) a message m after installing some view v_i and before installing any other view v_j we say m is multicast (resp. delivered) in view v_i. The definition is as follows [CKV01]:

View Synchrony If a process p installs two consecutive views v_i and v_{i+1} and delivers a message m in view v_i, then all other processes installing both v_i and v_{i+1} deliver m in view v_i.

An alternative formulation of view synchrony applies also to failed processes [SS93]. The rationale of the strengthened definition is similar to that of Uniform Agreement and has identical consequences both in consistency and implementation cost.

In addition to View Synchrony, it is also possible to enforce an additional restriction on message delivery regarding view installation:

Sending View Delivery If a process p multicasts a message m in view v_i, then p does not deliver m in a view other than v_i.

Enforcing simultaneously *Sending View Delivery* and *Validity* requires that processes are prevented from multicasting messages while a view is being installed [FvR95].

5.3 Group Communication Protocols

This section describes some existing group communication protocols. In order to support generic applications, these protocols assume that the application is correct. The group communication protocols must also ignore the application semantics. Even assuming application correctness, these protocols must support the system instability (node failures and network partitions).

Group awareness is a costly property. In order to provide group awareness, most group communication systems block processes from sending messages from the time that the need for

a view change is recognized until the view is delivered to the application. Such blocking can cause an expensive waste of valuable computation and network resources. Virtual Synchrony also requires ordering of messages and the usage of protocols that can be very costly. On the other hand, when the system is stable it's possible to provide performance with optimistic protocols. It is also possible to provide better performance when the application adds semantic information on the messages that need to be sent to the group. This Section describes these alternatives to improve the performance of group communication systems.

5.3.1 Optimistic Protocols

Optimistic protocols are based on optimistic assumption on the network connectivity and order of messages. There are several protocols based on these assumptions that will be described in this section.

Optimistic Virtual Synchrony

Optimistic virtual synchrony allows applications to send messages during periods in which existing group communication services block, by making optimistic assumptions on the network connectivity. [SKM00] describes how to build Virtual Synchrony allowing applications to to determine the policy as to when messages sent optimistically should be delivered and when they should be discarded and gives applications fine-grain control over the specific semantics they require.

In [SKM00], it is presented Optimistic Virtual Synchrony (OVS), a form of group communication that provides group awareness without the performance penalty of blocking. In Optimistic Virtual Synchrony, each view event is preceded by an optimistic view event, which provides the application with an estimate of the next view. After this event, applications may optimistically send messages that will provisionally be delivered in the next view. If some application defined property about the next view holds, then the messages will be delivered. Otherwise, the messages are rolled back, i.e. they are discarded, and the sending application is informed. Thus, the application specifies the policy for optimistic message delivery, and OVS provides the mechanism for implementing any application-specified policy.

Optimistic ordering of messages

Optimistic Atomic Broadcast [PS98] is an algorithm that exploits the spontaneous total order message reception property experienced in local area networks in order to allow fast delivery of messages. In some protocols such as those based on a sequencer the total order decided is the spontaneous ordering of messages as observed by some process. In addition, in local area networks it can be observed that the spontaneous ordering of messages is the same in all processes. The latency of total order protocols can therefore be masked by tentatively delivering messages based on this order, thus allowing the application to proceed the computation in parallel with the ordering protocol. Later, when the total order is established and if it confirms the

optimistic ordering, the application can immediately use the results of the optimistic assumption. If not, it must undo the effects of the computation. The effectiveness of the technique rests on the optimistic assumption that a large share of correctly ordered tentative deliveries offset the cost of undoing the effect of mistakes.

The optimistic assumption as it was described until know, only works in local area networks. On wide area networks, the spontaneous delivery of messages is not the same in all processes due to network latency. [SPMO02] extends the optimistic protocol described previously to wide area networks. It is proposed a protocol which enables optimistic total order to be used in wide area networks with much larger transmission delays where optimistic delivery does not normally hold. This proposal exploits local clocks and the stability of network delays to reduce the mistakes in the ordering of tentative deliveries by compensating the variability of transmission delays. This technique, when applied to a sequencer based protocol, does not introduce additional messages.

Although several algorithms that support optimistic delivery have been proposed before [PS98, FS01], these are specialized to some specific network types [PS98] or interaction patterns [FS01]. Vicente & Rodrigues in [VR02] present a different approach that is more generic because, in stable conditions, both the optimistic and uniform order are derived from the output of a fully fledged non-uniform total order algorithm. The principles of this approach are the following. An efficient algorithm that provides non-uniform total order assuming a perfect failure detector is used to provide fast optimistic delivery. If processes are not suspected, this optimistic order is made uniform through an additional round of message exchange. If processes are suspected, a consensus-based reconfiguration phase is executed to ensure the termination of pending broadcasts (determining a certain delivery order) and to reconfigure the operation mode for the next stable period. The reconfiguration procedure does not assume a perfect failure detector. The challenge of this algorithm is to ensure that the order established by the reconfiguration phase never conflicts with the order established during the stable-period.

