Chapter 13
Practical Database Replication

Alfrânio Correia Jr., José Pereira, Luís Rodrigues, Nuno Carvalho, and Rui Oliveira

Abstract This chapter illustrates how the concepts and algorithms described earlier in this book can be used to build practical database replication systems. This is achieved first by addressing architectural challenges on how required functionality is provided by generally available software componentes and then how different components can be efficiently integrated. A second set of practical challenges arises from experience on how performance assumptions map to actual environments and real workloads. The result is a generic architecture for replicated database management systems, focusing on the interfaces between key components, and then on how different algorithmic and practical optimization options map to real world gains. This shows how consistent database replication is achievable in the current state of the art.

13.1 Introduction

This chapter illustrates how the concepts and algorithms described earlier in this book can be used to build practical database replication systems. Hereafter a practical database replication system is a system that has the following qualities:

- It can be configured to tune the performance of multiple database engines and execution environments (including different hardware configurations of the node replicas and different network configurations).
- It is modular: the system provides well defined interfaces among the replication protocols, the database engines, and the underlying communication and coordination protocols. Thus, it can be configured to use the best technologies that fit a given target application scenario.
- Its modularity is not an impairment to performance. In particular it provides the hooks required to benefit from optimizations that are specific to concrete database or network configurations.
- It combines multiple replica consistency protocols in order to optimize its performance under different workloads, hardware configurations, and load conditions.

B. Charron-Bost, F. Pedone, and A. Schiper (Eds.): Replication, LNCS 5959, pp. 253–285, 2010.

To achieve these goals we have defined an architecture based on three main blocks:

- replication-friendly database,
- group communication support, and
- pluggable replica consistency protocols.

First of all, to achieve modularity without losing performance, the system needs to have replication support from the database engine. As we will see later in this chapter, the client interfaces provided by a Database Management System (DBMS) do not provide enough information for replication protocols. The replication protocols need to know more about the intermediate steps of a transaction in order to achieve good performance. Secondly, we will focus on group communication-based replication protocols. A Group Communication Service (GCS) eases the implementation of replication protocols by providing abstractions for message reliability, ordering and failure detection. In this chapter, we will discuss some details that need to be addressed when applying GCS to practical database replication systems. Finally, we will describe the replication protocols, how they interact with the other building blocks and show how they can be instantiated using different technologies. The achievements described here are the result of our experience in architecting, building and evaluating multiple instantiations of our generic architecture [9, 13].

The rest of the chapter is structured as follows. An architecture for practical database replication is presented in Section 13.2. Then, we devote a separate section to each main component of the architecture. In detail: Section 13.3 describes how to offer replication-friendly database support; Section 13.4 presents the necessary communication and coordination support to the pluggable replication protocols, which are described in Section 13.5. Section 13.6 presents an evaluation of several consistent database replication protocols on top of the described architecture. Section 13.7 concludes the chapter.

13.2 An Architecture for Practical Database Replication

In the following paragraphs we will briefly describe a generic architecture for practical database replication. The architecture, illustrated in Figure 13.1, is composed of the following building blocks:

- The *Application,* which might be the end-user or a tier in a multi-tiered application.
- The *Driver* provides a standard interface for the application. The Driver provides remote accesses to the (replicated) database using a communication mechanism that is hidden from the application, and can be proprietary.
- The *Load Balancer* dispatches client requests to database replicas using a suitable load-balancer algorithm.
- The *DBMS, or Database Management System,* which holds the database content and handles remote requests to query and modify data expressed in standard SQL.

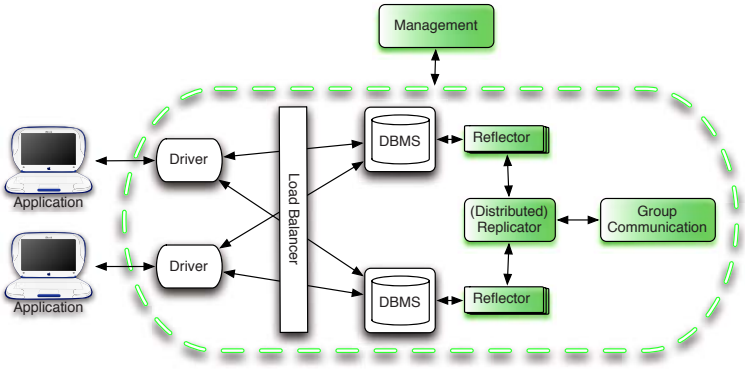


Fig. 13.1 Generic architecture for replication.

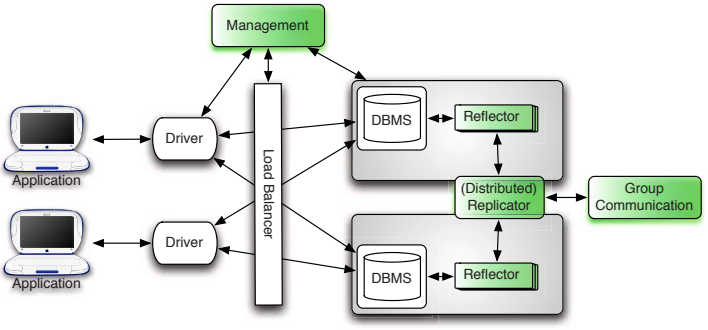
- *Management tools,* which are able to control the Driver and DBMS components independently from the Application using a mixture of standard and proprietary interfaces.
- The *Reflector* is attached to each DBMS and allows inspection and modification of on-going transaction processing.
- The *Replicator* mediates the coordination among multiple reflectors in order to enforce the desired consistency criteria on the replicated database. This is a distributed component that communicates using the group communication component.
- The *Group Communication* supports the communication and coordination of local replicators.

An important component of the architecture is the interface among the building blocks, which allows them to be reused in different contexts. The interfaces exposed by the reflector and group communication service are detailed in Sections 13.3 and 13.4 respectively. To support as much as possible off-the-shelf and third party tools, the call-level and SQL interfaces, and the remote database access protocol adhere to existing standards. For instance, the architecture can be easily mapped to a Java system, using JDBC as the call-level interface and driver specification, any remote database access protocol encapsulated by the driver and a DBMS, and an external configuration tool for the JDBC driver.

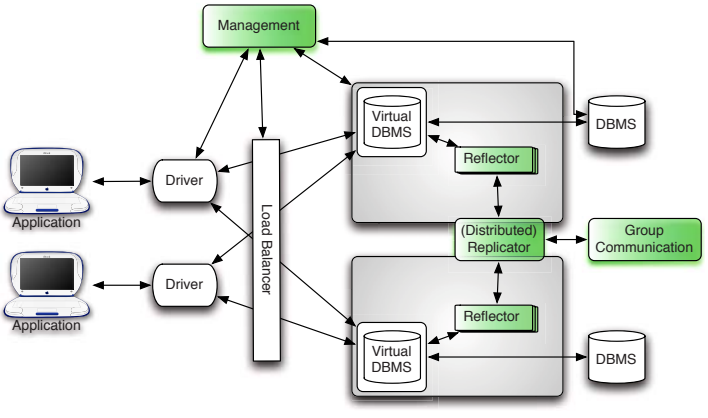
The generic architecture can be instantiated in several ways, for example, multiple logical components can be provided by multiple or by a single physical component. Figure 13.2 illustrates three relevant instantiations of the architecture.

The first instantiation, illustrated in Figure 2(a), is denoted as the in-core variant. In this case, the reflector is provided within the same physical component as the DBMS, where replication and communication components can be installed to control replication. Typically, such a variant is possible when the DBMS is augmented with replication support. Examples of protocols that need this support are [24, 31].

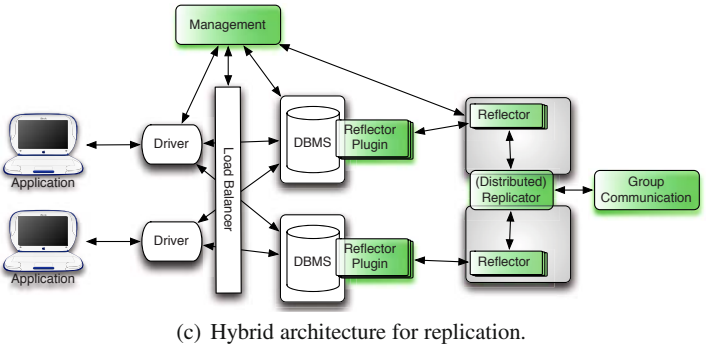
The second instantiation, illustrated in Figure 2(b), is denoted as middleware variant. In this scenario, clients connect to a virtual DBMS which implements the



(a) In-core architecture for replication.



(b) Middleware architecture for replication.



(c) Hybrid architecture for replication.

Fig. 13.2 Different instantiations of the generic architecture.

reflector interface. The virtual DBMS itself is implemented using the client interfaces provided by the real DBMS. The work presented in [26] exploits this approach. A commercial product that also implements this approach is Sequoia [11].

A hybrid approach can also be achieved by adding to the database the necessary hooks to export information about ongoing transactions by means of a reflector plugin. This plug-in interacts with the distributed replication protocol which runs on a different process. This solution is depicted in Figure 2(c).

13.3 Reflector: Replication-Friendly Database Support

A key component in the architecture is the reflector. The purpose of this component is to export a replication-friendly database interface to the replicator (described later in Section 13.5). In this way, the database replication protocols can be implemented independently of the specific DBMS system being used at deployment time, thus promoting the design and implementation of database replication protocols that can be used in a wide range of configurations.

The independence between a specific DBMS system and the replication protocols is achieved by augmenting the standard database interfaces with additional primitives that provide abstractions that reflect the usual processing stages of transactions (e.g. transaction parsing, optimization and execution) inside the DBMS engine. Naturally, the implementation details of the replicator vary depending on the specific DBMS instance and the architecture chosen. In this section we outline the replicator interface and the rationale for its design.

13.3.1 Reflection for Replication

A well known software engineering approach to building systems with complex requirements is reflection [27, 25]. By exposing an abstract representation of the systems' inner functionality, the later can be inspected and manipulated, thus changing its behavior without loss of encapsulation. DBMS have long taken advantage of this, namely, on the database schema, on triggers, and when exposing the log.

