Masters thesis

Database Replication Prototype

Roel Vandewall

supervised by Matthias Wiesmann, prof. André Schiper, and prof. Wim H. Hesselink

19.08.2000

Abstract

This report describes the design of a Replication Framework that facilitates the implementation and comparison of database replication techniques. Furthermore, it discusses the implementation of a Database Replication Prototype and compares the performance measurements of two replication techniques based on the Atomic Broadcast communication primitive: *pessimistic* active replication and *optimistic* active replication.

The main contributions of this report can be split into four parts. Firstly, a framework is proposed that accommodates the comparison of various replication techniques. Secondly, the implementation requirements and the theoretical performance characteristics of the pessimistic and the optimistic active replication techniques are thoroughly analysed. Thirdly, the two techniques have been implemented within the framework as a proof of concept, forming the Database Replication Prototype. Finally, we present the performance results obtained using the Database Replication Prototype. They show that in large-scale networks, optimistic active replication outperforms pessimistic active replication.

Contents

List of Figures

Introduction

A *database system* is a computer system that offers data storage facilities to client applications. Database systems constitute an increasingly vital part of contemporary applications, such as search-engines, groupware and banking systems. The popularity of database systems comes from the fact that they offer abstractions, features and guarantees that surpass those offered by, for example, the standard file system of the operating system. For instance, a database system [EN94]

- provides an interface that abstracts from the low level problems of data storage and retrieval (for example, a query language such as SQL)
- allows concurrent access to data while guaranteeing data integrity
- survives severe failures such as machine crashes or power failures without corrupting data

In the mentioned applications, many, often impatient users access the database system concurrently. These users demand high availability and quick response times. As we discuss next, current database systems sometimes slow down applications, which in turn fail to meet the expectations and demands of their users.

The problem: database reliability and performance

Traditionally, a database system is implemented *centrally*: the system runs on one single machine. This approach has the following disadvantages:

- when the machine fails, the whole database is unavailable for extended periods of time
- the machine can become a bottleneck to applications when the load exceeds the machine's maximum throughput.

A solution: database replication

An alternative approach is the *replicated* database system. In this system, identical and redundant copies (*replicas*) of a database are distributed over different machines linked by a network. Compared to the aforementioned centralized database, the following advantages can be obtained:

- availability is improved (due to the redundancy, the system can remain fully available even when a few replicas fail)
- performance is improved:
  - the throughput is larger (since computation can be distributed over the replicas)
  - response times are shorter (because replicas can be situated closer to applications)

In Figure 1, a replicated system is shown that consists of three replicas. The small cylinders represent the three replicas that each manage an identical copy of the database. Users of the system are not bothered by the fact that it is replicated: logically it acts as one database system, as depicted by the large cylinder.

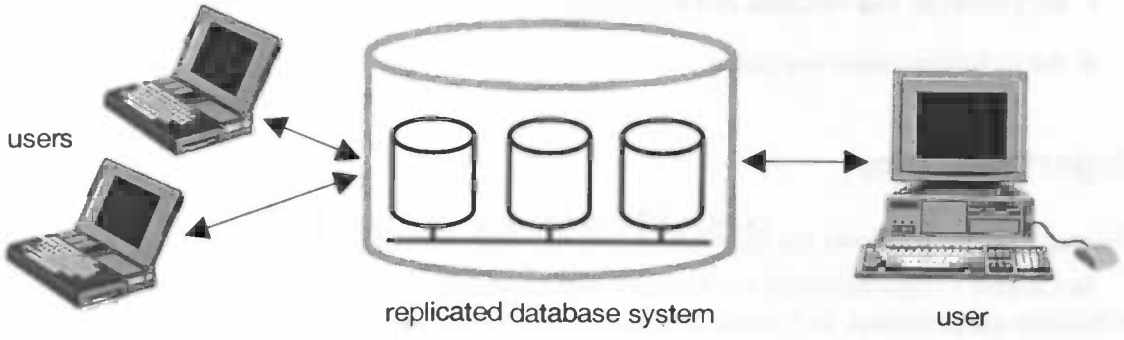

Figure 1: Replication example

Project goals and project steps

The replicas mentioned in the previous section need to stay up-to-date when the database is changed. To achieve this, special *replication techniques* (replication protocols) are used that operate over networks interconnecting the replicas. The database community has devised many techniques during the last twenty years, but either these techniques are very slow, or they fail to satisfy certain desirable correctness properties.[1]

A possibility recently suggested is to base replication techniques on Group Communication primitives known from the distributed systems community. It is expected that this way, alternative techniques can be devised that are correct and still perform reasonably fast. The main goal of the DRAGON project [DRA98], in the context of which this Masters project was conducted, is to perform research into this direction and decide whether these new techniques constitute a viable alternative.

The goals of this Masters project were to:

1. create a *replication framework* that facilitates the implementation and comparison of various replication techniques in a prototype environment called the **Database Replication Prototype**
2. implement two replication techniques based on Group Communication primitives (called pessimistic and optimistic active replication) as a proof of concept
3. measure the performance characteristics of both techniques using the Database Replication Prototype
4. compare both techniques by evaluating the measurement results obtained

To attain these goals, the following steps were taken:

1. the field of database replication was briefly studied to form an understanding of the problems and system models involved
2. existing replication techniques based on Group Communication were studied carefully to determine their characteristics and implementation requirements

[1] These correctness properties, also called the ACID properties, are discussed in section 1.1.

- the replication framework was devised
- the two techniques were implemented using the framework
- the prototype was validated and experiments were conducted to obtain performance measurements
- the techniques were compared

Report structure

This report closely follows the steps outlined in the previous section.

In Chapter 1 (System Model and Definitions), the adopted system model and some basic concepts and definitions are presented. In Chapter 2 (Replication Framework), the various components of the replication framework are defined. Chapter 3 (Active Replication Techniques) gives a detailed description of the implemented replication techniques in the context of the replication framework.

Chapter 4 (Database Replication Prototype) briefly discusses how the prototype was implemented according to the Replication Framework. In Chapter 5 (Performance Results and Evaluation), the performance results obtained using the prototype are presented and discussed. Finally, Chapter 6 (Conclusion) summarizes the results obtained and gives some directions for future work.

Chapter 1

System model and definitions

In this chapter, the system model and main definitions used for the Database Replication Prototype are presented. Then, the concept of replication is related to the model.

1.1 System model

We consider the following system model [BHG87]. Client applications connect to a server on which a database system runs. A client can submit read and write operations to a server in order to retrieve data from, and store data in the database system. These read and write operations are submitted in the context of transactions, so that certain desirable properties (e.g., data integrity) can be guaranteed. So, the system model roughly follows that of a distributed, client-server, transactional database system.

In this section we briefly explain the characteristics of such a database system and define the concepts that are used in the rest of report. At the end, we motivate the choice for this system model.

1.1.1 Client and server processes

In the system, two types of processes can be distinguished: *server processes* and *client processes*. Each server process has its own database storage and processes database operations (for example, retrieving or storing data). A *client process* submits operations to a server process using some communication system and may do computations (see Figure 1.1).

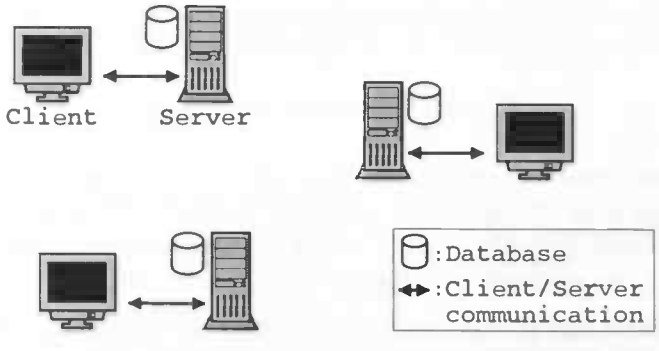

Figure 1.1: Basic system model

1.1.2 Data and operations

In the database storage of each server process, a constant number of items is stored. These items contain values of equal size[1] and are denoted by integer locations. The database may thus be viewed as a one-dimensional array that contains a constant number of values. The data operations supported on each item are:

- read(location), which returns the value stored at location
- write(location, value), which sets the value stored at location to value

A particular assignment of values to all items in the database is called a *database state*.

Notation: In most examples, r and w are used to mean read and write, respectively.

1.1.3 Transactions

Clients always submit data operations in the context of transactions. A *transaction* is an atomic unit of work that is either completed in its entirety or not completed at all [EN94].

The transaction concept facilitates reasoning about the execution of operations that logically belong together. Consider, for example, the updating of a bank account when doing a cash withdrawal action. The old account value must be read, the amount withdrawn must be subtracted, and the resulting value must be written back into the database system. This cash withdrawal *transaction* consists of two operations, read(Account) and write(Account, value), that belong together: either both must be performed, or none of them (resulting in cancellation of the withdrawal).

In the next subsection we discuss which properties the database system must fulfill when processing transactions. Before that, we define the transaction concept more formally.

A transaction is started with the transaction operation begin, followed by one or more data operations, and terminated with either the operation abort or the operation commit. The begin operation announces that a certain transaction is going to be submitted to the system.[2] When a transaction ends with an abort, all its effects on the database (that is, all its write operations) are discarded. If it ends with a commit, all its effects are made permanent.

For an example transaction, see Figure 1.2. In this example, the database items contain integer values. The transaction consists of five operations: two transaction operations and three data operations. A client submitting the transaction would do so sequentially, from left to right.

Figure 1.2: An example transaction

A transaction is called *read-only* (or a *query*) if its data operations are all reads. All other transactions are called *update transactions*. In Figure 1.2, an update transaction is shown. The *readset* (*writeset*) of a transaction contains every database item for which the transaction contains at least one read (write) operation. In Figure 1.2, the readset is $\{7\}$ and the writeset is $\{3, 42\}$.

A transaction (client) is said to *request commit* if it ends with (submits) a commit operation.

Two operations are said to *conflict* if they operate on the same database item, are issued by different transactions, and at least one of them is a write operation. Two transactions are said to *interfere* when one or more of their operations conflict.

[1] This size is called the *item granularity* [EN94].

[2] One could omit the begin operation and let the first data operation of a transaction implicitly announce the beginning of that transaction. For presentation clarity we decided to make the beginning of a transaction explicit.

The data operations and the transaction operations together form the full set of operations supported by the database system.