5.3.2 Semantically reliable multicast

Semantic reliability makes usage of information about message semantics, provided by the applications, in order to purge obsoleted messages. When a message m is obsoleted by other message m', it is represented as $m \sqsubset m'$. A more detailed description about semantic reliability can be found in [PRO00]. A semantically reliable protocol provides consistency guarantees even when a sender fails, thus being suited to be used for replication in fault tolerant systems. Semantically Reliable Multicast (S-RM), a protocol that enforces an agreement property despite the correctness of the sender, is defined by:

Semantic Validity: If a correct process multicasts a message m and there is a time after which no process multicasts m'' such that $m \sqsubset m''$, then it delivers some m' such that $m \sqsubseteq m'$.

Semantic Agreement: If a correct process delivers a message m and there is a time after which no process multicasts m'' such that $m \sqsubset m''$, then all correct processes deliver some m'

such that $m \sqsubseteq m'$.

Integrity: For every message m, every process delivers m at most once and only if m was previously multicast by some process.

FIFO Delivery: If a process multicasts a message m before it multicasts a message m', no process delivers m after delivering m'.

Semantic FIFO Completeness: If a process multicasts a message m before it multicasts a message m' and there is a time after which no process multicasts m''' such that $m \sqsubset m'''$, no correct process delivers m' without delivering some m'' such that $m \sqsubseteq m''$.

This protocol differs from S-SRM by including agreement properties. Semantic Agreement is similar to the Agreement property in conventional reliable multicast and ensures that messages are delivered to either all or none of correct processes despite the failure of the sender.

Together, Semantic Agreement and Semantic FIFO Completeness properties ensure that the same prefix of non-obsolete messages are received by all correct processes. This can be used, for instance, in the primary-backup replication scenario outlined. Notice that when the obsolescence relation is empty, this specification reduces to conventional reliable multicast [HT94]. On the other hand, if every message makes all its predecessors obsolete, it results in an extension to multicast of *stubborn channels* [GOS98].

5.3.3 Semantically view synchronous multicast

To describe the combination of view synchrony with semantic reliability we consider a simplified group membership service in which processes can only leave the group. The kind of events that may lead to a view change are not relevant to the definition of Semantic View Synchrony. Examples of possible causes for triggering a view change to remove a process from the group are the occurrence of failure suspicions [LH99], the lack of available buffer space at one or more processes [CBDS01] and simply the existence of processes that voluntarily want to leave. The properties for Semantically View Synchronous Multicast (S-VSM) are:

Semantic View Synchrony: If a process p installs two consecutive views v_i and v_{i+1} and delivers a message m in view v_i, then all other processes installing both v_i and v_{i+1} deliver some m' such that $m \sqsubseteq m'$ before installing view v_{i+1}.

Integrity: For every message m, every process delivers m at most once and only if m was previously multicast by some process.

FIFO Delivery: If a process multicasts a message m before it multicasts a message m', no process delivers m after delivering m'.

Semantic FIFO View Completeness: If a process multicasts a message m before it multicasts a message m' and a process p installs two consecutive views v_i and v_{i+1}, and delivers message m' in view v_i, then q delivers some m'', such that $m \sqsubseteq m''$, before installing view v_{i+1}.

The Semantic FIFO View Completeness property relaxes the traditional Reliable FIFO properties [CKV01]. Given a sequence of messages multicast by a process, this ensures that upon view installation only obsolete predecessors of the last message delivered can be omitted. With Semantic View Synchrony every two processes installing two consecutive views v_i and v_{i+1} do not necessarily deliver the same sequence of messages, thus being weaker than View Synchrony, but they are ensured to deliver (at least) the same sequence of messages that have not been made obsolete by subsequent messages up to view v_{i+1}. For instance, process p_1 may deliver in view v_i messages m_1 and m_2, such that $m_1 \sqsubset m_2$, and process p_2 may only deliver m_2 in the same view. If no messages m, m' exist such that $m \sqsubset m'$, Semantic View Synchrony reduces to conventional View Synchrony. This makes it more general as different concrete semantics, including View Synchrony, can be obtained by defining an appropriate obsolescence relation.

5.3.4 Generic Broadcast

Another way of using semantic information to improve the system performance is described in [PS99]. Although message ordering is a fundamental abstraction in distributed systems, these ordering guarantees are just syntactic in the sense that they do not take into account the semantics of the messages. For instance, active replication relies on total order delivery of messages on the active replicated servers. By considering the semantics of the messages sent to the active replicated servers, total order may not always be needed, for instance, if we distinguish read messages from write messages sent to active replicated servers since read messages do not need to be ordered with respect to other read messages. As message ordering has a cost, the work described in [PS99] avoids message ordering when not required. The Generic Broadcast establishes a partial order on message delivery. Semantic information about messages is introduced in the system by a conflict relation defined over a set of messages.

Generic Broadcast is specified by a conflict relation and four properties: (i) Validity, (ii) Agreement, (iii) Integrity and (iv) Partial Order. The first three properties are already defined in Section 5.2.3. Partial Order says that if two correct processes deliver m and m', and if m and m' conflict then the two processes deliver m and m' in the same order. The conflict relation determines the pair of messages that are sensitive to the delivery order.