Logging, debugging and tracing facilities are some examples of important addons to DBMS that are today widely available. The computation performed by such plug-ins is known as a computational reflection, and the systems that provide them are known as reflective systems. Specifically, a reflective system can be defined as a system that can reason about its computation and change it. Reflective architectures ease and smooth the development of systems by encapsulating functionality that is not directly related to the application domains. This can be done to a certain extent in an ad-hoc manner, by defining hooks in specific points of a system, or with support from a programming language. In both cases, there is a need for providing a reflective architecture where the interaction between a system (i.e. base-level objects) and its reflective counterpart is done by a meta-level object protocol and the reflective computation is performed by meta-level objects. These objects exhibit a meta-level programming interface.

Previous reflective interfaces for database management systems were mainly targeted at application programmers using the relational model. Their domain is therefore the relational model itself. Using this model, one can intercept operations that modify relations by inserting, updating, or deleting tuples, observe the tuples being changed and then enforce referential integrity by vetoing the operation (all at the meta-level) or by issuing additional relational operations (base-level).

A reflection mechanism for database replication was also recently proposed in [42]. In contrast to the approach described in this section, it assumes that reflection is achieved by wrapping the DBMS server and intercepting requests as they are issued by clients. By implementing reflection this way, one can only reflect computation at the first stage (statements), i.e. with coarse granularity. Exposing further details requires rewriting large portions of DBMS functionality at the wrapper level. As an example, Sequoia [11] does additional parsing and scheduling stages at the middleware level. In theory, this approach could be more generic and suitable to reflect black-box DBMSs. In practice, this is not the case, since DBMS do not offer the exact same interface. Therefore, the wrapper must be customized for each DBMS. Moreover, this approach can introduce significant latency by requiring extra communication steps and/or extra processing of requests.

Furthermore, some protocols are concerned with details that are not visible in the relational model, such as modifying query text to remove non-deterministic statements, for instance, those involving `NOW()` and `RANDOM()`. Also, one may be interested in intercepting a statement as it is submitted, whose text can be inspected, modified (meta-level) and then re-executed, locally or remotely, within some transactional context (base-level).

Therefore, a target domain more expressive than the relational model is required. We propose to expose a transaction object that reflects a series of activities (e.g. parsing) that is taking place on behalf of a transaction. This object can be used to inspect the transaction (e.g. wait for it to commit) or to act on it (e.g. force a rollback). Using the transaction object the meta-level code can register to be notified when specific events occur. For instance, when a transaction commits, a notification is issued and contains a reference to the corresponding transaction object (meta-object). Actually, handling notifications is the way that meta-level code dynamically acquires references to meta-objects describing the on-going computation.

13.3.2 Processing Stages

The reflector interface abstracts transaction processing as a pipeline [17]. This is illustrated in Figure 13.3. The replicator acts as a plug-in that registers itself to receive notifications of each stage of the pipeline. The notifications are issued by the reflector as meta-objects, where each meta-object represents one processing stage. The processing stages are briefly described below. The replicator is notified of these processing stages in the order they are listed bellow:

- The *Parsing stage* parses the received raw statements and produces a parse tree;
- The *Optimization stage* transforms the parse tree according to various optimization criteria, heuristics and statistics to an execution plan;

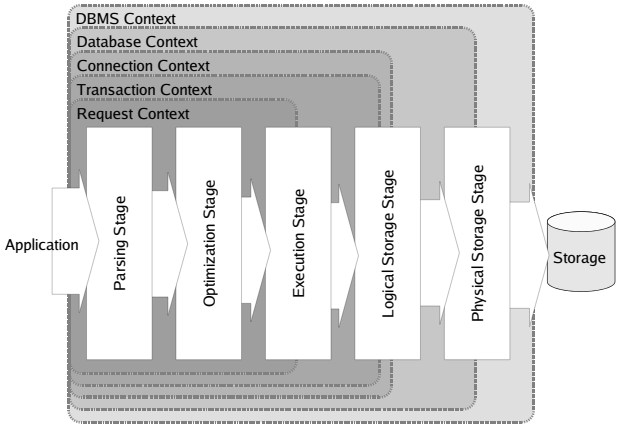


Fig. 13.3 Major meta-level interfaces: processing stages and contexts.

- The *Execution stage* executes the plan and produces object-sets (data read and written);
- The *Logical Storage stage* deals with mapping from logical objects to physical storage;
- The *Physical Storage stage* deals with block input/output and synchronization.

In general, the reflector will issue notifications at the meta-level (to the registered replicator) whenever computation proceeds from one stage to the next. For instance, if a replication protocol needs to ensure that all the requests are deterministic, it needs to be notified on the Parsing stage to modify the initial statement and remove non determinism; when the computation reaches the Execution stage, it will produce a set of read and written data that is reflected, issuing a notification. The interface thus exposes meta-objects for each stage and for data that moves through them.

13.3.3 Processing Contexts

The meta-interface exposed by the processing pipeline is complemented by nested context meta-objects, also shown in Figure 13.3. These context meta-objects show on behalf of whom some operation is being performed. In detail, the *DBMS* and *Database* context interfaces expose meta-data and allow notification of life-cycle events. *Connection* contexts reflect existing client connections to databases. They can be used to retrieve connection specific information, such as user authentication or the character set encoding used. The *Transaction* context is used to notify events related to a transaction such as its startup, commit or rollback. Synchronous event handlers available here are key to consistent replication protocols. Finally, to ease the manipulation of the requests within a connection to a database and the corresponding transactions one may use the *Request* context interface.

Events fired by processing stages refer to the directly enclosing context. Each context has then a reference to the next enclosing context and can enumerate all en-

closed contexts. This allows, for instance, to determine all connections to a database or the current active transaction in a specific connection. Some contexts are not valid at the lowest abstraction levels. Namely, it is not possible to determine on behalf of which transaction a specific disk block is being flushed by the physical stage.

Furthermore, replication protocols can attach an arbitrary object to each context. This allows context information to be extended as required by each replication protocol. As an example, when handling an event fired by the first stage of the pipeline signaling the arrival of a statement in textual format the replication protocol gets a reference to the enclosing transaction context. It can then attach additional information to that context. Later, when handling an event signaling the availability of the transaction outcome, the replication protocol follows the reference to the same transaction context to retrieve the information previously attached.

13.3.4 Base-Level and Meta-level Calls

An advantage of reflection is that base- and meta-level code can be freely mixed, as there is no inherent difference between base- and meta-objects. For instance, a direct call to meta-level code can be forced by the application programmer by registering it as a native procedure and then using the CALL SQL statement. This causes a call to the meta-level code to be issued from the base-level code within the *Execute* stage. The target procedure can then retrieve a pointer to the enclosing *Request* context and thus to all relevant meta-interfaces. Meta-level code can callback into base level in two different situations. The first is within a direct call from base-level to issue statements in an existing enclosing request context. The second option is to use the enclosing *Database* context to open a new base-level connection to the database.

A second issue when considering base-level calls is whether these also get reflected. The proposed interface allows to disable reflection on a case-by-case basis by invoking an operation on context meta-objects. Therefore, meta-level code can disable reflection for a given request, a transaction, a specific connection or even an entire database. Actually this can be used on any context meta-object and thus for performance optimization. For example, consider a replication protocol, which is notified that a connection will only issue read-only operations, and thus ceases monitoring them.

A third issue is how base-level calls issued by meta-level code interact with regular transaction processing regarding concurrency control. Namely, how conflicts that require rollback are resolved in multi-version concurrency control where the first committer wins or, more generally, when resolving deadlocks. The proposed interface solves this by ensuring that transactions issued by the meta-level do not abort in face of conflicts with regular base-level transactions. Given that replication code running at the meta-level has a precise control on which base-level transactions are scheduled, and thus can prevent conflicts among those, has been sufficient to solve all considered use cases. The implementation of this simple solution resulted in a small set of localized changes within the DBMS.

13.3.5 Exception Handling

The DBMS handles most of the base-level exceptions by aborting the affected transaction and generating an error to the application. The proposed architecture does not change this behavior. Furthermore, the meta-level is notified by an event issued by the transaction context object; this allows meta-level to cleanup after an exception has occurred.

Most exceptions within a transaction context that are not handled at the meta-level can be resolved by aborting the transaction. However, some event handlers should not raise exceptions to avoid inconsistent information on databases or recursive exceptions, namely, while starting up or shutting down a database, while rolling back or after committing a transaction. In these cases, any exception will leave the database in a panic mode requiring manual intervention to repair the system. Furthermore, interactions between the meta-level and base-level are forbidden and any attempt of doing so, puts the database in panic mode.

Exceptions from meta-level to base-level calls need additional management. For instance, while a transaction is committing, meta-level code might need to execute additional statements to keep track of custom meta-information on the transaction before proceeding, and this action might cause errors due to deadlock problems or low amount of resources. Such cases are handled as meta-level errors to avoid disseminating errors inside the database while executing the base-level code.

13.3.6 Existing Reflector Bindings

In this section we discuss how the reflector interface was implemented in three different systems, namely, Apache Derby, PostgreSQL, and Sequoia. These systems represent different tradeoffs and implementation decisions and are thus representative of what one should expect when implementing the architecture proposed in this chapter.

Apache Derby Binding Apache Derby [3] is a fully-featured database management system with a small footprint developed by the Apache Foundation and distributed under an open source license. It is also distributed as IBM Cloudscape and in Sun JDK 1.6 as JavaDB. It can either be embedded in applications or run as a standalone server. It uses locking to provide serializability. The initial implementation of the Reflection interface takes advantage of Derby being natively implemented in Java to load meta-level components within the same JVM and thus closely coupled with the base-level components. Furthermore, Derby uses a different thread to service each client connection, thus making it possible for notifications to the meta-level to be done by the same thread and thus reduced to a method invocation, which has negligible overhead. This is therefore the preferred implementation scenario. The current implementation exposes all context objects and the parsing and execution stages, as well as calling between base-level and meta-level as described in Section 13.3.4.