1.1.4 ACID properties

It is considered desirable for the following four transaction properties, called the ACID properties, to always hold (directly from [EN94]):

- **A**tomicity: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **C**onsistency preservation: A correct execution of the transaction must take the database from one consistent state to another.
- **I**solation: A transaction should not make its updates (i.e., the effect of its write operations) visible to other transactions until it is committed.
- **D**urability (or recoverability): Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

The Consistency preservation property mentions a *consistent* database state. To see what this means, consider a database that stores employee records and a list of departments on behalf of some application. Each employee record contains information about an employee, including the department she belongs to. Assume that the application specifies the following invariant: in every database state, all employee records contain the name of a **listed** department. Now imagine a transaction that tries to set the department name in an employee record to a **non-listed** name. Committing such a transaction would break the aforementioned invariant and leave the database in an inconsistent state. However, when the Consistency preservation property is satisfied, such a transaction can not be committed.

In general, the Consistency preservation property specifies that every committed transaction leaves the database in a consistent state, i.e., a state that satisfies all invariants specified for the database. Since the property depends on the application-level meaning of the data stored in the database, we consider it an issue that should be solved at that level. This means that we expect the application to commit only transactions that respect the invariants.[3] Thus, we do not consider Consistency preservation in the remainder of this report.

The other three properties are considered when the correctness of the replicated database system described in this report is treated (section 3.4). To reason about the Isolation property, some more concepts are needed. These are presented in the next subsection.

1.1.5 Histories and serializability

To increase the performance of transaction processing, transactions are often executed in parallel. To reason about these (parallel) executions, for example to see if an execution upholds the Isolation property, the concepts of (execution) *history* and *serializability* are used [BHG87].

In this report we use the following, simplified notion of history: A *history* is a partial order of operations that includes only those operations specified by all committed transactions in the system.[4] Furthermore, the following must hold:

[3] Some database systems allow application invariants to be expressed as *constraints* on the database state. Upon committing a transaction, such a system checks whether the constraints will be violated because of the transaction. If so, it forcefully aborts the transaction. While the Consistency preservation is handled by the database system in this case, we see the support for constraints as an additional feature that could be added on top of the system model presented in this report.

[4] This notion corresponds to the committed projection of the history as defined in [BHG87].

- operations of the same transaction are ordered and appear in the history in the same order as they were specified by that transaction
- conflicting operations are ordered

For an example history, see Figure 1.3. At the top, two transactions are shown, consisting of five and four operations respectively. At the bottom, a possible history is shown. The history could occur on, e.g., a time shared system with one processor. It would first execute the first operation of transaction 1, then the first operation of transaction 2, then operation 2 and 3 of transaction 1, etc., as shown in the figure.

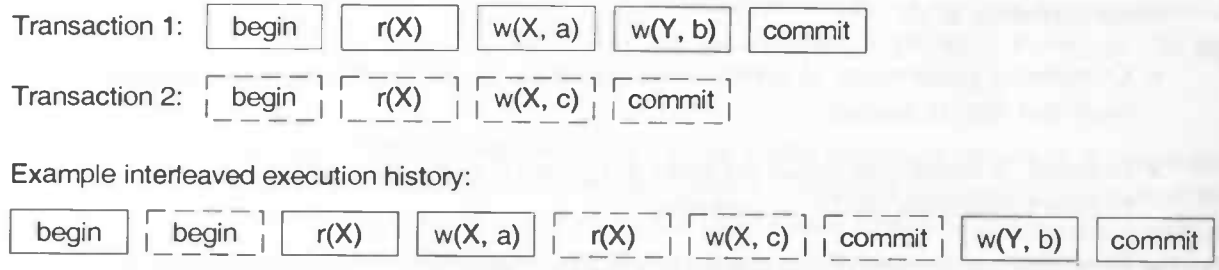

Figure 1.3: A history containing two transactions

To reason about database system correctness, it is useful to compare different histories. First, we give an informal description of the notion of history equivalence. Assume that every transaction's writes are a function of the values it reads [BHG87]. Then, two histories result in the same final database state if all read operations read the same values in both histories (because per assumption, all write operations then write the same values). Since the effect on the database state is the same for both histories, we call them equivalent.

A read operation r(X) of a transaction T_1 is said to *read from* a transaction T_2 when r(X) reads the value written to X by the last write(X) operation of T_2. Using this, we formally define that two histories H_1 and H_2 are *view equivalent* if the following conditions hold:

- exactly the same transactions and operations appear in both
- if some r(X) reads from some transaction T in H_1, it also reads from T in H_2
- if some w(X) is the last operation that writes to X in H_1, it is also the last operation that writes to X in H_2

A history is *serial* if none of the transaction operations are interleaved (i.e., such histories are produced by a database system that processes transactions sequentially as opposed to in parallel). A history is called *serializable* if it is *view equivalent* to some serial history.[5]

1.1.6 Database system correctness

Using the definitions of the previous subsection, we can now state the correctness criterion for database systems: **a database system is correct if all execution histories it produces are serializable**.

The rationale behind this definition is as follows: executing all transactions serially guarantees the Isolation property. So does an interleaved execution history if it is equivalent to a serial one, because all transactions *see* the same database view and thus behave identically in both histories.

[5]More precisely, this is called *view serializability*. Other notions of serializability exist but are not be considered here, because view serializability is needed to reason about replicated database systems [BHG87].

For a history that is serializable, see once again Figure 1.3 and observe that the interleaved execution is equivalent to the following serial execution: Transaction 1 followed by Transaction 2. For a history that is not serializable, see Figure 1.4. It contains the same transactions as the previous figure, but with a slight difference: the r(X) of transaction 2 is executed **in between** the r(X) and w(X, a) of transaction 1.

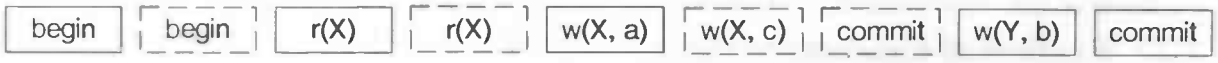

Figure 1.4: A non-serializable history

One could try to serialize the history shown in Figure 1.4 in two ways: Transaction 1 followed by Transaction 2 or the other way around. In the first case, T_2 would read X from T_1, whereas in the second, T_1 would read X from T_2. In both cases, this differs from the original history, where both T_1 and T_2 read the initial value of X. This means that the proposed serial histories are not view equivalent to the original history. Since we exhausted all possible serial histories, it follows that the history shown is not serializable.

An algorithm that ensures that concurrent transactions do not violate serializability, is called a *concurrency control technique* [EN94]. In section 2.2.3 we discuss how concurrency control is realized in the Database Replication Prototype.

1.1.7 Motivation for the system model

The system model for the Database Replication Prototype has been kept as simple as possible. This way, a working prototype could be constructed within a short period of time. However, the question arises whether the system model presented is general enough to encompass common real-world situations. This question is briefly discussed here.

The answer to the generality question can be short: the model allows any conceivable database function, including those used in real-world situations, to be expressed. To see this, imagine a client that is submitting the following transaction to the database:

1. begin the transaction
2. read all data from the database into some local storage
3. perform any calculations and make any modifications to the local storage (possibly including outputting data and/or asking a human user for input)
4. write all data in the local storage to the database
5. commit the transaction

Thus, clients can transfer the database from any state to any other, allowing all possible database applications to be modelled.

The fact that the database size is fixed could be seen as a limitation. However, this is not considered an actual constraint, since a real database system would run on a specific hardware configuration imposing its own size limitations. Setting the database size in the model to the maximum size supported by the hardware would in fact not constrain our database system any more than any other imaginable database system running on the same hardware.

Of course, creating a client application within the proposed model is a tedious task because only the low level read and write operations are available. Therefore, most database systems used in real-world applications provide higher level interfaces, such as the SQL database query language. These languages allow manipulation of data using additional operations and powerful abstractions. However, the system

model is still applicable to the lower level parts of these database systems. This because at some point during processing, the SQL statements are (automatically) transformed to low level read and write operations and passed on to a transaction processing engine. Conceptually, the Database Replication Prototype could fit in at exactly this level.

1.2 Replicated database systems

The model presented in the previous section does not specify how data is distributed among the different server processes. One possibility[6] is to require that all server processes, or *replicas*, contain identical copies of the database. This is the approach taken for the Database Replication Prototype. Its key properties are presented in this section.

synchronization Since it is invariant that each replica holds a fully replicated version of the database, the versions must be synchronized when a client submits a transaction that modifies the state of the database. When trying to commit a transaction, the replica must somehow communicate with all other replicas, using a *replication technique*, to decide on a new, system wide state of the database. Figure 1.5 shows a replicated database system: in addition to the server processes holding the databases and the clients, there is a communication network connecting the server processes.

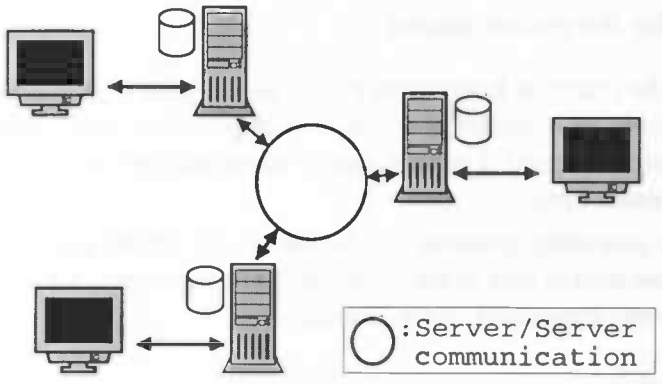

Figure 1.5: Communication in the replicated system model

One way to classify replication techniques is by the moment in time the client is informed of the result of a commit request [WPS+00b]. This can be (1) **after** the state changes have been synchronized and recorded among all replicas or (2) **before** there has been synchronization among the replicas.

The first class is called *eager* replication: replicas are not allowed to independently modify their copies of the database. The replicas always communicate and agree upon state changes before they commit transactions. The advantage of eager replication is that it satisfies the ACID properties. The drawback is that it tends to slow down response times because of the intra-replica communication taking place before the result of a commit request is sent to the client.

The second class is called *lazy* replication: transactions are independently committed at different replicas and state changes are propagated to the other replicas afterwards. The advantage of lazy replication is that response times are quick and that the updates of different transactions can be sent to the other replicas in batches, leading to less communication overhead. The drawback is that

[6] It is also possible to design a more complex system in which data can be replicated to different degrees, varying from fully replicated to not at all. We have chosen the simplicity of full replication, since we want to investigate and compare only the basic properties of replication techniques without worrying about complex implementation issues.