5.4 Group Communication Toolkits

This section resumes the features of the existing group communication systems. For each system we give an overview, including the language used to implement the system and interfaces, and the license that protects the software. Finally, we add some notes that serve has guidelines to chose the system that it will be used in GORDA.

5.4.1 Previous toolkits

The current existing toolkits are based on research made on previous group communication toolkits, that will be resumed in this section.

Isis was one of the first systems to provide group communication and was developed out of a study of fault tolerance in distributed systems. The system implements a collection of techniques for building software for distributed systems.

Horus was originally perceived as a redesign of the Isis group communication system, but has evolved into a general purpose communication architecture with advanced support for the development of robust distributed systems in wide variety of settings. The Ensemble toolkit, described later, is the next generation of Horus.

Totem provides reliable totally ordered delivery of messages to processes within process groups on a single local-area network, or over multiple local-area networks interconnected by gateways. Superimposed on each local-area network is a logical token-passing ring. The fields of the circulating token provide properties such as reliable delivery and total ordering of messages.

Transis is a multicast communication layer built at the *University of Jerusalem* that introduced causal order in group communication systems.

5.4.2 Appia Framework

Appia is a layered communication support framework that was implemented in *University of Lisbon*. It is protocol independent, that is, the framework layers any protocol as long as it respects the predefined interface. This services can be found in several previous works. Different ordering of protocols can also provide different set of properties.

Appia is implemented in Java and is very flexible and can build communication channels that fit exactly in the user needs. Channels can have common sessions and can have different QoSs. All this flexibility is payed with performance. Appia has some features that make the performance in communication be lower than other systems. To improve the communication performance, Appia must have a good flow control protocol in the network and inside the channel. It is also needed to verify with some profiler some lacks in the code and improve it. Appia is maintained by FCUL and its license permits the usage of the source on the GORDA project.

5.4.3 Spread Toolkit

Spread is a toolkit implemented by researchers of the *Johns Hopkins University* and provides a messaging service that is resilient to faults across external or internal networks. Spread functions as a unified message bus for distributed applications, and provides application-level multicast and group communication support. Spread services range from reliable message passing to fully ordered messages with delivery guarantees, even in case of computer failures and network partitions.

The Spread system is based on a daemon-client model where generally long-running daemons establish the basic message dissemination network and provide basic membership and

ordering services, while user applications linked with a small client library can reside anywhere on the network and will connect to the closest daemon to gain access to the group communication services.

Spread is configurable to local and wide area networks. It has interfaces in several languages. Its performance is a unknown, but previous work [BCMT02] suggest it is not the best. The mailing lists are populated and there are people always responding, which means that the system is supported.

Spread is written in C, and has the following interfaces:

1. C/C++ libraries with and without thread support.
2. Java Class to be used by applets or applications.
3. Perl interface.
4. Python interface.
5. Ruby interface.

The basic Spread toolkit is available under the Spread Open Source license[2] and may be freely used under some conditions about advertising. Spread Concepts provides additional licensing options and commercial support for the Spread Toolkit.

5.4.4 Ensemble Group Communication System

Ensemble is the next generation of the Horus group communication toolkit, both developed at Cornell University. They seek to introduce guarantees such as reliability, high availability, fault-tolerance, consistency, security and real-time responsiveness into applications that run on modern networks.

Their broad approach is to focus on what they call process group communication structures that arise in many settings involving cluster-style computing, scalable servers, groupware and conferencing, distributed systems management, fault-tolerance and other execution guarantees, replicated data or computing, distributed coordination or control, and so forth.

Ensemble is structured as a client-server system with a server providing group-communication services through a socket based interface. Clients can connect to the server and send/receive reliable point-to-point and multicast messages. The server is written (mostly) in the OCaml programming language, the client is a small library that has implementations in several languages. There are clients in C and Java.

Previous versions of the system did not distinguish between client and server. The client was implemented with an internal server. This provides good performance. However, since the server is written in ML, in order to link with a C program written by a user the foreign-language interface of ML needs to be used. This causes very difficult portability issues. As of

[2] `http://www.spread.org/download.html`

release 2.00 it was decided to separate client from server; this should improve portability at the expense of performance.

Ensemble is implemented in OCaml, but has also interfaces in C and Java. There is a TCP interface for the group membership service. This toolkit is well known by some members of FCUL and it seems to be maintained by the authors. Ensemble provides good performance on group communication. There is a good set of protocols that GORDA can benefit and there are several interfaces written in several languages.

The Ensemble license[3], similar to the Appia license, permits the usage of the work in other projects.

5.4.5 JGroups Toolkit

JGroups is a Group Communication tool implemented in Java and provides a flexible protocol stack architecture that allows users to put together custom-tailored stacks, ranging from unreliable but fast to highly reliable but slower stacks.

Because of its implementation in Java (like Appia), JGroups could be bad to the performance. The channels created on JGroups are not so flexible has we can build using Appia. Sharing sessions of a layer can be a way to improve performance in some cases, but this cannot be done in JGroups. INRIA and a research group in University of Rome La Sapienza tested this system and they show the lack of performance of JGroups, even when compared to other group communication systems.