PostgreSQL Binding PostgreSQL [39] is also a fully-featured database management system distributed under an open source license. It has been ported to multiple operating systems, and is included in most Linux distributions as well as in recent versions of Solaris. Commercial support and numerous third party add-ons are available from multiple vendors. It currently provides a multi-version concurrency control mechanism supporting snapshot isolation. The major issue in implementing the interface is the mismatch between its concurrency model and the multi-threaded meta-level runtime. PostgreSQL uses multiple single-threaded operating system processes for concurrency. This is masked by using the existing PL/J binding to Java, which uses a single standalone Java virtual machine and inter-process communication. This imposes an inter-process remote procedure call overhead on all communication between base and meta-level. Furthermore, the implementation of the reflector interface in PostgreSQL uses a hybrid approach. Instead of directly coding the reflector interface on the server, key functionality is added to existing client interfaces and as loadable modules. The meta-level interface is then built on these. The two-layer approach avoids introducing a large number of additional dependencies in the PostgreSQL code, most notably in the Java virtual machine. As an example, transaction events are obtained by implementing triggers on transaction begin and end statements. A loadable module is then provided to route such events to meta-objects in the external PL/J server. The current implementation exposes all context objects and the parsing and execution objects, as well as calling between base-level and meta-level as described in Section 13.3.4. It avoids base-level operations blocking meta-level operations simply by modifying the choice of the transactions to be terminated upon deadlock detection and write conflicts.

Sequoia Binding Sequoia [11] is a middleware system for database clustering built as a server wrapper. It is primarily targeted at obtaining replication or partitioning by configuring the controller with multiple backends, as well as improving availability by using several interconnected controllers. Nevertheless, when configured with a single controller and a single backend, Sequoia provides a state-of-the-art JDBC interceptor. It works by creating a virtual database at the middleware level, which reimplements part of the abstract transaction processing pipeline and delegates the rest to the backend database. The current implementation exposes all context, parsing and execution objects, as well as calling from meta-level to base-level with a separate connection. It does not allow calling from base-level to meta-level, as execution runs in a separate process. It can however be implemented by directly intercepting such statements at the parsing stage. It neither avoids base-level operations interfering with meta-level operations, and this cannot be implemented as described in the previous sections as one does not modify the backend DBMS. It is however possible to the clustering scheduler already present in Sequoia to avoid concurrently scheduling base-level and meta-level operations to the backend, thus precluding conflicts. This implementation is of great interest when with a closed source DBMS that does not natively implement reflector interfaces.

13.4 GCS: Communication and Coordination Support

All database replica consistency protocols require communication and coordination support. Among the most relevant abstractions to support database replication we may identify: reliable multicast (to disseminate updates among the replicas), total order (to define a global serial order for transactions) and group membership (to manage the set of currently active replicas in the system).

A software package that offers this sort of communication and coordination support is typically bundled in a package called a *Group Communication Toolkit*. After the pioneer work initiated two decades ago with Isis [8], many other toolkits have been developed. Appia [28], Spread [2], and JGroups [5] are, among others, some of the group communication toolkits in use today. Therefore, group communication is a mature technology that greatly eases the development of practical database replication systems.

At the same time, group communication is still a hot research topic, as performance improvements and wider applicability are sought [47, 43, 33, 35, 34]. Furthermore, group communication is clearly an area where there is no one solution that fits all application scenarios. For instance, just to offer total order multicast, dozens of different algorithms have been proposed [15], each outperforming the others for a specific setting: there are protocols that perform better for heavily loaded replicas in switched local area networks [18], others for burst traffic in LANs [22], others for heterogeneous wide-area networks [40], etc. More details about the primitives offered by a group communication toolkit can be found in Chapter 3 and Chapter 6.

Therefore, having a clear interface between the replication protocols and the GCS has multiple practical advantages. To start with, it allows to tune the communication support (for instance, by selecting the most appropriate total order protocol) without affecting the replication protocol. Furthermore, given that different group communication toolkits implement different protocols, it should be possible to re-use the same replication protocols with different group communication toolkits.

To address these problems we have defined a generic interface to group communication services that may be used to wrap multiple toolkits. The interface, called *Group Communication Service for Java*, or simply jGCS, has been designed for the Java programming language and leverages several design patterns that have recently become common ground of Java-based middleware. The interface specifies not only the API but also the (minimum) semantics that allow application portability. jGCS owns a number of novel features that makes it quite distinct from previous attempts to define standard group communication interfaces, namely:

- jGCS aggregates the service in several complementary interfaces, as depicted in Figure 13.4, namely a set of *configuration interfaces* (namely, GroupConfiguration, ProtocolFactory and Service), a *message passing interface* (Data), and a set of *membership interfaces* (Control). The configuration interface specifies several opaque configuration objects that encapsulate specifications of message delivery guarantees. These are to be constructed in an implementation dependent manner to match application requirements and then supplied using some dependency injection technique. The message passing interface exposes a straightforward in-

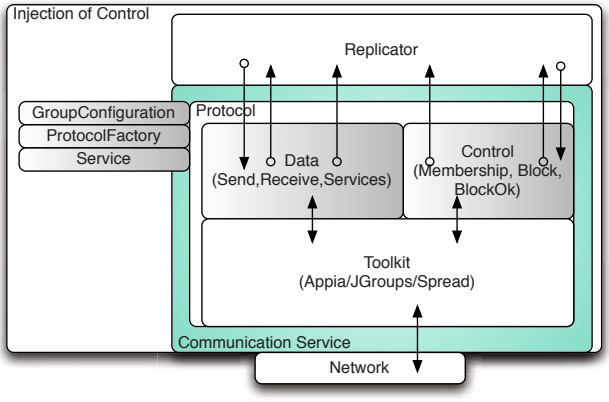


Fig. 13.4 Components of the GCS.

terface to sending and receiving byte sequences, although concerned with high throughput, low latency and sustainable concurrency models in large scale applications. Finally, a set of membership interfaces expose different membership management concepts as different interfaces, that the application might support or need.

- jGCS provides support for recent research results that improve the performance of group communication systems, namely, *semantic annotations* [34, 35, 33] and *early delivery* [32, 45, 43, 41].
- the interface introduces negligible overhead, even when jGCS is implemented as wrapper layer and is not supported natively by the underlying toolkit.

13.4.1 Architectural and Algorithmic Issues

In this section we discuss the main features that must be provided by the group communication toolkit to cope with the requirements needed by database replication protocols. As proof-of-concept, we implemented the presented features in the Appia group communication toolkit.

Optimistic Uniform Total Order The notion of optimistic total order was first proposed in the context of local-area broadcast networks [32]. In many of such networks, the spontaneous order of message reception is the same in all processes. Moreover, in sequencer-based total order protocols the total order is usually determined by the spontaneous order of message reception in the sequencer process. Based on these two observations a process may estimate the final total order of messages based on its local receiving order and, therefore, provide an optimistic delivery as soon as a message is received from the network. With this optimistic delivery, the application can make some progress. For example, a database replication protocol can apply the changes in the local database without committing it. The

commit procedure can only be made when the final order is known and if it matches the optimistic order. If the probability of the optimistic order matching the final order is very high, the latency window of the protocol is reduced and the system gains in performance.

Unfortunately, spontaneous total order does not occur in wide-area networks. The long latency in wide-area links causes different processes to receive the same message at different points in time. Consider a simple network configuration with three nodes a, b, and s such that network delay between nodes a and b is $2ms$, and network delays to and from node s are $12ms$. Assume that process a multicasts a message m_1 and that, at the same time, the sequencer process s multicasts a message m_2. Clearly, the sequencer will receive m_2 before m_1, given that m_1 would require $12ms$ to reach the sequencer. On the other hand, process b will receive m_1 before m_2, as m_1 will take only $2ms$ to reach b while m_2 will require $12ms$. From this example, it should be obvious that the spontaneous total order provided by the network at b is not a good estimate of the observed order at the sequencer.

To address the problem above, a system can be configured to use a total order protocol such as SETO [29]. SETO is a generalization of the optimistic total order protocol proposed in [43] and operates by introducing artificial delays in the message reception to compensate for the differences in the network delays. It is easier to describe the intuition of the protocol by using a concrete example. Still considering the above simple network configuration, assume also that we are able to provide to each process an estimate of the network topology and of the delays associated with each link. In this case, b could infer that message m_1 would take $10ms$ more to reach s than to reach b. By adding a delay of $10ms$ to all messages received from a, it would mimic the reception order of a's messages at s. A similar reasoning could be applied to messages from other processes.

When configured to use this protocol, the group communication toolkit delivers the original message as soon as it is received (network order). Notifications about optimistic total order and final uniform total order are later delivered, indicating that progress can be done regarding a particular message.

Primary Partition Support Partitions in the replica group may happen due to failures in the cluster (network, switching hardware, among others). In asynchronous systems, virtual partitions (indistinguishable from physical partitions) may happen due to unexpected delays. A partitionable group membership service allows multiple concurrent views of the group, each corresponding to a different partition, to co-exist and evolve in parallel [4, 16]. In the context of database replication, this is often undesirable as it may lead to different replicas processing and committing conflicting updates in a uncoordinated form. A partition in the group membership can then easily lead to the *split-brain* phenomenon: the state in different replicas diverges and is no longer consistent. In contrast, a primary-partition group membership service maintains a single agreed view of the group at any given time, delivering a totally ordered sequence of views (processes that become disconnected from the primary partition block or are forced to crash and later rejoin the system).

In our implementation, primary partitions are defined by majority quorums. The initial composition of the primary partition is defined at configuration time, using standard management interfaces. The system remains alive as long as a majority of the previous primary partition remains reachable [23, 6]. The dynamic update of the primary partition is coordinated and has to be committed by a majority of members of the previous primary. This is deterministic and ensures that only one partition exists at a time. Using this mechanism, a replica that belongs to a primary partition can move to a non-primary partition when a view changes. In this case, the replication protocol only gets notified that the group has blocked and does not receive any view while it is not reintegrated in a primary partition.

13.4.2 Existing GCS Bindings

Open source implementations of jGCS for several major group communication systems have been already developed, namely, Appia [28], Spread [2] (including the FlushSpread variant), and JGroups [5]. All these bindings are open source and available on SourceForge.net.[1] Besides making jGCS outright useful in practice, these validate that the interface is indeed generic. These implementations are described in the following paragraphs.