(temporary) data inconsistencies can occur between replicas, which violates the ACID properties. Furthermore, the database state needs to be reconciliated manually or automatically to correct the more severe inconsistencies.

Because we think that correctness is important to database users and that the use of suitable Group Communication primitives can minimize the time needed for communication, we focus on eager replication techniques. An additional reason is that comparing lazy techniques is very difficult because they violate the ACID properties in various ways (i.e., they fulfill different specifications).

distribution transparency In the proposed replicated database system, a client can submit its transactions to any replica. This because all replicas hold copies of the same database and all of them are available for operation processing.

As far as one client is concerned, there exists only one database. Since the client need not concern itself with updating all replicas when modifying the database, it may behave exactly the same as when connected to a centralized database system. This feature is called *distribution transparency* [EN94][7].

correctness The correctness criterion for a replicated database system is formulated as follows: **the execution history produced by all replicas together must be *one-copy serializable*** [BHG87]. Informally, this means that (1) every database item appears as one logical copy to all transactions, despite the fact that a copy of each item exists at every replica, and (2) every execution history produced by the replicated database system is view equivalent to some serial execution involving the logical copies. The formal definition of one-copy serializability requires a more thorough introduction to serializability theory than is provided in this report, so we do not present it here.

1.2.1 Advantages and cost of replication

In a centralized database system, all clients connect to the same server, whereas in a replicated database system, the clients may choose various servers to connect to. In the introduction chapter, both systems were already briefly compared. We now do this in some more detail. For quantitative comparisons, we refer to Chapter 5 (Performance Results) .

When each replica is running on a different machine, replicated database systems have, in theory, two key advantages over centralized databases:

high availability If a replica crashes due to a software or hardware failure, the remaining replicas can continue to operate. Compare this to a centralized database system, which becomes completely unavailable after one crash.

better performance The transaction processing load can be distributed among all replicas (machines) in the system. This leads to two improvements:

larger throughput The replicas can independently execute queries and the read operations of update transactions, because these don't change the database state. Only the write operations need to be executed on all replicas. Because of the fact that a part of the transaction load can be handled decentrally, the replicas can process more transactions per time unit than a centralized system (which must always execute all operations).

shorter response times Response times are shorter for queries because these can be executed on one replica, close to the client, without further communication between the replicas.

[7]More precisely, when talking about distribution transparency, [EN94] use the term *user* to refer to a client, and the term *client* to refer to the replication technique.

However, these advantages come at the following cost:

added processing and communication overhead The replicas must communicate to ensure that modifications are applied to all database copies. This increases the load on the machines (more precisely, the communication subsystem) and on the communication network, which may degrade overall performance.

higher system complexity The replicas run asynchronously on different machines and asynchronously receive client requests that modify the database. Reliably synchronizing the database copies across the replicas requires advanced communication and transaction processing algorithms.

Chapter 2

Replication Framework

The *replication framework* provides a conceptual infrastructure for the implementation and comparison of various replication techniques. The framework consists of five components, which are identified and specified in this chapter. First, the design methodology is explained.

2.1 Object oriented frameworks

The replication framework and the Database Replication Prototype, which is an instance of this framework, have been developed using object oriented methodology. An object oriented *framework* is a reusable design expressed as a set of abstract classes and the way their instances collaborate [Fel98].

Two types of frameworks can be distinguished: white-box and black-box frameworks. In a *white-box framework*, the user of the framework may modify the internal structure of components to suit his needs. In a *black-box framework*, the user may only access the components using the well-defined interfaces they provide.

The replication framework is a mixed white-box/black-box framework. As described in the next section, one of the components (the Replication Manager) is a white-box component: it needs to be internally modified according to the replication technique that is implemented. (How this can be done is explained in Chapter 3 for two specific replication techniques) The other four components are black-box components: all replication techniques rely on the interface defined for these components in the current chapter. In the next section, the components as well as their positions in the framework are outlined.

2.2 Framework Components

The following components, which each have a clearly distinctive function within the replication framework, can be identified:

Database Service Database storage and concurrency control facilities. A black-box component.

Group Communication Service Group Communication primitives used for communication between replicas. A black-box component.

Replication Manager Handles the transactions submitted by clients. Implements a given replication technique and provides (additional) concurrency control. A white-box component.

Client Transaction source that generates a workload. In the Database Replication Prototype, the Client is used to test the system and measure the performance of replication techniques. A black-box component.

Operations The database and transaction operations. A black-box component.

Figure 2.1 shows the dependency relationships between these components, as well as the processes in which they reside. In every replica, there exist more or less independent instances of the Database Service, the Group Communication Service and the Replication Manager. On the other hand, Operations are visible to a particular Client *and* to one or more replicas.

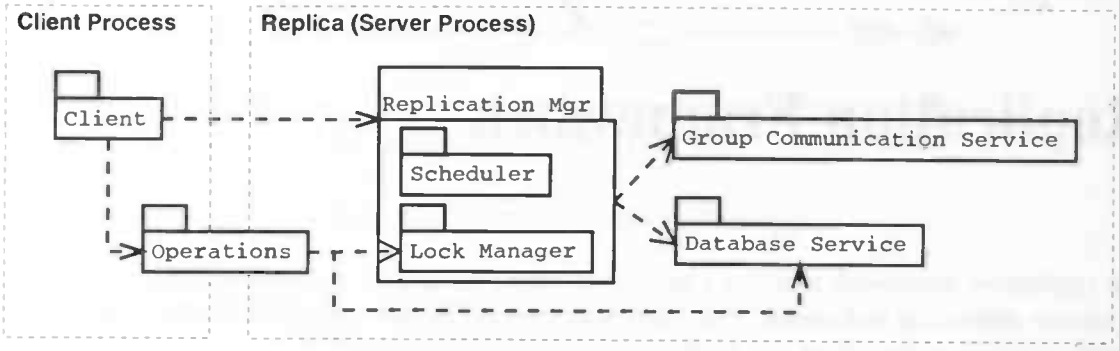

Figure 2.1: Component dependency relationships

In the following sections, all components are specified and related to each other. However, collaboration diagrams that show more precisely how the components interact appear in the next chapter. This because these graphs are specific to the replication techniques implemented.

2.2.1 Database Service

This service is used as the local database storage on every replica. The service provides the following standard database functionality:

storage management This subcomponent offers persistent (stable) storage and retrieval of a constant number n of database items, with values that are arbitrary sequences of bytes of equal length. The items are indexed by an integer in the range $0 \ldots n-1$.

This component *executes* the data operations of transactions. When some read(X) is executed, the value of database item X is retrieved. When some write(X, a) is executed, the value of item X is set to a.

lock management This subcomponent provides a concurrency control technique, which is needed to prevent concurrent transactions from violating serializability. The technique chosen is *two phase locking*, which is presented in section 2.2.3. Basically, the service offers lock and unlock primitives for every database item. When these primitives are used correctly, they ensure mutual exclusion for conflicting operations.

transaction management This subcomponent executes the transaction operations begin, commit and abort according to their semantics defined in section 1.1.3.

The Database Service upholds the ACID properties.

There is one special requirement: this service should never decide by itself to abort a transaction. Under normal system conditions this is usually not a problem, but in the case of heavy loads or full disks, standard database systems tend to unilaterally abort transactions. If this requirement is not met, the replica at which the service resides is considered to have crashed. Recall that the replicated system is able to continue as long as the other replicas are still running.

2.2.2 Group Communication Service

This service provides communication primitives that allow the replicas to exchange messages that synchronize the database state. The advantage of using Group Communication primitives as opposed to the normal means of network communication available in operating systems, is that the primitives offer higher-level abstractions with properties that are desirable in the context of replication.

The exact specifications of the primitives depend on the needs of the replication techniques to be implemented. However, we explain the Atomic Broadcast primitive right now because it serves as the basis for the active replication techniques considered in this report. Why Atomic Broadcast is needed is explained in Chapter 3 (Active Replication Techniques).

Atomic Broadcast

Most operating systems allow processes to communicate across networks by message passing. They offer the send(m, p) and receive(m) primitives for sending a message m to process p and receiving a message m, respectively.

The Atomic Broadcast primitive is built on top of these standard send and receive primitives (see Figure 2.2) [GS97]. Informally, it allows processes to *broadcast* messages to a group of processes, while guaranteeing that all members of the group deliver all messages in the same order. Furthermore, it ensures that all (non-crashing) group members deliver every message that is broadcast. Finally, the primitive keeps working even when some of the group members crash[1].

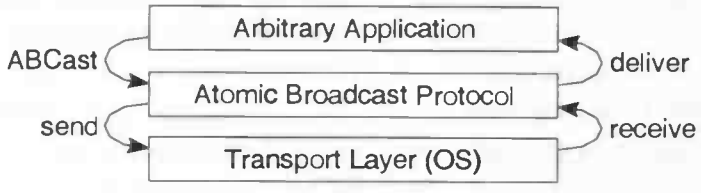

Figure 2.2: Protocol stack featuring Atomic Broadcast

Formally, if a process executes ABCast(m, G), it sends a copy of message m to all processes in the group G. deliver(m) is executed on each process in G to deliver the message m. The semantics of these operations are specified as follows:

Order Consider two messages m_1 and m_2, ABCast(m_1, G), ABCast(m_2, G) and $p_j, p_k \in G$. If p_j and p_k deliver m_1 and m_2, they do so in the same order.

Atomicity Consider ABCast(m, G). If $p \in G$ executes deliver(m), all correct[2] processes in G eventually execute deliver(m).

Termination Let ABCast(m, G) be executed by process p. If p is correct then every correct process in G eventually executes deliver(m).

In Figure 2.3, an example run of two ABCast invocations is shown where two sender processes each broadcast one message to a group of three member processes. The rectangles depict the delivery of the messages. As shown by the message arrows arriving at member 3, this member receives the messages in a **different** order than the other members. However, the messages are **delivered** in the **same** order.[3]

[1] A process that *crashes* ceases functioning forever.

[2] A *correct* process is a process that does not crash.

[3] To achieve this, an algorithm that implements Atomic Broadcast exchanges additional messages (not shown here) between the group members to agree on a certain order. In point-to-point networks, typical message amounts are 1 to 3 times the number of group members per ABCast invocation [UDS00].