JGroups is open source licensed under the Library (or Lesser) GNU Public License (LGPL). LGPL relaxes the GPL constraints by allowing the software to be included in any product being it closed or open source. However, modifications to the original code should be released under LGPL.

5.4.6 Cactus Framework

The goal of Cactus is to develop a design and implementation framework for supporting customizable dynamic fine-grain Quality of Service (QoS) attributes related to dependability, real time, and security in distributed systems. This framework is based on a multi-level integrated approach, with middleware services between the application and operating system implementing the relevant attributes. Cactus has implementations in C, C++ and Java, but the Java implementation is incomplete. The Cactus license was not found in the web page. There is only a Copyright Notice included in the source code that gives no warranty on the usage of the software.

[3] `http://www.cs.cornell.edu/Info/Projects/Ensemble/license.html`

5.5 Performance evaluation

There are several competitive products for database replication that have been built based on group communication primitives, such as the ones described in the Section 5.4. Given that atomic broadcast is an expensive procedure, and that many different implementations are available, it is interesting to identify which aspects in its implementation are most relevant for the support of database replication and what is the influence of group communication toolkits in the overall system performance. For that purpose, this Section briefly describes and the performance of several different atomic broadcast toolkits in the support of Sequoia[4], a middleware framework for database replication.

5.5.1 The Experimental Testbed

This Section describes the Sequoia database replication framework and show how does this system uses the atomic broadcast primitive. This Section also introduces the several group communication toolkits used in Sequoia.

Sequoia

Sequoia [Con06b] is a middleware database replication system that exports a JDBC interface to applications and routes client requests to a set of databases, through a controller. Sequoia is composed by a JDBC driver, that is used by applications that want to access the databases and a controller that receives the client requests and forward them to a set of databases. For availability and fault tolerance, the Sequoia controller can be replicated. Each controller manages a set of databases. In a system with more than one controller, the application can use any controller to make the requests. The controllers exchange their requests using an atomic broadcast primitive, in order to execute the same set of requests in the same order in all databases.

Sequoia is an interesting framework because is highly configurable and can reflect the most heterogeneous scenarios. The Sequoia framework behaves as a JDBC driver for database clients. It contains a controller that receives the client requests and forwards them to all databases. Only writes are forwarded to all databases, reads are load-balanced. The Sequoia controller is replicated for fault tolerance and uses group communication to disseminate write requests to all other controllers.

The implementation of primitives that make use of group communication is distributed as a separate package called Hedera [Con06a]. In detail, it provides access to an application specific subset of group communication and additional functionality for explicitly acknowledged messages, multiplexing and dispatching. Hedera has been previously implemented twice, using JGroups and Appia. We implemented Hedera also with Spread. Although there are other group communication toolkits, such as JazzEmsemble [Hay98] and Cactus [HSW99], we will briefly describe here just the ones used in Sequoia.

[4]Previous tests were made using the ObjectWeb C-JDBC, at the beginning of the GORDA project. This Section shows only the latest results.