Appia Binding Appia [28] is a layered communication support framework that was implemented in the University of Lisbon. It is implemented in Java and aims at high flexibility to build communication channels that fit exactly in the user needs. More details about Appia are described in Section 13.4.1.

The implementation of GCS is built directly on Appia's protocol composition interfaces as an additional layer. GCS configuration objects thus define the micro-protocols that will be used in the communication channels. Each service identifies an Appia channel and messages are sent through the channel that fits the requested service. As Appia supports early delivery in totally ordered multicast, this is exposed in the GCS binding using the ServiceListener interface. Appia implements all extensions of the ControlSession, depending on the channel configuration.

JGroups Binding JGroups [5] is a group communication toolkit modeled on Ensemble [19] and implemented in Java. It provides a stack architecture that allows users to put together custom stacks for different view synchronous multicast guarantees as well as supporting peer groups. It provides an extensive library of ordering and reliability protocols, as well as support for encryption and multiple transport options. It is currently used by several large middleware platforms such as JBoss and JOnAS.

The JGroups implementation of GCS also uses the configuration interface to define the micro-protocols that will be used in the communication channel. JGroups can provide only one service by the applications, since configurations only support

[1] GCS and its bindings are available in `http://jgcs.sf.net`

one JGroups channel per group communication instance. JGroups implements all extensions of the ControlSession.

Spread Binding Spread/FlushSpread [2] is a toolkit implemented by researchers of the Johns Hopkins University. It is based on an overlay network that provides a messaging service resilient to faults across local- and wide-area networks. It provides services ranging from reliable message passing to fully ordered messages with delivery guarantees. 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. Although there are interfaces for Spread in multiple languages, these do not support the FlushSpread extension, which provides additional guarantees with a different interface.

The Spread and FlushSpread implementations of GCS use the configuration interface to define the location of the daemon and the group name. The implementation to use (FlushSpread or just Spread) is also defined at configuration time. In Spread, the quality of service is explicitly requested for each message, being thus encapsulated in Service configuration objects.

13.5 Replicator: Pluggable Replication Protocols

The replicator is a distributed component responsible for coordinating the interaction among all DBMS replicas in order to enforce the consistency of the replicated database. It directly interfaces with the reflector and relies on the GCS module for all communication and replica membership control, as shown in the Figure 13.5.

It is within the replicator that the replica consistency protocols are implemented. The module is built around four process abstractions that are able to express most,

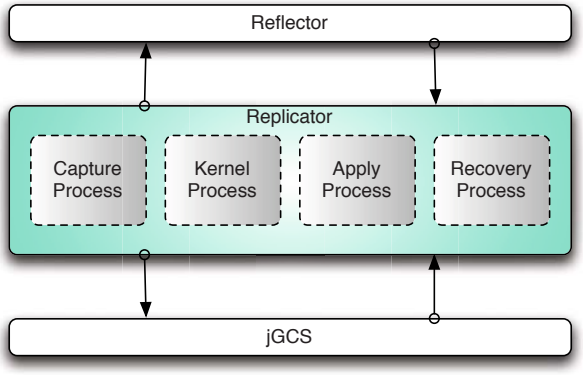


Fig. 13.5 Replicator architecture.

if not all, database replication protocols. These are the Capture, Kernel, Apply and Recovery processes and are described next.

Capture Process The capture process is the main consumer of the reflector events. It receives events from the DBMS, converts them to appropriate events within the replicator and notifies the other processes. In particular, it receives a transaction begin request and registers the current transaction context. For instance, for update transactions, the capture process may instruct the reflector to receive the write and read sets of the transaction when the commit request is performed. Using this information, it may construct an internal transaction event that carries the transaction identification along with the corresponding read and write sets. It then notifies the kernel process which, in turn, is responsible for distributing the transaction data and enforcing the consistency criterion.

Kernel Process This process implements the core of the replica consistency protocol. In general, it handles the replication of local transactions by distributing relevant data and determining their global commit order. Additionally, it handles incoming data from remotely executed transactions. The local outcome of every transaction is ultimately decided by the kernel process, in order to ensure a target global consistency criterion. To execute its task, the kernel process exchanges notifications with the capture and apply processes, and interfaces directly the GCS component.

Apply Process The apply process is responsible for efficiently injecting incoming transaction updates into the local database through the reflector component. To achieve optimum performance, this implies executing multiple apply transactions concurrently and, when possible, batching updates to reduce the number of transactions. This needs however to ensure that the agreed serialization order is maintained.

Recovery Process The recovery process intervenes whenever a replica joins or rejoins the group. It is responsible for exporting the database state when acting as a donor or to bring the local replica up-to-date if recovering.

Both the recovery and the kernel modules cooperate closely with the GCS module. To allow the integration of the new replica into the group, the kernel module is required to temporarily block any outgoing messages until the complete recovery of the new replica is notified by the recovery process.

13.6 Consistent Database Replication

In this section we consider a representative set of database replication protocols providing strong replica consistency and elaborate on their suitability to handle demanding workloads (see Chapter 1 for more details about consistency models for replication). We start by analyzing each protocol with respect to its contention path and concurrency restrictions. Then we compare their performance using a common

test-bed, implemented as plug-ins for the replicator component of our architecture, using the industry standard TPC-C benchmark and workload.

Database replication protocols differ greatly in whether transactions are executed optimistically [31, 24] or conservatively [37]. In the former, a transaction is executed by any replica without a priori coordination with other replicas. It is just before committing that replicas coordinate and check for conflicts between concurrently executed transactions. Transactions that would locally commit may end up aborting due to conflicts with remote concurrent transactions. On the contrary, in the conservative approach, all replicas first agree on the execution order for potentially conflicting transactions ensuring that when a transaction executes there is no conflicting transaction being executed remotely and therefore its success depends entirely on the local database engine. Generally, two transactions conflict if both access the same *conflict class* (e.g. table) and one of them updates it.

As expected, both approaches have their virtues and problems [21]. The optimistic execution presents very low contention and offers high concurrency levels. However, it may yield concurrency-induced aborts which, occasionally, may impair the protocol's fairness since long-running transactions may experiment unacceptable abort rates. On the contrary, the conservative approach does not lead to aborts and offers the same committing opportunities to all transaction types. The resulting degree of concurrency heavily depends on the granularity of the defined conflict classes. Fine conflict classes usually require application-specific knowledge and any labeling mistake can lead to inconsistencies.

Another crucial aspect of database replication protocols is whether replication is active or passive. With active replication each transaction executes at all replicas while with passive protocols only a designated replica actually executes the transaction and the state updates are then propagated to the other replicas. Active replication is required for structural or system wide requests, such as the creation of tables and users, and desired for update intensive transactions. The passive approach is otherwise preferable, as it confines the processing to a single replica, is insensible to non-deterministic requests, and allows for more concurrency.

In the following sections we discuss and compare five consistent database replication protocols: a conservative and two optimistic passive replication approaches, an active replication protocol (inherently conservative regarding transactions execution), and a hybrid solution that combines both conservative and optimistic execution as well as active and passive replication. In all cases we consider a common practice that only update transactions are handled by the replication protocols. Queries are simply executed locally at the database to which they are submitted and do not require any distributed coordination. The discussion on the impact of this configuration in the overall consistency criterion has been discussed elsewhere [30].

Our analysis is focused on dynamic aspects, namely on the queuing that happens in different parts of the system and on the amount of concurrency that can be achieved. Then, we contrast the original assumptions underlying the design of the protocols with our experience with the actual implementations using the TPC-C workload [21].

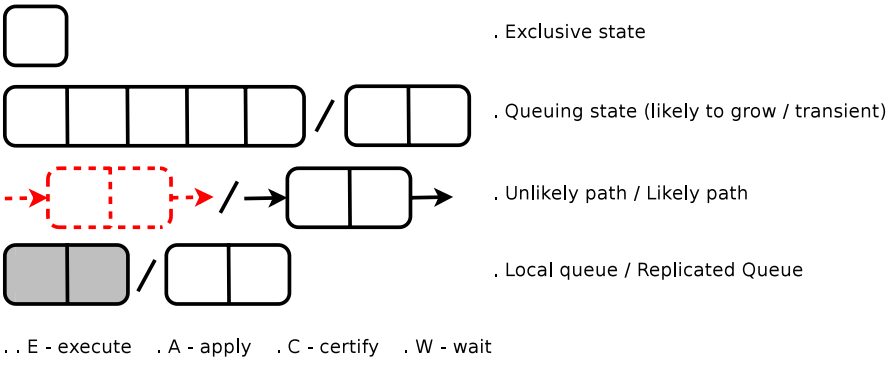


Fig. 13.6 Notation.

Figure 13.6 introduces the notation used to represent the state maintained by the protocol state-machines. Given the emphasis on dynamic aspects, we use different symbols for states that represent queuing and for states in which at most a single non-conflicting transaction can be at any given time. We show also which queues are likely to grow when the system is congested. When alternative paths exist, due to optimistic execution, we show which is the more likely to be executed. We make a distinction between local and replicated queues and identify relevant actions: execute, apply, certify, and wait.

At the core of all these protocols is an atomic (or total ordered) multicast. For all of them we use a consistent naming for queues according to the use of the *atomic mcast* primitive. Queue Q0 is before the atomic mcast, Q1 is between the atomic mcast and its delivery, and Q2 is after the delivery.

Some of the discussed algorithms [36, 31] have been originaly proposed using atomic primitives with optimistic delivery. The goal is to compensate the inherent ordering latency by allowing tentative processing in parallel with the ordering protocol. If the final order of the messages matches the predicted order then the replication protocol can proceed, otherwise the results obtained tentatively are discarded. Protocols with this optimistic assumption use messages in Q1. Queue Q1 has messages with tentative order. In contrast, messages in Q2 have a final order.

13.6.1 Replication with Conservative Execution

We consider the Non-disjoint Conflict Classes and Optimistic Multicast (NODO) protocol [36] as an example of the conservative execution. In NODO data is a priori partitioned in conflict classes, not necessarily disjoint. Each transaction has an associated set of conflict classes (the data partitions it accesses) which are assumed to be known in advance. In practice, this requires the entire transaction to be known before it is executed, precluding the processing of interactive transactions.