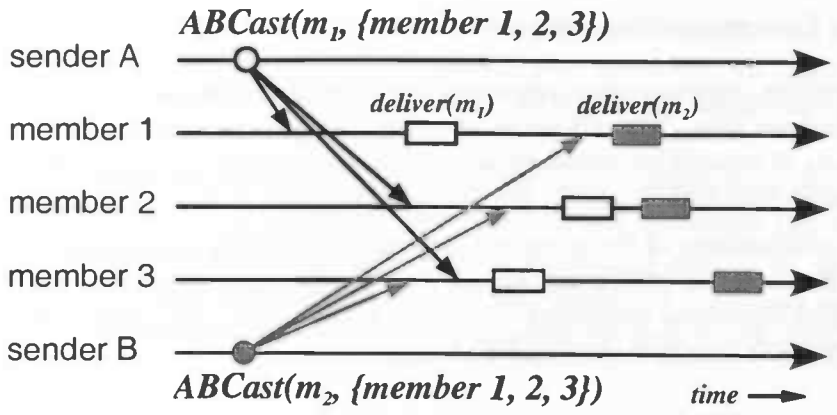

Figure 2.3: Example of Atomic Broadcast

2.2.3 Replication Manager

The Replication Manager is responsible for processing transactions and their operations. It consists of two subcomponents: the Scheduler, which accepts transactions submitted by clients and executes them on the database, and the Lock Manager, which provides concurrency control to the Scheduler. The two subcomponents are now discussed in detail.

Lock Manager This subcomponent provides concurrency control that the Scheduler uses to ensure serializability and durability. The concurrency control used in the Database Replication Prototype is strict two phase locking (strict 2PL).

Briefly, strict 2PL[4] entails the following: A *lock* is associated with every item in the database. An item's lock is used to ensure mutual exclusion of conflicting operations on the item. There exist two types of locks: read locks and write locks. The Lock Manager manages a data structure, called the lock table, and offers an interface to the Scheduler for acquiring and releasing locks on behalf of transactions.

Formally, when processing transaction operations, the Scheduler must enforce the following strict 2PL rules (adapted from [BHG87]):

- if a client issues a data operation o as part of transaction T on database item X then it must first request a read or write lock for X from the Lock Manager. The type of lock requested is in accordance with the type of X. The request is granted unless a conflicting lock is already held. In the first case, the lock is said to be *held* (also *acquired* or *obtained*), and o can be executed on the database. In the latter case, the execution of o is delayed until the lock becomes available and T is said to be *blocked* on the lock request for X.

  Two locks *conflict* if they lock the same database item, they are issued by different transactions, and at least one of them is a write lock.

- the read locks held by a given transaction can not be released until all its data operations have been executed. The write locks it holds can not be released until the transaction has been committed or aborted. After the transaction is committed or aborted, it releases all the locks it still holds. After a lock is released, it can be acquired by other transactions.

A problem of the strict 2PL approach is that deadlocks can occur. A *deadlock* is a situation in which each of two transactions is waiting for the other to release its locks: neither transaction can

[4]For a more detailed discussion and a correctness proof of (strict) 2PL, see [BHG87].

ever continue processing. In the literature, there exist different solutions to counter this problem. The solution that is adopted for the Database Replication Prototype is to always request locks in some fixed order, for example, in the order of the database item locations for which the locks are requested. This way, deadlocks are prevented from occurring [EN94].

Since locks are requested on behalf on operations, this means that clients must submit all operations of a transaction in a fixed order. While this imposes a severe limitation on the client, it is sufficient for comparing active replication techniques, and avoids the implementation complexity of other solutions such as deadlock detection and time out mechanisms. The motivation for this argument is given in section 3.5 because it requires an understanding of the compared replication techniques.

When implementing the functionality described above, the Lock Manager is supposed to build upon (reuse) the Database Service's lock management.

Scheduler The Scheduler processes the Operations (subsection 2.2.5) submitted to the Replication Manager by the Client (subsection 2.2.4). The Scheduler may delay operations, or reject them and forcefully abort the transaction instead. In the latter case, the Scheduler behaves as if the rejected operation was actually an abort operation. However, the Scheduler must explicitly inform the client about this "conversion". A client may always try to resubmit a forcefully aborted transaction at a later time.

The Scheduler is replication-aware: it ensures that the database state changes caused by write operations (of committed transactions) are consistently recorded in all databases at all replicas. In other words, the Scheduler must guarantee *one-copy serializability*. A Scheduler instance can directly modify the state of its local database. For this it uses the local Database Service's storage and transaction management. To modify the state of other databases, it needs to communicate with the Schedulers of other replicas using the Group Communication Service. How the Scheduler ensures *one-copy serializability* depends on the replication technique that is implemented.

Replication techniques

Using the components of the Replication Framework, we can now more precisely define the term replication technique. A *replication technique* is an algorithm that exactly describes how the components Scheduler, Lock Manager, Database System and Group Communication Service interact and operate to achieve a replicated database system. Many replication techniques have been proposed, with different levels of performance and varying levels of Isolation[5] (for a discussion, see [WPS+00b]).

For this project, we have focused on *eager* replication techniques (for a classification, see [WPS+00a]). They all achieve one-copy serializability, but are implemented differently and have varying performance characteristics. This replication framework hopes to accommodate most of the interesting techniques. As a proof of concept, and to obtain comparative performance measurements, two algorithms based on the Atomic Broadcast communication primitive are implemented (for a classification, see [WPS99]). They are discussed in the next chapter.

2.2.4 Client

In general, a Client generates a transaction workload: it models an application that uses the database. A Client creates Operations (see next section) and submits these to one of the Replication Managers.

[5] In this report, upholding the Isolation property is considered equivalent to ensuring serializability. Other interpretations of the property exist that do not require serializability: the isolation level is said to be lower [KA]. E.g., some techniques allow *lost updates*: consider two transactions T and U and the allowed history $r(X)_T$ $w(X)_U$ $w(X)_T$. The problem is that if the last $w(X)_T$ depends on the first $r(X)_T$, the intermediate value of X written by U is not considered by T and thus lost upon the last $w(X)_T$.

Operations that belong to the same transaction may not be submitted in parallel, but must be submitted one by one. In other words, the Client must always wait until it receives the result for the previous Operation before it submits the next one. In the rest of this report, the uncapitalized term *client* also refers to the Client defined in this section.

When submitting Operations, clients must respect the operation ordering rules imposed by the transaction mechanism (see section 1.1.3) and by the concurrency control mechanism (see section 2.2.3).

In the Database Replication Prototype, clients test the system and measure its performance. They randomly create Operations and submit these. The kind and number of Operations submitted is specified in relation to the transactions they belong to. We parameterize these transactions according to the following criteria:

Interactivity of transactions Transactions can be either *one-shot*, all operations are sent to the Replication Manager at once, or *interactive*, operations are sent one by one and may depend on the results of earlier operations.

The one-shot variant models the *stored procedure* that is popular in the database world. A stored procedure is a predefined program, stored in the database, that is executed when it is called by the client. A one-shot transaction contains exactly the operations that would be executed by such a stored procedure. The major advantages of stored procedures are that the client needs to send only one message to the database system, and that the transaction can be efficiently executed within the database. Both advantages are preserved by the one-shot transactions in our model.

Because one-shot transactions model stored procedures, the fact that they are a query or an update transaction is only known at the time the last data operation of the transaction is executed. Our model does not support *predeclared* queries: transactions for which the client announces that they are queries upon submission of the first operation (or the one-shot transaction).

Number of operations per transaction

This number is highly dependent on the type of application. For example, in the simple banking application of retrieving money from an ATM, only a few read and write operations are needed [TPC94], whereas in decision support systems, millions of database items may be read when compiling a company status report.

Percentage of commit transactions The number of times that a transaction ends with a commit operation instead of an abort operation.

One would maybe expect clients to always try to commit transactions because aborting a transaction would mean that the whole transaction was superfluous. However, this is only true for one-shot transactions. In the case of interactive transactions, clients may not know whether or not they are going to commit a transaction before having started processing it. For example, when an ATM application detects, by reading from the database, that a user is low on cash, it may decide to abort a money withdrawal transaction.

Percentage of queries The number of times that a transaction contains zero write operations.

Number of transactions submitted per time period

In a physical system, this number is bound by the maximum load the system can handle.

Fraction of write operations in update transactions The number of times that a data operation is a write operation instead of a read operation.

This fraction may not be chosen lower than (1/number of operations per transaction) because this would imply that the transactions are queries.

An additional parameter that is often used is the distribution of the database items accessed. To simplify things, we have chosen the uniform distribution. I.e., the transaction load produced by the client accesses each database item equally often.

2.2.5 Operations

This component models the data and transaction operations as defined in section 1.1. In some programming methodologies one would not define a component for the data that is manipulated by other components. However, since we follow the object oriented approach, we define Operations and their behaviours explicitly.

Operations are created and submitted by Clients (defined in the previous section). An Operation can request the locks it needs from the Lock Manager and execute itself using the Database Service. When the Scheduler processes an Operation, it delegates the locking and executing tasks to this Operation.

An Operation models one of the operations read, write, begin, commit and abort (as defined in section 1.1). In the case of a read or write, the Operation contains the locations to access and, in the latter case, the value to write.

An Operation can also model a one-shot transaction. In this case, it contains a sequence of Operations that forms a transaction. The processing of such an Operation entails the sequential processing of the Operations it contains.

In the rest of this report, we identify the abstract read, write, begin, commit and abort operations and their respective Operation instances. If an Operation represents a one-shot transaction, we denote this by using the term oneshottransaction. In the rest of the report we do not make the distinction between the capitalized "Operation" and the uncapitalized "operation".

Chapter 3

Active Replication Techniques

Within the Replication Framework defined in Chapter 2, replication techniques can be constructed. As a proof of concept, and to obtain comparative performance measurements, two *active replication techniques* based on Atomic Broadcast (see Figure 3.1) are implemented: *pessimistic* active replication and *optimistic* active replication. The implementations are part of the Database Replication Prototype.