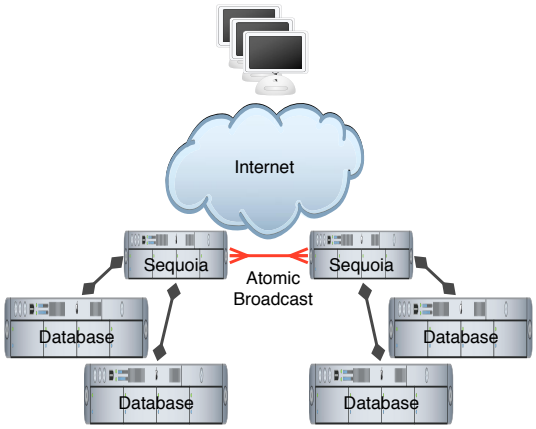


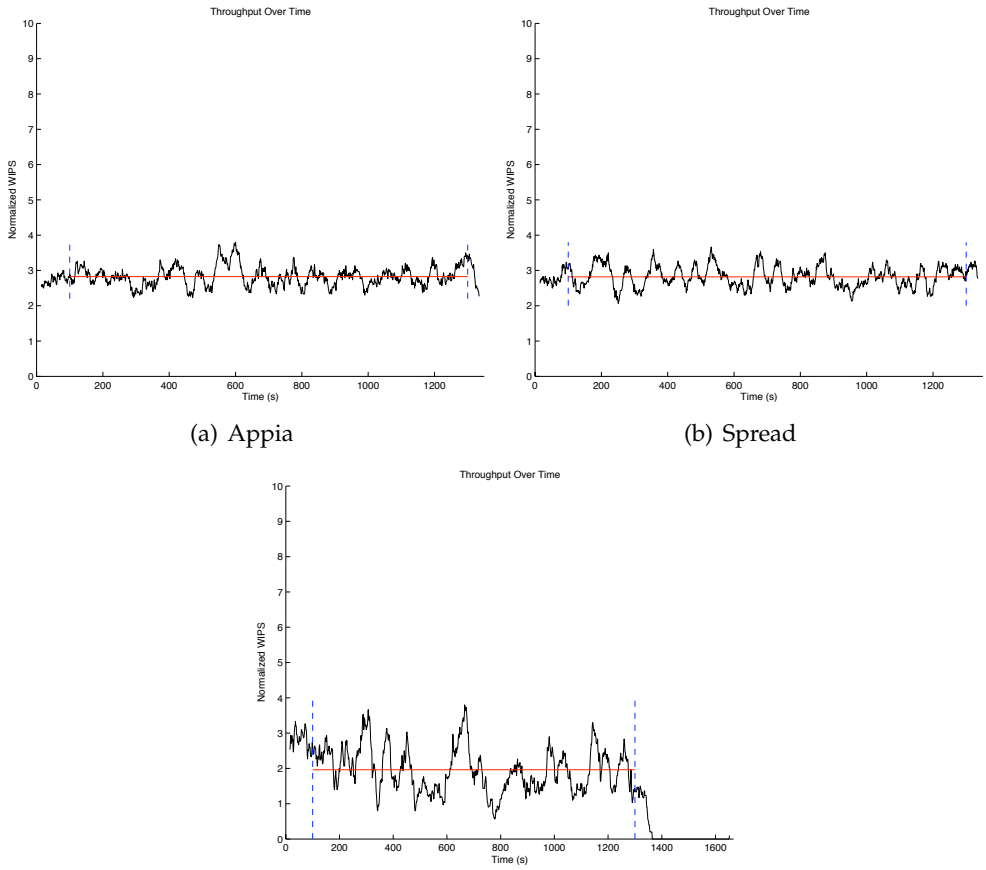
Figure 5.1: Sequoia architecture with a replicated controller.

5.5.2 Experimental environment

One typical scenario is the replication of databases that maintain e-commerce information. In an e-commerce scenario with a big number of clients, the sales volume can increase with the system performance. If the system can handle transactions more efficiently, more transactions per time unit are processed and more products are sold and delivered to clients. Using this scenario we will show that the system overall performance can be different just by choosing a different group communication toolkit.

To materialize the impact of the group communication framework on a real system, we used a benchmark for an e-commerce scenario: a transactional web benchmark that simulates an on-line book store. The workload is performed in a controlled Internet commerce environment that simulates the activities of a business oriented transactional web server. The workload includes multiple on-line browser sessions, dynamic page generation with database access and update, simultaneous execution of multiple transaction types that span a breadth of complexity, databases consisting of many tables with a wide variety of sizes, attributes, and relationships, transaction integrity (ACID properties) and contention on data access and update.

The performance metric reported by the benchmark is the number of web interactions processed per second (WIPS). Multiple web interactions are used to simulate the activity of a retail store. The benchmark simulates three different profiles by varying the ratio of browse to buy: primarily shopping, browsing and web-based ordering. In the tested system, we are interested in having a large amount of write operations and some read operations that can conflict with write operations. The amount of write operations depends on the profile. The shopping profile has 20% of write operations, the browsing profile has 5% of write operations and the web-based ordering profile has 50% of write operations.



(a) Appia (b) Spread

(c) JGroups

Figure 5.2: Throughput over time of Sequoia in WIPS.

5.5.3 Experimental results

The framework used to provide group communication primitives to the sequoia controller could be more or less significant to the system performance, depending on several variables and scenarios. In this section we will show a worst case scenario where the group communication primitive is widely used. We will discuss the group communication impact on this scenario.

The system is composed by a set of MySQL databases, replicated using Sequoia. The application that we run was a Java implementation of previously described transactional web benchmark [RDJB01, oWM06] that uses emulated browsers to access to a virtual book store. The emulated browsers access the system using the Jakarta Tomcat [Fou06] application server.

The tests focused on the influence of atomic broadcast primitive on the system. To focus on this primitive, the system is configured as follows. There are 3 machines, each one with one MySQL database controlled by a Sequoia controller. The clients run on a separated machine and make HTTP requests through the Tomcat application server. The HTTP requests are converted into SQL requests that are sent (by the Tomcat) to the Sequoia controller. The Sequoia controller sends them to the atomic broadcast primitive and, upon reception, sends it to the

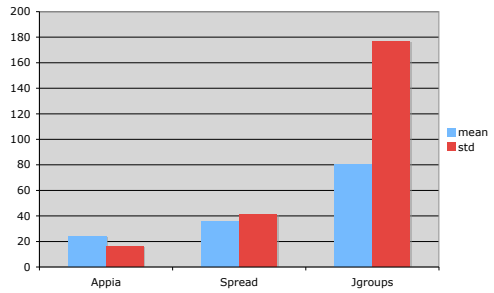


Figure 5.3: Latency of Requests on Sequoia using 3 controllers (ms).

MySQL database. All databases receive the same set of requests in the same order. All the four machines are in the same switched local network. The tests used a virtual synchrony protocol stack and a token based total order protocol. Each member of the group runs in a Pentium IV/2.8GHz server with 1Gb of memory. The three machines are connected through 100Mbps ethernet switch. Each test was made with different message sizes.