NODO's execution is depicted in Figures 13.7 and 13.8. The former shows exchanged messages and synchronization points whereas the second focuses on its

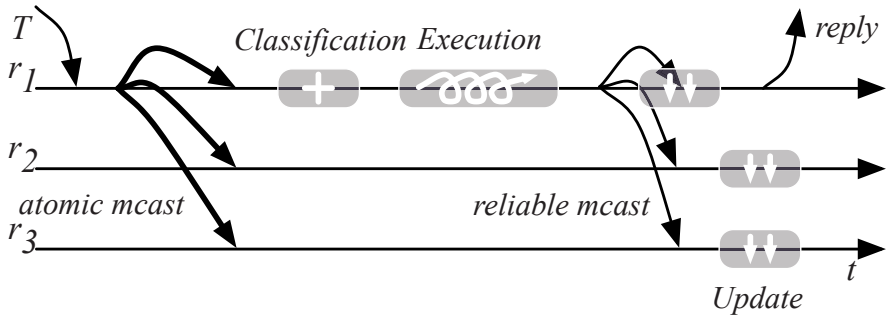


Fig. 13.7 Conservative execution: NODO.

dynamic aspects. When a transaction is submitted, its identifier (id) and conflict classes are atomically multicast to all replicas obtaining a total order position. Each replica has a queue associated with each conflict class and, once delivered, a transaction is classified according to its conflict classes and enqueued in all corresponding queues. As soon as a transaction reaches the head of all of its conflict class queues it is executed. In this approach, a transaction is only executed by the replica to which it was originally submitted.

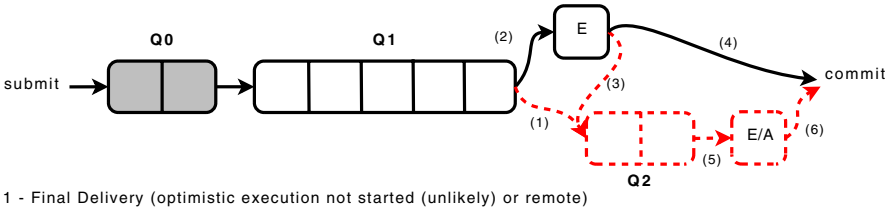
Clearly, the definition of the conflict classes has a direct impact on performance. The fewer the number of transactions with overlapping conflict classes, the better the interleave among transactions. Conflict classes are usually defined at the table level but can have a finer grain at the expense of a non-trivial validation process to guarantee that a transaction does not access conflict classes that were not previously specified.

When the commit request is received, the outcome of the transaction is reliably multicast to all replicas along with the replica's updates (write-set) and a reply is sent to the client. Each replica applies the remote transaction's updates with the parallelism allowed by the initially established total order of the transaction.

The protocol ensures 1-copy serializability [7] as long as transactions are classified taking into account read/write conflicts. To achieve 1-copy snapshot isolation [26] transactions must be classified taking into account just write/write conflicts.

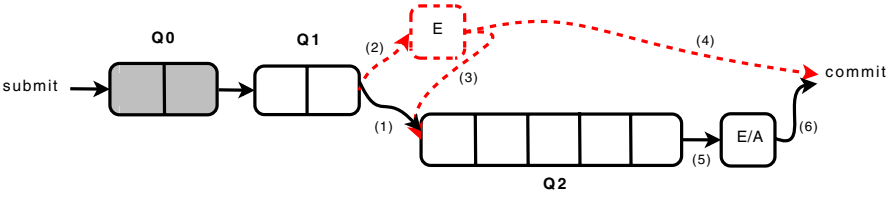
A transaction is scheduled optimistically if there is no conflicting transaction already ordered (Q2). This tentative execution may be done at the expense of an abort if a concurrent transaction is later on ordered before it.

Figure 8(a) shows the states that a transaction goes through upon being submitted by a client. Assuming that group communication is the bottleneck, the time spent in the queue waiting for total order (Q1) is significant enough compared to the time taken to actually execute such that it is worthwhile to optimistically execute transactions (transition 2 instead of transition 1). This makes it possible that when a transaction is finally ordered, it is immediately committed (transition 4). Assuming that the tentative optimistic ordering is correct, a rollback (transition 3) is unlikely.



1 - Final Delivery (optimistic execution not started (unlikely) or remote)
2 - Submit transaction to optimistic execution
3 - Final Delivery (missed order and then rollback (unlikely))
4 - Final Delivery (correct order)
5 - Submit transaction to execution in order
6 - Execution finished in order

(a) Assuming that atomic mcast is the bottleneck.



1 - Final Delivery (optimistic execution not started or remote)
2 - Submit transaction to optimistic execution (unlikely)
3 - Final Delivery (missed order and rollback (unlikely, but it doesn't matter))
4 - Final Delivery (correct order)
5 - Submit transaction in order
6 - Execution finished in order

(b) Assuming that transaction execution is the bottleneck.

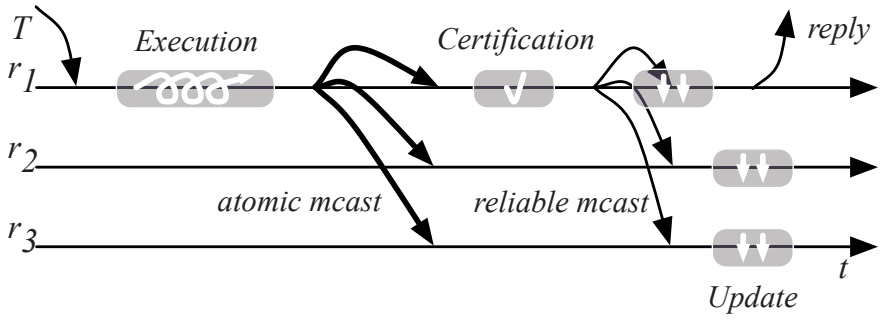
Fig. 13.8 States, transitions, and queues in NODO.

On the other hand, if the transaction execution is the bottleneck, then queuing will happen in queue Q2 and not in queue Q1. Thus if Q2 is never empty, then no transaction in queue Q1 is eligible for optimistic execution. This scenario is depicted in Figure 8(b): The optimistic path is seldom used and the protocol boils down to a coarse-grained distributed locking approach, which has a very large impact on scalability. Notice that if there are k (disjoint) conflict classes, there can be at most k transactions executing in the whole system.

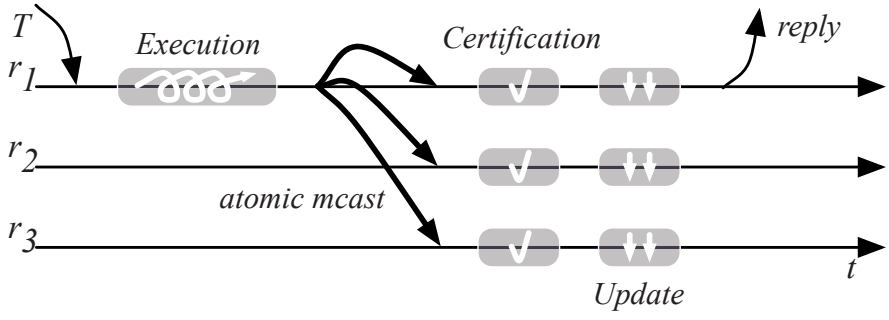
Experiments using the TPC-C workload show that in a local area network, group communication is not the bottleneck. Figure 13.13 shows the NODO protocol saturating when there are still plenty of system resources available.

13.6.2 Replication with Optimistic Execution

To illustrate the optimistic execution approach we consider two protocols: Postgres-R (PGR) [24] and Database State Machine (DBSM) [31]. In both protocols, transactions are immediately executed by the replicas to which they are submitted without



(a) Optimistic Execution:PGR.



(b) Optimistic Execution:DBSM.

Fig. 13.9 Optimistic executions: PGR and DBSM.

any prior global coordination. Locally, transactions are synchronized according to the specific concurrency control mechanism of the database engine.

The messages exchanged and the synchronization points of the execution of these protocols are depicted in Figure 13.9. The dynamic aspects are depicted in Figures 13.10 (PGR) and 13.11 (DBSM). Upon receiving a commit request, a successful transaction is not readily committed. Instead, its changes (write-set) and read data (read-set) are gathered and a termination protocol initiated. The goal of the termination protocol is to decide the order and the outcome of the transaction such that a global correctness criterion is satisfied (e.g. 1-copy serializability [7] or 1-copy snapshot isolation [26]). This is achieved by establishing a total order position for the transaction and certifying it against concurrently executed transactions. The certification of a transaction is done by evaluating the intersection of its read- and write-set (or just write-set in case of the snapshot isolation) with the write-set of concurrent, previously ordered transactions. The fate of a transaction is therefore determined by the termination protocol and a transaction that would locally commit may end up aborted.

These protocols differ on the termination procedure. Considering 1-copy serializability, both protocols use the transaction's read-set in the certification procedure. In PGR, the transaction's read-set is not propagated and thus only the replica exe-

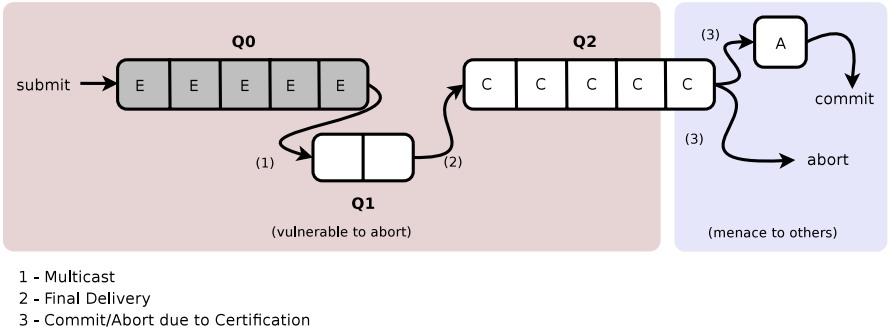


Fig. 13.10 States, transitions, and queues in PGR.

cuting the transaction is able to certify it. In DBSM, the transaction's read-set is also propagated allowing each replica to autonomously certify the transaction.