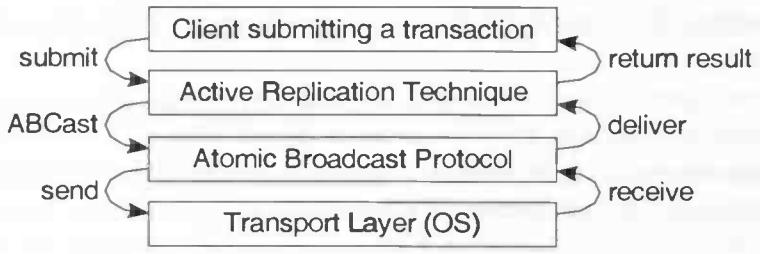

Figure 3.1: Protocol stack featuring active replication

For reasons that are explained shortly, the replication techniques require different scheduling and lock management behaviour. Therefore, the component Replication Manager contains technique specific algorithms. On the other hand, the components Database Service, Group Communication Service and Operations function the same independently of the techniques.

In this chapter, we first explain the concept of active replication. Following that, the techniques to be implemented are presented. Finally, the tradeoffs between both techniques are discussed by means of a qualitative comparison.

3.1 Active replication

The standard notion of active replication is that clients submit a transaction by Atomic Broadcasting all its operations to all replicas. The replicas contain a centralized database system which deterministically processes the database modifications (updates) in the same order. This way, the database state remains consistent at all replicas at all times.

In *pessimistic* active replication [SR96], replication is accomplished by distributing all submitted transactions to all replicas using the Atomic Broadcast primitive. This technique is also called immediate update replication.

The *optimistic* active replication [PGS99] tries to avoid some of the communication overhead of pessimistic active replication. This is done by Atomic Broadcasting only the write operations of transactions to all replicas, avoiding the needless broadcasting and processing of read operations by all replicas. Instead, read operations are only processed by one replica. This technique is also called deferred update

replication.

Active replication techniques are sometimes referred to as the *database state machine approach* [PGS99], since the database system is seen as a state machine that deterministically computes the next database state as a function of the previous state and a given operation. As read operations do not affect the state, they need not be processed on every replica.

3.1.1 Active replication within the Replication Framework

To accommodate different techniques in the same framework, we have diverged a bit from the standard notion of active replication. In the Database Replication Prototype, clients only send their operations to one replica. It is the task of this replica to distribute the operations to all replicas (as needed) using Atomic Broadcast.[1] Another advantage of this approach is that it achieves distribution transparency (see section 1.2). The Scheduler running on the replica contacted by the client is called the *originating Scheduler* in the following sections.

Hereafter, the mentioned replication techniques are presented in detail and connected to the components of the Replication Framework.

3.2 Pessimistic active replication

In Figure 3.2, it is shown how the framework components collaborate in a typical run of the pessimistic replication technique. The diagram shows a system that contains three replicas. The steps taken when processing one particular Operation are numbered and named. These steps are referred to in the description of the technique given in the remainder of this section.

While steps 3 to 7 are the same on every replica, these are only detailed for the third one in the diagram. As explained hereafter, steps 5 and 6 are only applicable to data operations. Actually, there exists a part of the Group Communication Service on every replica, but because this is logically one system wide component, it is drawn only once.

Now follows the description of the technique. The numbers in parentheses refer to Figure 3.2.

When submitting a transaction, a Client connects to an arbitrary Replication Manager (1). Its Scheduler immediately passes the Operations on to the Group Communication Service that broadcasts them to all Replicated Schedulers using the Atomic Broadcast primitive (2).

When an Operation is delivered to the Scheduler (3), the Operation is processed in a deterministic way according to its type (4):

data operation the Operation requests the lock it needs from the Lock Manager and waits until it obtains the lock (5, 6). The Lock Manager deterministically grants the request or queues it until it can be deterministically granted. After granting or queuing the request, the Group Communication Service is allowed to deliver the next Operation.

When the Operation obtains its locks, it executes itself on the database by submitting the desired data operation to the Database Service (7). Upon receiving the service's response, the Operation returns it to the Scheduler, which in turn sends it back to the originating Scheduler. Note that for the response message, the standard send primitive is sufficient. We assume that this primitive is provided by the Group Communication System.

Upon receiving the first response from any Scheduler, the originating Scheduler forwards the response to the Client. The other responses are discarded since they are the same as the first response

[1] In this approach, the replica to which a client connects acts as a *proxy* that handles the task of Atomic Broadcasting the Operations instead of the client itself.

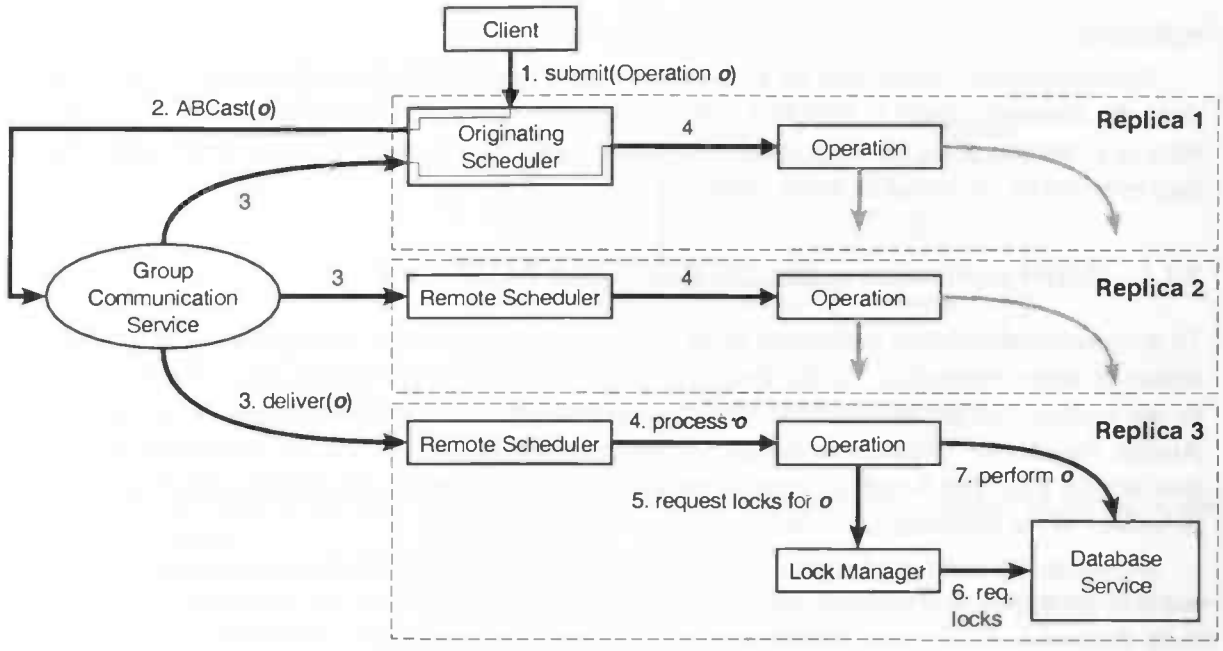

Figure 3.2: Collaboration diagram for pessimistic replication

(because of the deterministic processing at every replica).

transaction operation the Operation is executed immediately. It asks the Database Service to carry out the corresponding action and waits for the response (7). After receiving the response, the next Operation may be delivered. The response is returned to the Client like in the previous case.

oneshottransaction operation the Operations this transaction contains are deterministically processed in sequence, as described before. The transaction can be deterministically queued when an Operation's lock is not available. When deterministic processing or queuing is guaranteed, the next Operation may be delivered. When the transaction is committed or aborted, the result of the commit (or abort) operation is returned to the Client. The results of other Operations are not returned to the Client.

3.3 Optimistic active replication

The basic idea of optimistic active replication is that the operations of a transaction are executed locally on one replica until the commit operation is reached. Then the updates (write operations) of the transaction are collected and Atomic Broadcast to all replicas. Upon delivery of the updates, a certification test[2] is performed at every replica to check if committing the transaction does not violate serializability. If serializability cannot be guaranteed, the transaction is aborted on every replica, otherwise, the updates are applied on every replica.

The outcome of the certification of a given transaction T is the same on all replicas because the certification test only takes into account those transactions that were previously delivered. Because of the properties of Atomic Broadcast, all replicas are bound to have seen exactly the same delivered transactions before T is delivered. Therefore, they can deterministically make the same decision as to the abort or commit of T. The result is that the database state at every replica is guaranteed to remain identical.

[2]How the certification test works is explained in section 3.3.1. However, the proof showing that it ensures serializability is beyond the scope of this report. It can be found in [PGS99].

Before describing optimistic active replication in detail, we note that in this technique, a *state* is associated with each submitted transaction. This state can be: **executing, committing, committed** or **aborted**. The corresponding state transition diagram is shown in Figure 3.3. All states and transitions are explained in this section.

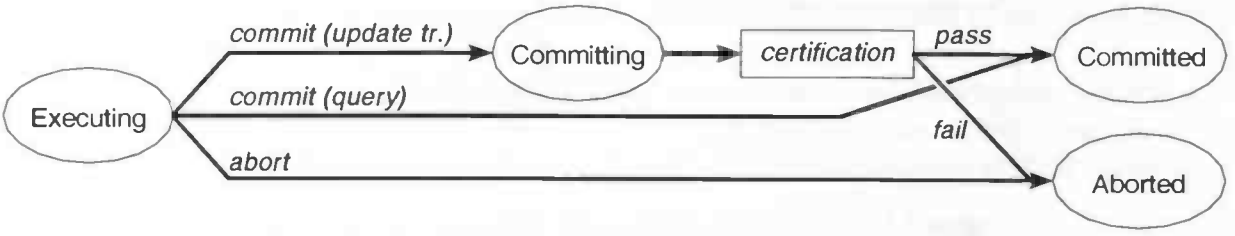

Figure 3.3: Transaction states and transitions for optimistic active replication

In Figure 3.4, it is shown how the framework components collaborate in a run of the optimistic replication technique. In this figure, it is assumed that an update transaction is submitted. The diagram shows a system that contains three replicas. The main steps taken when processing an Operation are numbered and named. All steps are explained in the remainder of this section.