The Figure 5.2 shows the throughput over time in WIPS of the Sequoia controller, using Appia, JGroups and Spread group communication toolkits. The figures show that with Appia and Spread the system have an equivalent performance, providing (in average) almost 3 normalized WIPS. When Sequoia is configured to use JGroups, the overall performance of the system decreases and only 2 normalized WIPS are computed. We can also see that the system is more unstable with JGroups.

5.5.4 Analysis

After showing that the overall performance of the system can differ just by changing from one group communication toolkit to other that provide the same service, we will discuss some reasons for these differences.

There are several options that can be made when building a group communication system. The architecture that can be used to implement such a system can also influence the performance of the group communication toolkit. One option that can be made when building a protocol composition framework is using a single threaded kernel vs using one thread (or a set of threads) for each protocol. For instance, Appia uses the single thread approach and JGroups can be configured to use two threads for each protocol. A big number of threads in a process can decrease its performance, since it requires context switching between threads.

Another architecture option is to have a separation between the application and the protocols that ensure the message properties, having both on separate processes. This can increase the performance of the system if the machine were it runs have more resources available. One drawback is the need for communication and synchronization between these processes. This is the case of the Spread toolkit. The results show that both Appia – that uses an approach that contains the application and the protocol stack in the same process – and Spread have equivalent performance results.

When writing protocol and applications using the Java language, one drawback for the performance of a system is the usage of the Java Serialization. This is known to be slow, thus need to be avoided. That is one big difference between Appia and JGroups. Appia optimizes its messages and avoids the Java Serialization using more complex techniques to read and write from/to buffers. It also avoids copies of buffers using techniques from x-kernel [HP91]. On the other side, JGroups uses the Java Serialization. This could be one cause for the performance lack when JGroups is compared to the other systems.

Taking a deeper look at the results, we can see in the Figure 5.3 the average and standard deviation time for a request on Sequoia, using 3 controllers. The figure shows that Appia remains the most stable system and achieves performance results compared to Spread. The Appia behavior can be explained by the usage of such optimizations previously described.

Chapter 6

Conclusion

6.1 Target Applications

Motivation for database replication is either better performance or fault-tolerance. Replication for performance while maintaining strong consistency usually reduces to allowing parallel read operations, which has shown to be useful for a large number of usage scenarios. Some proposals allow parallel updates by partial replication. Otherwise, lazy protocols that allow for inconsistency are used. An extreme case of these is disconnected operation in which updates can only be propagated in specific periods due to unavailability of network connectivity.

Replication for fault-tolerance addresses mostly site crashes. A notable exception is RAIDb which tolerates byzantine faults. This is specially interesting when replicating updates across multiple distinct database management engines.

The usage of volume level replication, shows that there is a demand for eager replication, even with this costly and limited approach.

6.2 Architecture and Interfaces

It is also interesting to consider how existing implementations of replication protocols interface with database management systems. Most implementations of database replication use client interfaces in one of two ways:

- The DBMS is wrapped and thus all client requests are intercepted. This allows for arbitrary manipulation of requests that can be examined, routed, and transformed. Transformed and additional requests made to the backend database use common client interfaces. This approach has the drawback of restricting what clients can be connected, but supports eager replication and greater flexibility, although sometimes at the expense of duplicating mechanisms inside database management systems (e.g. concurrency control).

- Clients access the database directly by means of any available client interface. Replication mechanisms use the database as any other client and usually install triggers to

receive notification of updates that need to be replicated, which might not be completely transparent to users. This mechanism is used commonly for asynchronous replication.

Privileged interfaces are found in some proposals and are used mainly for manipulating information that is not available or is not efficiently accessed by client interfaces. Namely, these are used to collect and deliver updates without the overhead of triggers and conversion to SQL statements. In these, we include direct reading of transaction logs.

Finally, interfacing by means of operating system I/O interfaces, as done in volume replication, is useful strictly for volume replication and is therefore of very limited interest for the project.

6.3 Replication Protocols

Existing work on group-based database replication establishes a set of protocols for a variety of environments and application scenarios. Existing prototype implementations and simulation models show that such protocols are interesting both for performance and consistency.

On the other hand, there are some open problems and several on-going research efforts. When implementing group-based database replication we are concerned mostly with how to interface group-based replication protocols with a large spectrum of DBMSs, as existing implementations are scarce and tightly coupled with specific DBMSs. Finally, replica management and recovery in the context of group replication protocols has not been thoroughly examined.

Bibliography

[AA92] D. Agrawal and A. El Abbadi. The generalized tree quorum protocol: an efficient approach for managing replicated data. *ACM Transactions on Database Systems (TODS)*, 17(4):689–717, 1992.

[ABKW98] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: are these mutually exclusive? In *Proceedings of the ACM SIGMOD International Conference on Management of data*, 1998.

[ACZ03] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003.

[ADA+02] Y. Amir, C. Danilov, M. M. Amir, J. Stanton, and C. Tutu. Practical wide-area database replication. John Hopkins University, USA.CNDS Technical Report, http://cnds.jhu.edu/publications, 2002.