In detail, upon the reception of the commit request for a transaction t, in PGR the executing replica atomically multicasts t's id and t's write-set. As soon as all transactions ordered before t are processed, the executing replica certifies t and reliably multicasts the outcome to all replicas. The certification procedure consists in checking t's read-set and write-set against the write-sets of all transactions ordered before t. The executing replica then commits or aborts t locally and replies to the client. Upon the reception of t's commit outcome each replica applies t's changes through the execution of a high priority transaction consisting of updates, inserts and deletes according to t's previously multicast write-set. The high priority of the transaction means that it must be assured of acquiring all required write locks, possibly aborting any locally executing transactions.

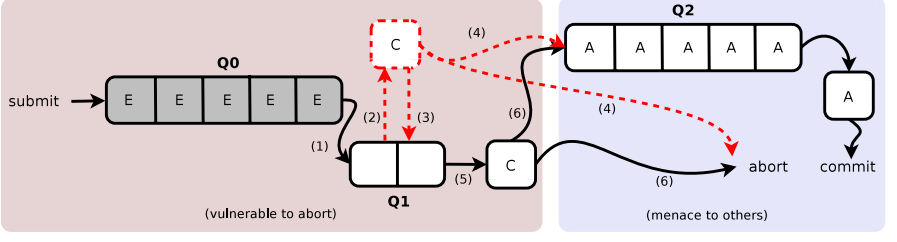
The termination protocol in the DBSM is significantly different and works as follows. Upon the reception of the commit request for a transaction t, the executing replica atomically multicasts t's id, the version of the database on which t was executed, and t's read-set and write-set. As soon as t is ordered, each replica is able to certify t on its own. For the certification procedure, t's read-set and write-set are checked against the write-sets of all transactions committed since t's database version. If they do not intersect, t commits, otherwise t aborts. If t commits then its changes are applied through the execution of a high priority transaction consisting of updates, inserts and deletes according to t's previously multicast write-set. Again, the high priority of the transaction means that it must be assured of acquiring all required write locks, possibly aborting any locally executing transactions. The executing replica replies to the client at the end of t.

In both protocols, transactions are queued while executing, as would happen in a non-replicated database, using whatever native mechanism is used to enforce ACID properties. This is queue Q0 in Figures 13.11 and 13.10.

The most noteworthy feature of both protocols is that since a transaction starts until it is certified, it is vulnerable to being aborted by a concurrent conflicting transaction that commits. On the other hand, from the instant that a transaction is certified

until it finally commits on every node, it is a menace to other transactions which will be aborted if they touch a conflicting item. Latency in any processing stage is thus bound to increase the abort rate. A side-effect of this is that the resulting system, when loaded, is extremely unfair to long running transactions.

In the DBSM, the additional latency introduced by replication is in the atomic multicast step, similarly to NODO (Q1) in Figure 8(a). This is an issue in WANs [20] and can be addressed with optimistic delivery. PGR [24] does not use optimistic delivery. In clusters, latency comes from exhausting resources within each replica as queues build up in Q0 and Q2. It is thus no surprise that any contention whatsoever makes the abort rate increase significantly.



1 - Multicast
2 - Certify Optmistically (unlikely)
3 - Final Delivery, missed order and then rollback (unlikely, but it doesn't matter)
4 - Final Delivery, correct order
5 - Certify in order
6 - Commit/Abort in order

Fig. 13.11 States, transitions, and queues in DBSM.

13.6.3 Active Replication

Active replication is a technique to build fault-tolerant systems in which transactions are deterministically processed at all replicas. Specifically, it requires that each transaction's statement be processed in the same order by all replicas. This might be ensured by means of a centralized or a distributed scheduler.

Sequoia [12], which was built after C-JDBC [10], for instance, uses a centralized scheduler at the expense of introducing a single point of failure. Usually, any distributed scheduler would circumvent this resilience problem but would require a distributed deadlock detection mechanism. To avoid distributed deadlocks, one might annotate transactions with conflict-classes and request distributed locks through an atomic multicast before starting executing a transaction. In contrast with NODO, however, a reliable message to propagate changes would not be needed as transactions would be actively executed. In both approaches, the consistency criteria would be similar to those provided by NODO.

The case against active replication is shown in NODO [36]: unbearable contention with high write ratio. This technique additionally has the drawback of re-

quiring a parser to remove non-deterministic information (e.g. random() or date()), thereby leading to re-implementing several features already provided by a database management system.

The active replication pays off when the overhead between transferring raw updates in a passive replication is higher than re-executing the statements. And of course, it makes it easy to execute DDL statements.

13.6.4 Hybrid Replication

Akara [20] pursues a hybrid approach: it ultimately enforces conservative execution to ensure fairness while leveraging the optimistic execution of transactions to attain an efficient usage of resources, and still provides the ability to actively replicate transactions when required.

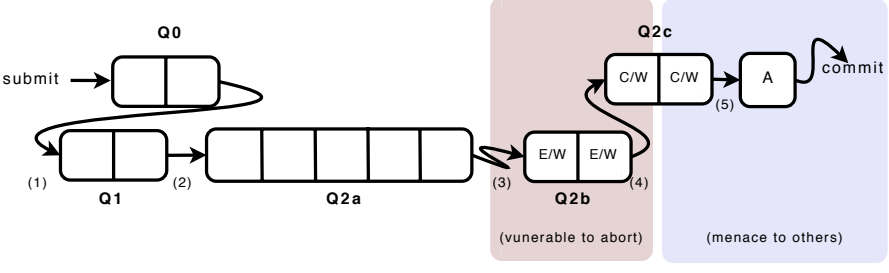


Fig. 13.12 States, transitions, and queues in Akara.

Figure 13.12 depicts the major states, transitions, and queues of the protocol. For the sake of simplicity, as in Section 13.6.1, we assume conflict classes correspond to tables (typical case) and that all transactions access at least a common table (interesting cases).

Upon submission, transactions are classified according to a set of conflict classes and totally ordered by means of an atomic multicast. Once ordered, a transaction is queued into Q2a waiting to be scheduled. Progression in Q2a depends on an admission control policy. When a transaction reaches the top of Q2a it is transferred to Q2b and then executed. Transactions run while in Q2b are said to be executed optimistically as they may end up aborting due to conflicts with concurrent transactions in Q2b or Q2c. After execution, and having reached the top of Q2b, a transaction is transferred to Q2c. When a transaction reaches the top of Q2c it may be ready to commit (it may also need to abort due to conflicts). If it is ready to commit, its changes are propagated to all other replicas and the transaction commits. Otherwise,

the transaction is forced to re-execute conservatively by imposing its priority on any locally running transaction.

The Akara protocol maximizes resource usage through the concurrent execution of potentially conflicting transactions by means of an admission control mechanism. It is worth noticing however that an admission policy that only allows to execute non-conflicting transactions according to their conflict classes makes Akara a simpler conservative protocol like NODO. The key is therefore to judiciously schedule the execution of each transaction in order to exploit resource availability thus reducing contention introduced by a conservative execution while at the same time avoiding re-execution. In [20] a simple policy that fixes the number of concurrent optimistically executed transactions is adopted. More sophisticated policies taking into account the actual resource usage or even dynamic knowledge of the workload could be used.

The mix of conservative and optimistic executions may lead to local deadlocks. Consider two conflicting transactions t and $t\prime$ that are ordered $< t, t\prime >$ and scheduled to run concurrently (both are in Q2b). If $t\prime$ grabs a lock first on a conflicting data item, it prevents t from running. However $t\prime$ cannot leave Q2b before t without infringing the global commit order.

If both transactions have the same conflict classes and, of course, are locally executed at the same replica, the proposed solution is to allow t' to overtake t in the global commit order. Notice that when a transaction t is totally ordered this ensures that no conflicting transaction will be executed concurrently at any other replica. Therefore, if t's order is swapped with that of $t\prime$ with the very same conflict classes then it is still guaranteed that both t and $t\prime$ are executed without the interference of any remote conflicting transaction. In the experiments conducted with the TPC-C (Section 13.6.5), for example, the likelihood of having two transactions with the very same conflict classes is more than 85% of the occurrences.

Finally, the protocol also allows transactions to be actively executed, thus providing a mechanism to easily replicate DDL statements and to reduce network usage for transactions with very large write-sets. A transaction t marked as active is executed at all replicas without distinction between an initiating or a remote replica, and its execution is straightforward. When t can be removed from Q2a, it is immediately moved to Q2b, and so forth, until it gets to Q2c. When t can proceed from Q2c, it is executed with high priority, committed, and then removed from Q2c. Active transactions are not executed optimistically to avoid different interleaves at different replicas.

13.6.5 Evaluation

To evaluate the protocols we use a hybrid simulation environment that combines simulated and real components [44]. The key components, the replication and the group communication protocols, are real implementations while both the database engine and the network are simulated.

In detail, we use a centralized simulation runtime based on the standard Scalable Simulation Framework (SSF) [1], which provides a simple yet effective infrastruc-

ture for discrete-event simulation. Simulation models are built as libraries that can be reused. This is the case of the SSFNet [14] framework, which models network components (e.g. network interface cards and links), operating system components (e.g. protocol stacks), and applications (e.g. traffic analyzers). Complex network models can be configured using these components, mimicking existing networks or exploring particularly large or interesting topologies.

To combine the simulated components with the real implementations the execution of the real software components is timed with a profiling timer [38] and the result is used to mark the simulated CPU busy during the corresponding period, thus preventing other jobs, real or simulated, from being attributed simultaneously to the same CPU. The simulated components are configured according to the equipment and scenarios chosen for testing as described in this section.

The database server handles multiple clients and is modeled as a scheduler and a collection of resources, such as storage and CPUs, and a concurrency control module. The database offers the reflector interface (Section 13.3) and implements multiversion concurrency control.

Each transaction is modeled as a sequence of operations: i) fetch a data item; ii) do some processing; iii) write back a data item. Upon receiving a transaction request each operation is scheduled to execute on the corresponding resource. The processing time of each operation is previously obtained by profiling a real database server.

A database client is attached to a database server and produces a stream of transaction requests. After each request is issued, the client blocks until the server replies, thus modeling a single threaded client process. After receiving a reply, the client is then paused for some amount of time (thinking time) before issuing the next transaction request.