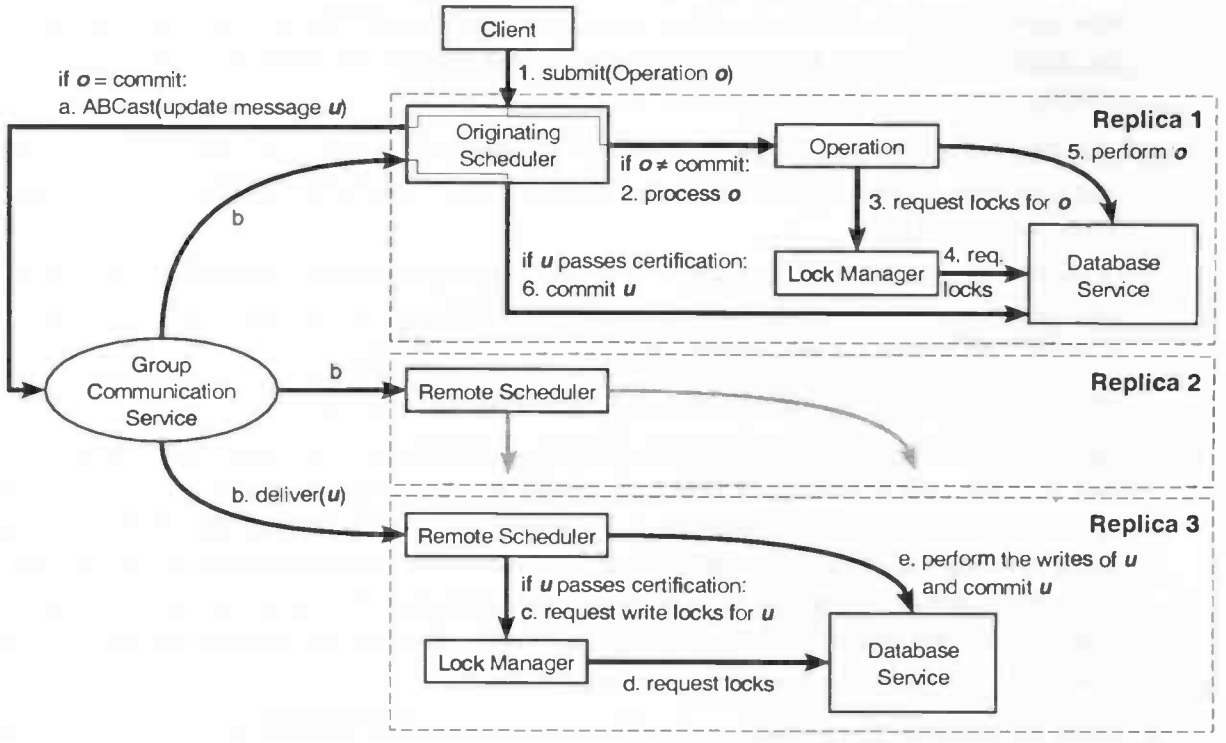

Figure 3.4: Collaboration diagram for optimistic replication (update transactions)

If a query is submitted to a given replica, the processing is only performed locally because a query contains only read operations. The collaboration diagram for this case is depicted in Figure 3.5. As is clearly shown, the Group Communication Service and the other replicas are not involved.

The algorithm for optimistic active replication entails the following procedure (adapted from [PGS99]). The numbers and letters in parentheses refer to Figures 3.4 and 3.5.

I. When submitting a transaction, a Client connects to an arbitrary Replication Manager (1). Its Scheduler starts processing the Operations locally using the Database Service until the last Operation is reached (2-5). During this local processing, the transaction is in the **executing** state. If

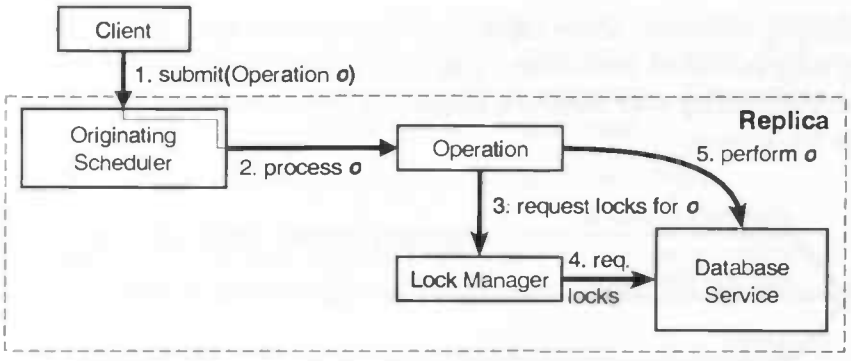

Figure 3.5: Collaboration diagram for optimistic replication (queries)

the last operation is an abort operation, the transaction is immediately aborted (it passes to the **aborted** state).

II. If the last operation is a commit operation, and the transaction is a query, it is committed immediately (5). Otherwise, the transaction passes to the **committing** state. First, it releases all its read locks. Then, all its updates are collected in one message and are broadcast to all Replicated Schedulers using the Atomic Broadcast primitive offered by the Group Communication Service (a). More precisely, this update message contains the readset, the writeset and the values to be written.

III. Upon delivery of the update message (b), each Scheduler deterministically *certifies* the **committing** transaction to test if serializability can be guaranteed. If this is the case, the transaction passes to the **committed** state.

Otherwise, the test fails and the transaction passes to the **aborted** state. On the originating Scheduler, the transaction is aborted. On the other Schedulers, the update message is simply not applied. After the certification the next update message may be delivered.

How the certification test works is discussed in subsection 3.3.1.

IV. The result of the certification, commit or abort, is returned to the originating Scheduler which returns the first result it receives to the Client. At this point, we slightly diverge from what is stated in [PGS99] (adapted to our terminology): The Client receives the result of the certification when the **originating Scheduler** has certified the transaction. The difference is that in our implementation, the Client does not have to wait for the originating Scheduler to perform the certification. Since the outcome of the certification is deterministically defined, the result of any Scheduler can be passed to the Client.

V. When a transaction has reached the **committed** state, its update message proceeds in one of two ways, depending on the Scheduler where it is delivered. Assume that the update message is delivered on a given Scheduler. Then:

- if the update message originated from this Scheduler, the transaction is simply committed because the write operations have already been performed on this replica (6 on Replica 1).
- if the update message originated from a remote Scheduler, a begin operation is executed at the Database Service to start the transaction. Then, write locks for all write operations contained in the update message are requested from the Lock Manager at the same time (c, d on Replicas 1 and 2). The Lock Manager deterministically grants the request or queues it until it can be deterministically granted. If the queue contains requests that ask for conflicting

locks, the Lock Manager grants these in the order the corresponding update messages were delivered.

Three cases need to be considered when granting the request:

- if a locally **executing** transaction holds a lock that conflicts with one of the locks that the update message requests, the Lock Manager aborts the local transaction so that the lock can be granted to the update message.
- if a transaction T that executed locally has reached the **committing** state (so it is sure that its update message will be delivered sometime after the current update message) **and** holds a conflicting write lock, the write operation is postponed until the certification of T:
  * if T is aborted, the write operation is immediately applied. Observe that because T held the write lock for the operation before aborting, the write operation can always be performed without locking conflicts.
  * if T passes to the **committing** state, the write operation can be forgotten since the value that T wrote is the most up to date value.
- if a previously delivered update message that passed to the **committed** state is holding a conflicting write lock, the current update message is deterministically queued until the lock is available.

After granting or queuing the lock request, the next update message may be delivered.

When the update message obtains its locks, it performs itself on the database by submitting its write operations to the Database Service (e on Replicas 1 and 2). Following that, the commit operation is submitted.

3.3.1 Certification

A transaction passes certification and enters the **committed** state if it does not *conflict* with any previously committed transaction. Otherwise the certification fails and the transaction passes to the **aborted** state.

We do not formally define the notion of conflicting transactions in this report (for a formal definition, see [PGS99]). Informally, if two transactions conflict, committing both of them violates serializability. The certification mechanism detects these conflicts and makes sure that one of the conflicting transactions is aborted, thus ensuring serializability.

In the rest of this subsection we describe how the certification test is performed.

Assume that a transaction T_1, originally processed at replica p_1, is to be certified. This means that we must check whether T_1 conflicts with any previously committed transaction T_2, originally processed at p_2 ($p_1 \neq p_2$). When checking for a conflict between T_1 and T_2, and deciding whether or not to certify T_1, three cases can be distinguished:

1. Before T_1 started **executing**, T_2 passed to the **committed** state (so it passed certification) at p_1. In this case, T_1 passes the certification test. This because there is no interleaving of T_1 and T_2, thus certifying T_1 will not violate serializability.

2. When T_1 was **executing**, T_2 passed to the **committed** state at p_1. Because of the local concurrency control at p_1 (strict two phase locking) this execution can never violate serializability. So T_1 passes the certification test.

3. When T_1 was in the **committing** state, T_2 passed to the **committed** state at p_1. This means that T_1 was delivered after T_2. Whether or not T_1 passes the certification depends on the data operations that T_1 and T_2 have performed:

- if the writeset of T_2 and the readset of T_1 overlap, T_1 fails the certification test. This because the overlap means that in the context of T_1, the (old) value of some database item X was read that has been subsequently overwritten when T_2 committed. Committing T_1 after T_2 would be a violation of one-copy serializability, so T_1 is aborted.
- otherwise, T_1 can be committed without violating serializability.

Observe that when T_1 and T_2 were originally processed on the same replica (i.e., $p_1 = p_2$), the transactions can never conflict because the local concurrency control mechanism prevents this.

3.3.2 Example execution

In Figure 3.6, an example execution of the optimistic replication technique is shown. Replica p_1 locally processes the update transaction T_1: begin r(X) w(X, a) commit. Replica p_2 locally processes the update transaction T_2: begin r(X) w(X, b) w(Y, c) commit.