[ADA+03] Y. Amir, C. Danilov, M. M. Amir, J. Stanton, and C. Tutu. On the performance of consistent wide-area database replication. John Hopkins University, USA. CNDS Technical Report, http://cnds.jhu.edu/publications, 2003.

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *IEEE International Symposium on Fault-Tolerant Computing*, July 1992.

[ADS00] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *IEEE International Conference on Dependable Systems and Networks*, June 2000.

[Alo97] G. Alonso. Partial database replication and group communication primitives, 1997. In 2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97), Zinal (Valais, Switzerland) , March 1997.

[AT02] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the International Conference on Dependable Systems and Networks (ICDCS)*, pages 494–503. IEEE, 2002.

[ATS+05] F. Akal, C. Türker, H. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

[BCMT02] R. Baldoni, S. Cimmino, C. Marchetti, and A. Termini. Performance Analisys of Java Group Toolkits: a Case Study. In *Proceedings of the International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI'2002)*, volume 2604 of *Lecture Notes in Computer Science*, pages pp. 49–60, Luxembourg-Kirchberg, Luxembourg, November 2002. Springer.

[BG84] P. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems (TODS)*, 9(4):596–615, December 1984.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BKR+99] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. In *Proceedings of the ACM SIGMOD International Conference on Management of data*, 1999.

[BL82] H. Breitwieser and M. Leszak. A distributed transaction processing protocol based on majority consensus. In *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 224–237, 1982.

[BMST93] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*, chapter 8. Addison Wesley, 1993.

[BvR94] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[CBDS01] B. Charron-Bost, X. Défago, and A. Schiper. Time vs. space in fault-tolerant distributed systems. In *IEEE International Workshop on Object-oriented Real-time Dependable Systems*, January 2001.

[CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4), December 2001.

[CL85] K. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1), February 1985.

[CMZ04] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Partial replication: Achieving scalability in redundant arrays of inexpensive databases. *Lecture Notes in Computer Science*, 3144:58–70, July 2004.

[CNO99] J. Cowie, D. Nicol, and A. Ogielski. Modeling the global Internet. *Comp. in Science and Eng.*, 1(1), January/February 1999.

[Con04] GORDA Consortium. User requirements report. Technical report, GORDA Project, 2004.

[Con06a] Continuent. Hedera version 1.5. http://hedera.continuent.org, 2006.

[Con06b] Continuent. Sequoia version 2.9. http://sequoia.continuent.org, 2006.

[Cor02] Veritas Software Corp. A guide to understanding veritas volume replicator, 2002.

[Dat 7] IBM DB2 Universal Database. Replication guide and reference. Technical Report SC26-9920-00, IBM, Version 7.

[DS83] D. Daniels and A. Spector. An algorithm, for replicated directories. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 104–113, 1983.

[e02] L. Marowsky-Br e. Drbd performance. http://www.drbd.org/performance.html, 2002.

[EMS95] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, May 1995.

[ES83] D. Eager and K. Sevcik. Achieving robustness in distributed database systems. *ACM Transactions on Database Systems (TODS)*, 8(3):354–381, 1983.

[Fou06] The Apache Software Foundation. Apache tomcat. http://tomcat.apache.org/, 2006.

[FS01] P. Felber and A. Schiper. Optimistic active replication. In *Proceedings of 21st International Conference on Distributed Computing Systems (ICDCS'2001)*, Phoenix, Arizona, USA, April 2001. IEEE Computer Society.

[FvR95] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. Technical Report TR95-1537, Cornell University, Computer Science Department, August 1995.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *ACM SIGMOD International Conference on Management of Data*, 1996.

[Gif79] D. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh symposium on Operating systems principles*, pages 150–162, 1979.

[GMUW02] H. Garcia-Mollina, J. Ullman, and J. Widom. *Database Systems The Complete Book*. Prentice Hall, 2002.

[GOS98] R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels. Technical Report 98-278, Département d'Informatique, École Polytechnique Fédérale de Lausanne, 1998.

[GPN05] S. Gançarski, C. Le Pape, and H. Naacke. Fine-grained refresh strategies for managing replication in database clusters. In *Proceedings of VLDB Workshop on Design, Implementation and Deployment of Database Replication*, 2005.

[Gro] Slony Development Group. Slony. http://slony.info.

[GS97] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4), April 1997.

[GS01a] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Trans. Softw. Eng.*, 27(1):29–41, 2001.

[GS01b] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1), January 2001.

[GSC+83] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, and D. Ries. A recovery algorithm for a distributed database system. In *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 8–15. ACM Press, 1983.

[GT91] A. Gopal and S. Toueg. Inconsistency and contamination. In *ACM Symposium on Principles of Distributed Computing*, August 1991.

[HAA99] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group multicast. In *Proc. of FTCS'99*, pages 158–165, 1999.

[Hay98] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Computer Science Department, January 1998.

[Hol01] J. Holliday. Replicated database recovery using multicast communications. In *Proceedings of Symposium on Network Computing and Applications (NCA'01)*, pages 104–107, 2001.