To determine the read-set and write-set of a transaction's execution, the database is modeled as a set of histograms. The transactions' statements are executed against this model and the read-set, write-set and write-values are extracted to build the transaction model that is injected into the database server. In our case, this modeling is rather straightforward as the database is very well defined by the TPC-C [46] workload that we use for all tests. Moreover, as all the transactions specified by TPC-C can be reduced to SPJ queries, the read-set extraction is quite simple.

Clients run an implementation that mimics the industry standard on-line transaction processing benchmark TPC-C. TPC-C specifies five transactions: *NewOrder* with 44% of the occurrences; *Payment* with 44%; *OrderStatus* with 4%; *Delivery* with 4%; and *StockLevel* with 4%. The *NewOrder*, *Payment* and *Delivery* are update transactions while the others are read-only.

For the experiments below we added to the benchmark three more transactions that mimic maintenance activities such as adding users, changing indexes in tables or updating taxes over items. Specifically, the first transaction *Light-Tran* creates a constraint on a table if it does not exist or drops it otherwise. The second transaction *Active-Tran* increases the price of products and is actively executed. Conversely, *Passive-Tran* does the same maintenance activity but its changes are passively prop-

agated. These transactions are never executed in the same run, have a probability of 1% and when are executing the probability of the *NewOrder* is reduced to 43%.

The database model has been configured using the transactions' processing time of a profiled version of PostgreSQL 7.4.6 under the TPC-C workload. From the TPC-C benchmark we only use the specified workload, the constraints on throughput, performance, screen load and background execution of transactions are not taken into account.

We consider a LAN with 9 replicas. In the LAN configuration the replicas are connected by a network with a bandwidth of 1Gbps and a latency of $120\mu s$. Each replica corresponds to a dual processor AMD Opteron at 2.4GHz with 4GB of memory, running the Linux Fedora Core 3 Distribution with kernel version 2.6.10. For storage we used a fiber-channel attached box with 4, 36GB SCSI disks in a RAID-5 configuration and the Ext3 file system.

We varied the total number of clients from 270 to 3960 and distributed them evenly among the replicas and each run has 150000 transactions.

Experimental Results In what follows, we discuss the queues for each protocol described in previous sections. For the NODO approach, we use the simple definition of a conflict class for each table, which can be easily extracted from the SQL code. We do not consider finer granularity due to the possibility of inconsistencies when labeling mistakes are made. Figures 13.13 and 13.14 compare the DBSM, PGR and NODO.

The DBSM and PGR show a throughput higher than 20000 *tpm* (Figure 13(a)). In fact, both present similar results and the higher the throughput the higher the number of requests per second inside the database (Figure 13(b)). These requests represent access to the storage, CPU, lock manager and to the replication protocol. Clearly, the database is not a bottleneck. In contrast, the throughput presented by NODO is extremely low, around 4000 *tpm*, and its latency is extremely high (Figure 13(c)). This drawback can be easily explained by the contention observed in Q2 (Figure 13(d)).

Unfortunately, with the conservative and optimistic approaches presented above, one may have to choose between latency and fairness. In the NODO, for 3240 clients, 2481 transactions wait in Q2 around 40 *s* to start executing (Figure 14(a)). In contrast, an optimistic transaction waits 1000 times less and the number of transactions waiting to be applied is very low.

The abort rate is below 1% in both optimistic approaches as there is no contention and the likelihood of conflicts is low in such situations (Figure 14(b)). However, to show that the optimistic protocols may not guarantee fairness, we conducted a set of experiments in which one requests an explicit table level locking on behalf of the *Delivery* transaction thus mimicking a hotspot. This is a pretty common situation in practice, as application developers may explicitly request locks to improve performance or avoid concurrency anomalies. In this case, the abort rate is around 5% and this fact does not have an observable impact on latency and throughput but almost all Delivery Transactions abort, around 99% (Figure 14(c)).

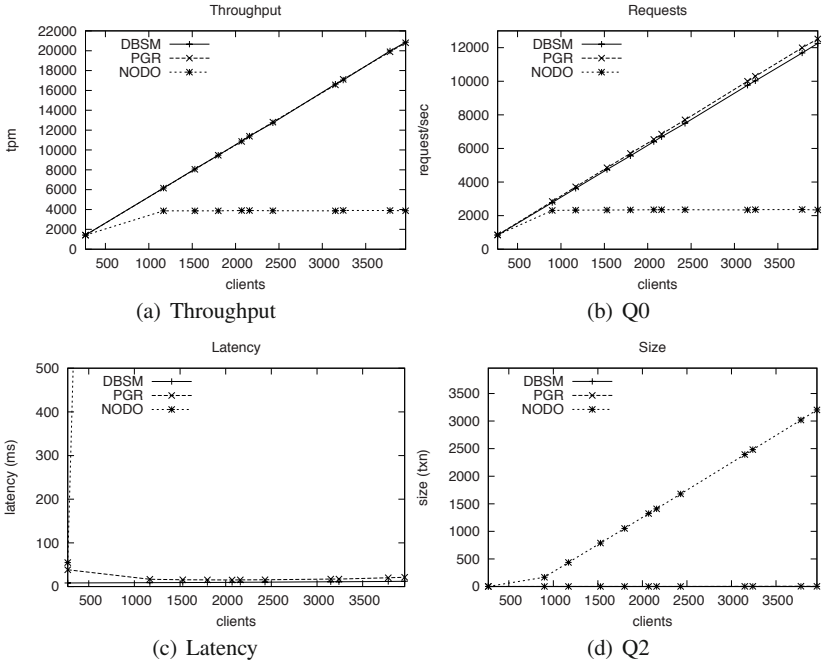


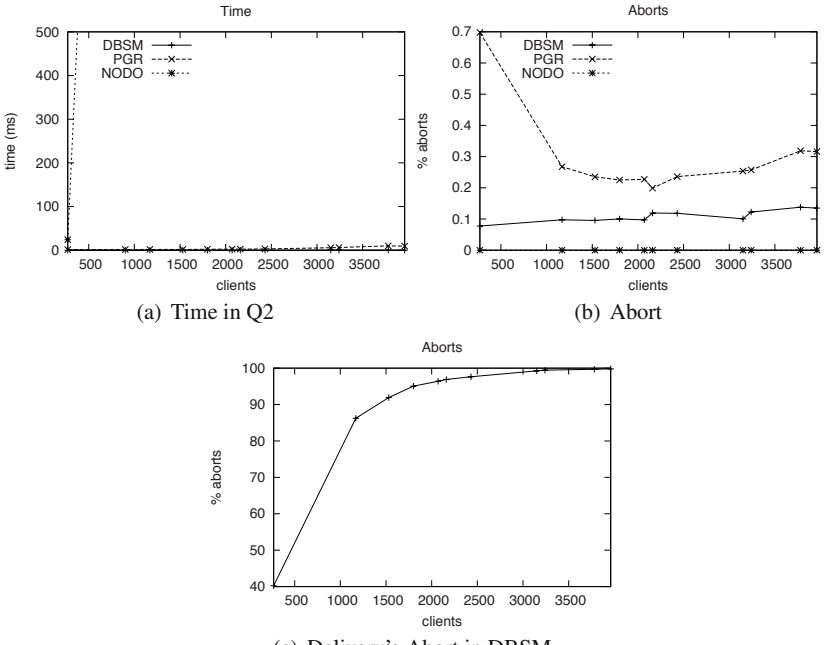
Fig. 13.13 Performance of DBSM, PGR and NODO.

In [21], a table level locking is acquired on behalf of the *Delivery* transaction to avoid flooding the network and improve the certification procedure. Although the reason to do so is different, the issue is the same.

In all the experiments, the time between an optimistic delivery and a final delivery were always below 1 *ms*, thus excluding Q1 from being an issue.

To improve the performance of the conservative approach while at the same time guaranteeing fairness, we used the Akara protocol. We ran the Akara protocol varying the number of optimistic transactions that might be concurrently submitted to the database in order to figure out which would be the best value for our environment. This degree of optimistic execution is indicated by a number after the name of the protocol. For instance, Akara-25 means that 25 optimistic transactions might be concurrently submitted and Akara-n means that there is no restriction on this number.

Table 13.1 shows that indefinitely increasing the number of optimistic transactions that might be concurrently submitted is not worth. Basically for Akara-n, latency drastically increases and as a consequence throughput decreases. This occurs because the number of transactions that fails the certification procedure increases. For 3240 clients, more than 89% of the transactions fail the certification procedure (i.e. in-core certification procedure like in PGR, see Section 13.6.2). Furthermore, after failing such transactions are conservatively executed and compete for resources



(a) Time in Q2 (b) Abort

(c) Delivery's Abort in DBSM

Fig. 13.14 Latency vs. abort rate (DBSM, PGR and NODO).

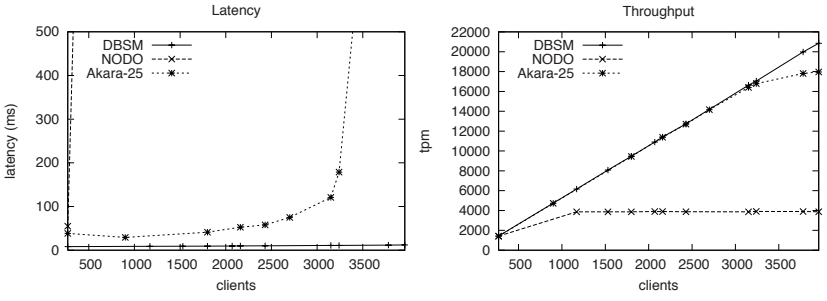
Table 13.1 Analysis of Akara.

	Lat (ms)	Tput (tpm)	Unsuccess(%)
Akara-25	178	16780	2
Akara-45	480	16474	5
Akara-n	37255	3954	89
Akara-25 with Light-Tran	8151	9950	21
Akara-25 with Active-Tran	109420	1597	21
Akara-25 with Passive-Tran	295884	625	22

with optimistic transactions that may be executing. Keeping the number of optimistic transactions low however reduces the number of transactions allowed in the database and neither is worth. After varying this number from 5 to 50 in steps of 1, we figured out that the best value for the TPC-C in our environment is 25.

In what follows, we used the DBSM as the representative of the family of optimistic protocols thus omitting the PGR. Although both protocols present similar performance in a LAN, the PGR is not worth in a WAN due to its extra communication step.