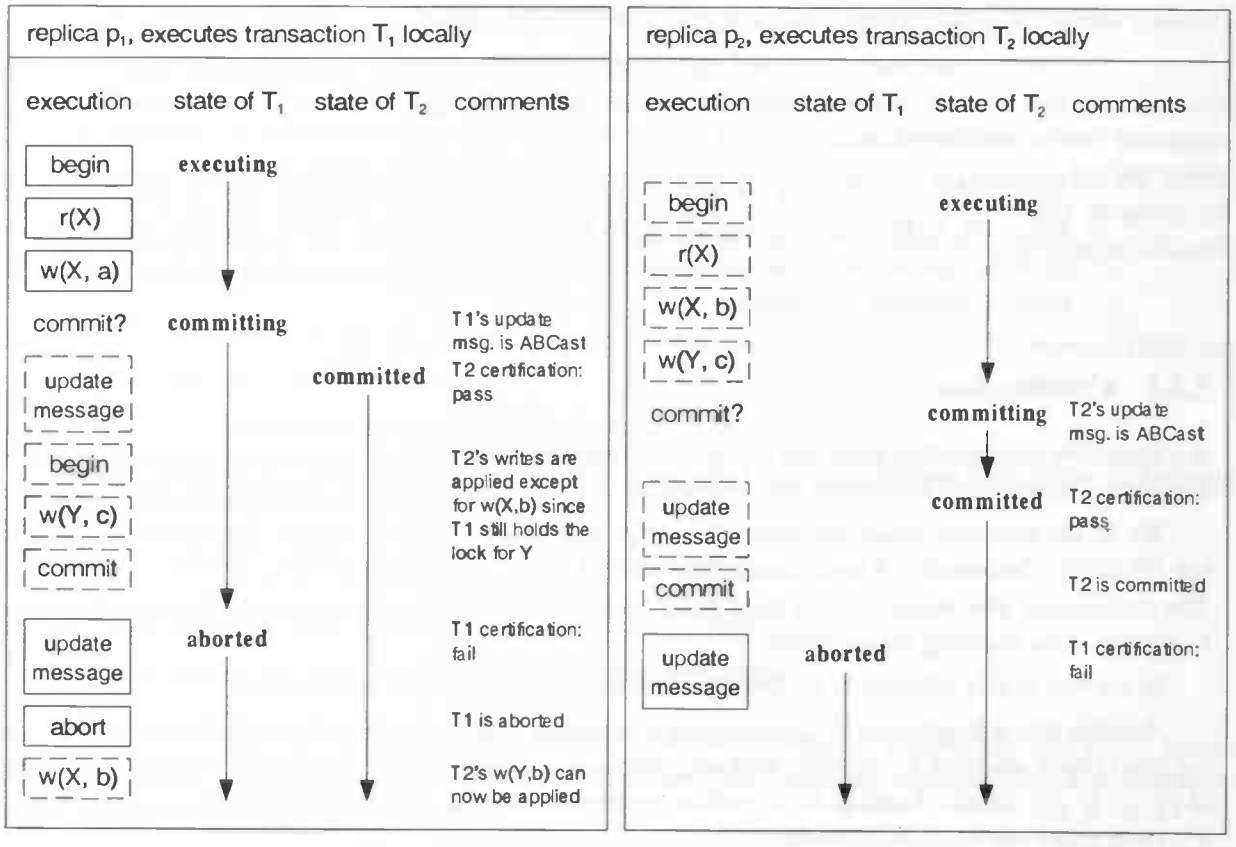

Figure 3.6: Example execution of the optimistic replication technique

As shown in the picture, the Atomic Broadcast algorithm decides that the update message of T_2 is delivered before that of T_1. T_2 then passes the certification test because no transactions have passed certification yet. On p_2, T_2 is simply committed. On p_1, T_2's writes are applied, but since T_1 still holds the lock for X, the w(X, b) operation can not be applied yet. The operation is delayed until the outcome of the certification of T_1 is known.

When the update message for T_1 is delivered, T_1 cannot pass the certification test **on both replicas**. The transaction states registered at p_1, show that T_2 passes to the **committed** state when T_1 is in the **committing** state. Subsequently, it must be checked whether the readset of T_1 and the writeset of T_2 are disjoint. This is not the case: database item X is contained in both sets. The result is that T_1 is aborted

on p_1. Since this means that the lock for X becomes available again, the delayed w(X, b) is applied. On p_2, the update message is discarded.

From the example it becomes clear that the certification test needs information about the events happened on other replicas to be able to make the same decision on every replica. More precisely, to certify T_1, the certification algorithm at p_2 must know if T_2 was already certified at p_1 when T_1 passed to the **committing** state. This information is obtained as follows:

- The delivery of every update message is tagged with a sequence number (starting at 1). By the Order property of Atomic Broadcast, this number is the same on every replica for a particular update message.
- When an update message is ABCast to all replicas, the sequence number of the update message that has most recently passed certification is piggy-backed.

The above steps are sufficient to perform the certification test, as we now demonstrate using the example.

When p_1 ABCasts the update message for T_1, there is no most recently certified transaction, so the value 0 is piggy-backed. When the update message for T_2 is delivered on p_2, it is tagged with sequence number 1. Similarly, the update message for T_1 is tagged with number 2. Using this information, the certification test at p_2 detects that T_1 passed to the **committing** state before T_2 was certified at p_1. This because T_2 has a sequence number (1), that is **higher** than the value that was piggy-backed (0).

3.4 ACID properties

Since clients may connect to any of the replicas that form the replicated database system, it is not immediately obvious that the ACID properties are upheld by the system as a whole. In this section, proof sketches are given that show that the ACID properties are satisfied for the pessimistic replication technique. The correctness proof for the optimistic replication technique requires a more thorough introduction to serializability theory than provided in this report. We therefore do not present it here, but refer to [PGS99].

First, the proof sketch for the Durability property given, it applies to both techniques. Thereafter, the Atomicity and Isolation properties are treated in the case of the pessimistic technique.

Durability Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

In centralized database systems, it is assumed that the database state is stored on some durable medium that never fails (e.g., a hard disk). The failed system can then recover by retrieving the database state from the durable medium and continuing from that state. These systems do not guarantee Durability when the durable medium fails.

In replicated database systems, Durability can be enforced when it is assumed that during the lifetime of the system, always at least one replica is running.[3]

Theorem: The Durability property holds for both pessimistic active replication and optimistic active replication.

Proof (sketch): The property trivially holds for queries and for transactions that end with an abort operation, because these do not modify the database state. For update transactions that end with a commit operation, the property is proven as follows.

[3]Section 3.6.2 explains how the system can recover when all replicas crash at the same time.

By definition, the Database Service at every replica process satisfies the Durability property. From the Atomicity property of Atomic Broadcast it follows that if a commit operation is delivered on one replica, it is delivered on all replicas. To enforce Durability, this commit operation must have the same outcome (commit or abort) on all replicas.[4] This is guaranteed by the determinism constraint on the Database Service, which completes the proof.

In the next two subsections, the Atomicity and Isolation properties are treated separately for each technique.

3.4.1 Pessimistic active replication

Atomicity A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

Theorem: The Atomicity property holds for pessimistic active replication.

Proof (sketch): Similarly to the proof of the Durability property, we only consider update transactions that try to commit.

Since Atomicity is assumed for the Database Service, we only need to check that for a certain transaction, all write operations have been processed by all replicas before its commit operation is processed. Remember that clients submit operations one by one, always waiting for the result returned by the previous operation before submitting the next one. By the Order property of Atomic Broadcast, all replicas process these operations before the commit operation is ABCast and delivered, which was to be proved.

Isolation A transaction should not make its updates (i.e., the effect of its write operations) visible to other transactions until it is committed.

Theorem: The Isolation property holds for pessimistic active replication.

Proof (sketch): As we show next, the property is satisfied by this technique because the technique is correct in respect to one-copy serializability.

Observe that all operations are processed on every replica

- in the same order (by the Atomic Broadcast primitive), and
- in exactly the same way (by the determinism constraint).

This means that the replicas always behave identically. Since it is assumed that every local Database Service guarantees serializability, it follows that one-copy serializability is guaranteed by the replicated database system.

3.5 Deadlocks

As noted in section 2.2.3, deadlocks do not occur in the Replication Framework because locks are always requested in a fixed order. The absence of deadlocks makes the Database Replication Prototype simpler to implement. However, the order restriction has a considerable impact on the Client, because it needs to submit operations in the order of the locations these operations refer to. In real-world applications, this may not be feasible for two reasons:

[4] To see this, consider that one diverging replica aborts the transaction and all others commit. Subsequently, the client is told that the transaction has committed. Then, all replicas except the diverging one crash. The system is still running, but has violated the Durability property because it "lost" the committed transaction.

- the Client usually does not know how database items are ordered. For example, if SQL is used to access the database, the exact locations of database items are not visible to the Client.
- even if we assume that the Client somehow knows the order in which to submit the operations, a program may want to access some items out of order to perform certain functions. One solution for this is to let the Client predeclare the items it intends to access. Then, the Lock Manager can already obtain locks for these items in the correct order at the beginning of the transaction. However, this approach is also not feasible because the Client often does not know in advance which items it is going to access in a given transaction.

To overcome these limitations, the order restriction needs to be lifted. In principle, the Replication Framework can be easily adapted to allow operations to be submitted in any order. However, this may result in deadlocks. The way to resolve deadlocks is to abort transactions involved in a deadlock until the remaining transactions can progress again. We now outline how each technique can be adapted to resolve deadlocks.

Pessimistic active replication. After every n operations delivered, the Scheduler stops delivering operations. It waits until all transactions are blocked for a lock **or** or waiting for an operation to be delivered. In other words, the Scheduler waits until all operations have stopped executing. If the execution of an operation leads to the release of locks, the Lock Manager deterministically grants these to transactions that were previously blocked for these locks. The Scheduler waits also for these transactions to finish executing operations.

After this, the Lock Manager runs a *deadlock detection algorithm*[5] [EN94]. This algorithm finds the transactions that are involved in a deadlock and deterministically picks and aborts some of them until the deadlock ceases to exist. Observe that all replicas abort exactly the same transactions at this point because each one has delivered and executed exactly the same set of operations before it runs the deadlock detection algorithm. How this algorithm is implemented is beyond the scope of this report, but any standard algorithm can be used because it needs to consider only those transactions on the local replica (i.e., a distributed deadlock [EN94] detection algorithm is not needed).

As to the choice of n: n is preferably large so that the Scheduler and Lock Manager do not spend too much time waiting and searching for deadlocks. On the other hand n must be small enough for deadlocks to be detected in a timely manner. Since the exact value of n depends on the actual system configuration and the kind of transactions processed, we do not further discuss it here.

Optimistic active replication. We first consider transactions in the **executing** state. These run locally and can deadlock with other **executing** transactions, but not with transactions in other states as we explain later. For resolving these local deadlocks, any standard (nondeterministic) deadlock detection or prevention algorithm can be periodically run that only takes locally **executing** transactions into account.

One more adaptation needs to be made to this technique: when an update message requests write locks, it must do so in a fixed order. We now show that with this change, transactions in the **committing** or **committed** states can not be involved in a deadlock situation.