[HP91] N. C. Hutchinson and L. L. Peterson. The x-kernel: An ar5chitecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[HS95] M. Hiltunen and R. Schlichting. Properties of membership services. In *IEEE International Symposium on Autonomous Decentralized Systems*, April 1995.

[HSW99] M. Hiltunen, R. Schlichting, and G. Wong. Implementing integrated fine-grain customizable qos using cactus. In *Fast Abstracts, The 29th International Symposium on Fault-Tolerant Computing Systems*, pages 59–60, Madison, Wisconsin, USA, June 1999.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[JPPMA02] R. Jimenez-Peris, M. Patino-Martinez, and G. Alonso. An algorithm for non-intrusive, parallel recovery of replicated data and its correctness. In *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS'02)*, Osaka, Japan, October 2002.

[JPPMAK03] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294, September 2003.

[JPPMKA02] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[KA98] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *International Conference on Distributed Computing Systems*, pages 156–163, 1998.

[KA00] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceeding of the 26th VLDB Conference*, 2000.

[KBB01] B. Kemme, A. Bartoli, and Ö. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*. IEEE, July 2001.

[KPA+03] 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.

[KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proc. the Int'l Conf. on Dist. Comp. Syst.*, Austin, Texas, June 1999.

[Lam78] L. Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7), 1978.

[Lam98] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[LH99] K. Lin and V. Hadzilacos. Asynchronous group membership with oracles. In *International Symposium on Distributed Computing (DISC)*, 1999.

[Mal96] C. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, Département d'Informatique, École Polytechnique Fédérale de Lausanne, 1996.

[Mic05] Microsoft. *Microsoft SQL Server 2000*. Microsoft, 2005.

[MPR01] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, April 2001.

[Ora97] Oracle. Oracle8 product documentation library, 1997.

[oWM06] University of Wisconsin Madison. Tpc-w in java. http://www.ece.wisc.edu/ pharm/tpcw.shtml, 2006.

[PA04] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, 2004.

[Ped99] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[PF00] F. Pedone and S. Frolund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, page 176. IEEE Computer Society, 2000.

[PGS02] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14:71–98, 2002.

[PMJPKA00] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters, 2000. In Proc. of DISC, Toledo, Spain, 2000.

[PMJPKA05] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems (TOCS)*, 2005.

[pos] PostgreSQL. http://www.postgresql.org.

[PRO00] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically reliable multicast protocols. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Nurnberg, Germany, October 2000.

[PS98] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC'98, formerly WDAG)*, September 1998.

[PS99] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, 1999.

[rac04] Scalable simulation framework. http://www.ssfnet.org, 2004.

[RDJB01] D. Menascé R. Dodge JR and D. Barbará. Testing e-commerce site scalability with tpc-w. In *Proceedings of 2001 Computer Measurement Group Conference*, Orlando, FL, December 2001.

[Rei02] P. Reisner. Drbd – distributed replicated block device. In *9th Intl. Linux System Technology Conference*, 2002.

[RV92] L. Rodrigues and P. Veríssimo. xAMp: a multi-primitive group communications service. In *IEEE International Symposium on Reliable Distributed Systems*, October 1992.

[Sch93] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, chapter 7. Addison Wesley, second edition, 1993.

[SKM00] J. Sussman, I. Keidar, and K. Marzullo. Optimistic virtual synchrony. In *Symposium on Reliability in Distributed Software*, pages 42–51, 2000.

[SPMO02] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, pages 190–199. IEEE CS, October 2002.

[SPOM01] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *IEEE Int'l Symp. Networking Computing and Applications*. IEEE CS, October 2001.

[SPS+04] A. Sousa, J. Pereira, L. Soares, A. Correia Jr.and L. Rocha, R. Oliveira, and F. Moura. Testing the dependability and performance of group communication based database replication protocols. Technical report, University of Minho, 2004.

[SS93] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, May 1993.

[Syb03] Sybase. Replication strategies: Data migration, distribution and synchronization. Technical Report L02038 MIL6143, Sybase, Inc., 2003.

[TGGL82] I. Traiger, J. Gray, C. Galtieri, and B. Lindsay. Transactions and consistency in distributed database systems. *ACM Transactions on Database Systems (TODS)*, 7(3):323–342, 1982.

[Tho79] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, jun 1979.

[TKS] L. Tan, M. Kerby, and J. Senicka. Veritas performance brief – remote mirroring using vxvm for metropolitan area disaster recovery. Veritas Software Corp.

[(TP01] Transaction Processing Performance Council (TPC). TPC Benchmark^{TM} C standard specification revision 5.0, February 2001.

[Urb03] R. Urbano. *Oracle Streams Replication Administrator's Guide*. Oracle, release 1 edition, December 2003.

[VR02] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of the 21th IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 92–101, Osaka, Japan, October 2002.

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings of 19th IEEE International Symposium on Reliable Distributed Systems (SRDS2000)*, pages 206–215, Nürnberg, Germany, October 2000. IEEE Computer Society.

[WS95] U. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *IEEE International Symposium on Reliable Distributed Systems*, September 1995.