Figure 13.15 depicts the benefits provided by the Akara-25. In Figure 15(a), we notice that latency in the NODO is extremely high. In contrast, the Akara-25 starts degenerating after 3240 clients. For 3240 clients the latency in the DBSM is about 9 *ms*, and in the Akara-25, it is about 178 *ms*. This increase in latency directly af-



(a) Latency (b) Throughput

Fig. 13.15 DBSM, NODO and Akara-25.

fects throughput as shown in Figure 15(b). The NODO presents a steady throughput of 4000 *tpm*; the Akara-25, a steady throughput of 18605 *tpm* after 3960 clients; while the DBSM increases its throughput almost linearly. The DBSM starts degenerating when the database becomes a bottleneck what was not our goal with these experiments.

Table 13.1 shows the impact on performance when the maintenance activities are handled by our protocol. These maintenance activities represented by the transactions *Active-Tran* and *Light-Tran* are actively executed and integrated in runs with the *Akara-25*: *Akara with Active-Tran* and *Akara with Light-Tran*, respectively. In order to show the benefits of an active execution in such scenario, we provide a run named *Akara with Passive-Tran* in which the updates performed by the *Active-Tran* are atomically multicast. The run with the *Passive-Tran* presents a latency higher than that with the *Active-Tran* as the former needs to transfer the updates through the network. However, both approaches have a reduced throughput and high latency when compared to the normal Akara-25 due to contention caused by a large number of updates.

The run with the *Light-Tran* does not have a large number of updates but its throughput decreases when compared to the Akara-25 due to failures in the certification procedure. This is caused by the fact that the transaction *Light-Tran* mimics a change on the structure of a table and thus requires an exclusive lock on it.

In a real environment, we expect that maintenance operations occur with a rate lower than 1% and so they should not be a problem as the optimistic execution of other transactions might compensate the temporary decrease in performance.

13.7 Conclusions

This chapter addresses the existing trade-offs when implementing database replication in different environments. It shows that database replication in practice is constrained by a variety of architectural, algorithmic, and dynamic issues.

To address these issues, a generic architecture that supports legacy database management systems without compromising the performance that can be achieved in na-

tive implementations is described. Then, a communication abstraction that encapsulates distributed agreement and supports a range of implementations and advanced optimizations is presented. Finally, a modular approach to implementing replication protocols is put together and evaluated, showing how different algorithmic choices match assumptions on system dynamics and performance.

The experimental results reported here point out that a successful practical application of database replication, in particular, when strong consistency is sought, depends on a combination of factors. Namely, that the architectural approach to interfacing the database server dictates which replication algorithms are feasible; and that the availability of different communication primitives directly impacts the efficiency of different algorithms in a particular setting. By taking these factors into account, it is possible to achieve good performance in face of variable workloads and environments.

Acknowledgements This work was partially supported by the GORDA (FP6-IST2-004758) and the Pastramy (PTDC/EIA/72405/2006) projects.

References

1. http://www.ssfnet.org/
2. Amir, Y., Danilov, C., Stanton, J.: A low latency, loss tolerant architecture and protocol for wide area group communication. In: IEEE/IFIP International Conference on Dependable Systems and Networks (2000)
3. Apache DB Project. Apache Derby version 10.2 (2006), http://db.apache.org/derby/
4. Babaoglu, O., Davoli, R., Montresor, A.: Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications. Operating Systems Review 31(2) (1997)
5. Ban, B.: Design and implementation of a reliable group communication toolkit for Java (1998), http://www.cs.cornell.edu/home/bba/Coots.ps.gz
6. Bartoli, A., Babaoglu, O.: Selecting a "primary partition" in partitionable asynchronous distributed systems. In: IEEE International Symposium on Reliable Distributed Systems (1997)
7. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Distributed Database Systems. Addison-Wesley, Reading (1987)
8. Birman, K.P., van Renesse, R.: Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press, Los Alamitos (1993)
9. Carvalho, N., Correia Jr., A., Pereira, J., Rodrigues, L., Oliveira, R., Guedes, S.: On the use of a reflective architecture to augment database management systems. Journal of Universal Computer Science 13(8) (2007)
10. Cecchet, E., Marguerite, J., Zwaenepoel, W.: C-JDBC: Flexible database clustering middleware. In: USENIX Annual Technical Conference (2004)
11. Continuent. Sequoia v2.10 (2007), http://sequoia.continuent.org
12. Continuent. Sequoia 4.x (2008), http://sequoia.continuent.org
13. Correia Jr., A., Pereira, J., Rodrigues, L., Carvalho, N., Vilaça, R., Oliveira, R., Guedes, S.: GORDA: An open architecture for database replication. In: IEEE International Symposium on Network Computing and Applications (2007)
14. Cowie, J., Liu, H., Liu, J., Nicol, D., Ogielski, A.: Towards realistic million-node Internet simulation. In: International Conference on Parallel and Distributed Processing Techniques and Applications (1999)

15. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys 36(4) (2004)
16. Dolev, D., Malki, D., Strong, R.: A framework for partitionable membership service. In: ACM Symposium on Principles of Distributed Computing (1996)
17. Garcia-Molina, H., Ullman, J., Widom, J.: Database Systems The Complete Book. Prentice-Hall, Englewood Cliffs (2002)
18. Guerraoui, R., Kostic, D., Levy, R., Quema, V.: A high throughput atomic storage algorithm. In: IEEE International Conference on Distributed Computing Systems (2007)
19. Hayden, M.: The Ensemble System. PhD thesis, Cornell University, Computer Science Department (1998)
20. Correia Jr, A., Pereira, J., Oliveira, R.: AKARA: A flexible clustering protocol for demanding transactional workloads. In: International Symposium on Distributed Objects and Applications (2008)
21. Correia Jr., A., Sousa, A., Soares, L., Pereira, J., Moura, F., Oliveira, R.: Group-based replication of on-line transaction processing servers. In: Maziero, C.A., Gabriel Silva, J., Andrade, A.M.S., de Assis Silva, F.M. (eds.) LADC 2005. LNCS, vol. 3747, pp. 245–260. Springer, Heidelberg (2005)
22. Kaashoek, M., Tanenbaum, A.: Group communication in the Amoeba distributed operating system. In: IEEE International Conference on Distributed Computing Systems (1991)
23. Keidar, I., Dolev, D.: Totally ordered broadcast in the face of network partitions. In: Dependable Network Computing, Kluwer Academic Publishers, Dordrecht (2000)
24. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In: VLDB Conference (2000)
25. Kiczales, G.: Towards a new model of abstraction in software engineering. In: IMSA Workshop on Reflection and Meta-level Architectures (1992)
26. Lin, Y., Kemme, B., Jiménez Peris, R., Patiño Martínez, M.: Middleware based data replication providing snapshot isolation. In: ACM SIGMOD (2005)
27. Maes, P.: Concepts and experiments in computational reflection. In: ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (1987)
28. Miranda, H., Pinto, A., Rodrigues, L.: Appia: a flexible protocol kernel supporting multiple coordinated channels. In: IEEE International Conference on Distributed Computing Systems (2001)
29. Mocito, J., Respicio, A., Rodrigues, L.: On statistically estimated optimistic delivery in large-scale total order protocols. In: IEEE International Symposium on Pacific Rim Dependable Computing (2006)
30. Oliveira, R., Pereira, J., Correia Jr, A., Archibald, E.: Revisiting 1-copy equivalence in clustered databases. In: ACM Symposium on Applied Computing (2006)
31. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Journal of Distributed and Parallel Databases and Technology (2002)
32. Pedone, F., Schiper, A.: Optimistic atomic broadcast. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 318–332. Springer, Heidelberg (1998)
33. Pedone, F., Schiper, A.: Handling message semantics with generic broadcast protocols. Distributed Computing 15(2) (2002)
34. Pereira, J., Rodrigues, L., Monteiro, M.J., Oliveira, R., Kermarrec, A.-M.: NeEM: Network-friendly epidemic multicast. In: IEEE International Symposium on Reliable Distributed Systems (2003)
35. Pereira, J., Rodrigues, L., Oliveira, R.: Semantically reliable multicast: Definition, implementation and performance evaluation. IEEE Transactions on Computers, Special Issue on Reliable Distributed Systems 52(2) (2003)
36. Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: Scalable replication in database clusters. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, p. 315. Springer, Heidelberg (2000)
37. Jiménez Peris, R., Patiño Martínez, M., Kemme, B., Alonso, G.: Improving the scalability of fault-tolerant database clusters. In: IEEE International Conference on Distributed Computing Systems (2002)

38. Pettersson, M.: Linux performance counters, http://user.it.uu.se/~mikpe/linux/perfctr/
39. PostgreSQL Global Development Group. Postgresql version 8.1 (2006), http://www.postgresql.org/
40. Rodrigues, L., Fonseca, H., Veríssimo, P.: Totally ordered multicast in large-scale systems. In: IEEE International Conference on Distributed Computing Systems (1996)
41. Rodrigues, L., Mocito, J., Carvalho, N.: From spontaneous total order to uniform total order: different degrees of optimistic delivery. In: ACM Symposium on Applied Computing (2006)
42. Salas, J., Jimenez-Peris, R., Patino-Martinez, M., Kemme, B.: Lightweight reflection for middleware-based database replication. In: IEEE International Symposium on Reliable Distributed Systems (2006)
43. Sousa, A., Pereira, J., Moura, F., Oliveira, R.: Optimistic total order in wide area networks. In: IEEE International Symposium on Reliable Distributed Systems (2002)
44. Sousa, A., Pereira, J., Soares, L., Correia Jr., A., Rocha, L., Oliveira, R., Moura, F.: Testing the dependability and performance of GCS-based database replication protocols. In: IEEE/IFIP International Conference on Dependable Systems and Networks (2005)
45. Sussman, J., Keidar, I., Marzullo, K.: Optimistic virtual synchrony. In: Symposium on Reliability in Distributed Software (2000)
46. Transaction Processing Performance Council (TPC). TPC benchmark C Standard Specification Revision 5.0 (2001)
47. Vicente, P., Rodrigues, L.: An indulgent uniform total order algorithm with optimistic delivery. In: IEEE International Symposium on Reliable Distributed Systems (2002)