When a transaction T passes to the **committing** state, its update message m is bound to be delivered and certified. After the certification, T passes to either the **aborted** or the **committed** state. In the former case, T's write locks are released on the originating replica and m is ignored on the other replicas, so T can not be in a deadlock situation.

[5] Actually, a deadlock prevention algorithm [EN94] could be used just as well, as long as it acts in a deterministic way. For example, a timestamp based algorithm that aborts transactions for which the last operation has been delivered more than some fixed amount of delivers ago.

In the latter case, if m is delivered on its originating replica, T is committed and releases its locks. If m is delivered on another replica, it requests its write locks in the order of the locations of the corresponding database items.[6] Then T cannot deadlock because:

- any **executing** transaction that holds a lock needed by T is aborted.
- when a lock that T needs is held by some **committing** transaction U, T does not request the lock (it delays the write until the certification outcome of U is known).
- when a lock is held by some transaction U that passed to the **committed** state before T did, T is blocked. But, since locks for update messages are always requested in a fixed order, T does not deadlock with U.

Discussion. We have sketched how both techniques can be extended to deal with deadlocks. However, the adopted solutions are quite different. In the pessimistic technique, the Scheduler needs to wait now and then for "things to settle down". Only this way the deadlock detection algorithm will deterministically find the same deadlocked transactions on every replica. The optimistic technique is affected in two ways: the ordering requirement on the write locks of update messages and the local deadlock detection or prevention algorithm.

It is difficult to predict how these changes affect the performance of the adapted techniques when many transactions interfere. However, when few transactions interfere, the performance degradation will be negligible. This because deadlock checks can be done very infrequently without risking that many transactions are waiting because of deadlock. These infrequent checks incur little overhead for both techniques. When comparing the (unadapted) replication techniques in the rest of this report, we consider scenarios with relatively little transaction interference. Therefore, the results obtained are also applicable for the adapted techniques as long as the transaction interference is low.

3.6 Some reliability aspects

As noted in section 1.2.1, a major advantage of replicated database systems is their ability to tolerate replica crashes. In this section, we briefly discuss how crashes affect the replication techniques presented in this chapter.

In section 3.4 it was shown that the ACID properties are upheld when at least one replica is always running. Now, we explicitly discuss a scenario in which a crash occurs. A crash means that one replica suddenly stops processing. Observe that in every technique, there is an ABCast invocation and corresponding delivery before **any** update transaction is committed. Imagine that the replica crashes just after it has committed a transaction and informed the client. This transaction is bound to be delivered and committed on all replicas, guaranteeing Durability.

The Atomicity and Isolation properties are also satisfied in this scenario: The crashed replica was processing transactions correctly until the moment it crashed. Because all remaining correct replicas are bound to deliver and deterministically process these transactions, the properties are upheld.

The presented replication techniques fail to solve two problems: the addition of replicas to the system, and the crashing of all replicas. Also, it has not yet been discussed what happens to a client that is submitting a transaction to a crashing replica. We discuss these issues in the next subsections.

[6] Since all needed locks are known in advance and since the Lock Manager is a low level component that knows the locations of the database items, this order restriction is easily met.

3.6.1 Adding replicas to a running system

While the addition of (previously crashed) replicas is not possible in the Database Replication Prototype designed in this report, this could be added easily. The following requirements would need to be satisfied:

dynamic groups The Group Communication Service should have the possibility to dynamically change the configuration of the group of replicas at runtime. This way, the added replica can perform ABCast invocations and is able to deliver the messages sent to the group.

state transfer mechanism When the added replica joins the group, its database state is not synchronized with the state of the other replicas in the group. Therefore, there should be a mechanism that allows the complete database state to be transferred from (one of) the running replicas to the newly added one.

The dynamic groups requirement is solved by adding a *Group Membership Service* to the Group Communication Service. Basically, this service makes sure that the Atomic Broadcast algorithm knows the different group configurations that exist during the lifetime of the system.

The state transfer mechanism is a harder problem, because transferring the state may involve sending large amounts data over the network. An additional problem is that for the state to be consistent, the sending replica may not modify its state during the transfer.[7] Thus, the mechanism may lead to high network loads and the delaying of clients connected to the sending replica.

If the replica that joins was previously crashed, the state transfer can be done more efficiently. The Database System at the joining replica has durably stored the transactions that were committed before the crash, so only the database modifications since the crash would need to be transferred. Depending on the amount of modifications between the crashing and the joining of the replica, this "delta" state transfer can be (much) smaller than a full state transfer.

3.6.2 What if all replicas crash?

If all replicas crash (i.e., zero replicas are running), the replicated database system stops functioning forever. This because the *crash-stop* model is assumed: if a process crashes, it stops altogether and never recovers.

A solution to prevent this situation, is to allow new replicas to be added to the system, as described in the previous subsection. However, this only works when there is at least one replica still running.

In case the system has stopped, the database states that were correct before the crash are still durably stored on the crashed replicas. One might try to resurrect (one of) these replicas. To uphold the Durability property, the replica with the most up to date state must be found and resurrected first. Only then, the other replicas can be added to the system following the standard state transfer procedures.

Note that the first replica to be resurrected can be found as follows: All replicas that were running before the crash are restarted and first agree on the sequence number of the last message that was delivered before the crash. The replica with the highest sequence number is the first one that must be added to the system. Of course, if not all replicas can be restarted, Durability can not be guaranteed.

3.6.3 Clients facing replica crashes

When the replica to which a client is connected crashes, the client must find a new replica and resubmit the transaction. To detect the crash of the replica, the client can use a failure detection mechanism (for

[7] A way to shorten the delay of clients is to make an in-memory copy of the state and transferring this copy afterwards. The sending replica can continue serving replicas immediately after making the copy, which is usually faster than waiting until the state has been transferred over the network [DMS98].

example, a mechanism based on timeouts). Next we demonstrate that such a mechanism introduces a problem.

The problem: update transactions committed twice

In the case of slow network connections, the failure detection mechanism may falsely indicate that a replica has crashed. It could be that the transaction was committed, but that the replica did not respond fast enough. Another possibility is that in the case of a commit operation, the replica crashed after ABCasting the commit. It is possible that the transaction has been committed on every correct replica, but the crashed replica could not inform the client of this result.

In both cases, the client can decide to resubmit an update transaction to a different replica, even though the update transaction was already committed. Often, this behaviour is undesirable. For example, the transaction of transferring a certain amount of money from one bank account to another must not be applied twice without notice to the client.

A solution: attaching ID numbers to transactions

To prevent the problem outlined before, the client attaches a unique transaction sequence number to every submitted operation. Such a number can be constructed by combining, for example, the client's machine ID, its process ID and a local sequence number. When a client resubmits a transaction, it uses the **same** sequence number as used for the first submission.

It is the task of the replicas to detect that an already committed update transaction is resubmitted. Aborted transactions and queries need not be detected because retrying or executing these multiple times is not considered harmful. We only sketch the procedure for pessimistic active replication to keep this section from becoming too detailed:

- Every replica manages a list containing the IDs of the most recently delivered and committed update transaction on a per client basis. This list is updated by piggy-backing the sequence number of a transaction on every ABCasted commit operation. The list is thus the same on every replica.
- Using the list, each replica can detect if an operation is delivered on behalf of an update transaction that has already been committed. If this is the case, the operation is discarded and the transaction being resubmitted is aborted. Then, the replica informs the client of the duplication and of the fact that the transaction has already been committed.

3.7 Qualitative comparison of pessimistic and optimistic replication

In this section, the active replication techniques presented are qualitatively compared. As discussed in section 1.2.1, a replicated database system provides a higher degree of availability[8] than a centralized database system. However, the degree of availability provided by the two active replication techniques is the same, since they both tolerate the crashing of all replicas except one. Therefore, only their expected performance characteristics are compared in this section.

The main performance criterion considered in this report is the mean response time per transaction, given a fixed transaction load. In Chapter 5, this criterion is explained in more detail. In this section, we qualitatively discuss the following five factors that could influence the relative performance of the techniques:

- number of ABCast invocations per transaction

[8] In terms of tolerated machine crashes.

- amount of processing in addition to the processing performed on a centralized database system
- load balancing
- determinism requirements
- abort rate

In the next subsections, the performance factors above are explained one by one. In every subsection, the tradeoff between pessimistic active replication and optimistic active replication are be outlined in respect to each factor. Then follows a discussion that takes all factors into account. Finally, some rules of thumb are given to help choosing between the techniques given a particular situation. For a quantitative comparison, we refer to Chapter 5.

3.7.1 Number of ABCast invocations per transaction

This subsection discusses the number of ABCast messages sent on behalf of a single transaction.

In today's computer systems, the speed of network communication is usually lower than the processor speed, so this factor may have a significant impact on the performance. We do not consider the message size because this does not have a large impact on (local area) network performance.

Pessimistic active replication. In this technique, the number of ABCast messages sent per transaction depends on the **interactivity** of the transaction:

- **interactive** transactions are submitted and ABCast operation by operation, so the number of ABCast invocations per transaction is equal to the number of operations of the transaction
- **one-shot** transactions are submitted and ABCast at once, so there is 1 ABCast invocation per transaction

Optimistic active replication. In this technique, the number of ABCast messages sent per transaction depends on the **type of operations** that the transaction contains:

- **queries** are executed locally, so there is no ABCast invocation at all
- **update transactions** are first processed locally and then ABCast using an update message, so there is 1 ABCast invocation per transaction

3.7.2 Amount of processing in addition to the processing performed on a centralized database system

This subsection discusses the overhead incurred by the replication technique, which may slow down the processing of transactions.

Pessimistic active replication. Submitted operations are immediately ABCast by the replica that was contacted by the client. Upon delivery, the operations are executed by the Database System without additional processing. Therefore, there is no processing overhead compared to a centralized database system.

Optimistic active replication. Submitted update transactions are first processed locally and then AB-Cast. Upon delivery, a certification test is performed. This test requires additional processing because the update message needs to be compared to the write operations of recently committed transactions.

3.7.3 Load balancing

This subsection discusses the amount of operation and transaction processing that is performed independently on the replicas. This factor is important for situations of high loads, where performance may be increased by balancing (parallelizing) some of the processing load across the replicas instead of replicating all processing on all replicas. We assume in this subsection that the clients are fairly distributed over the replicas.

Pessimistic active replication. The load is never[9] balanced, because there is no independent decentralized processing. All replicas process exactly the same operations and transactions.

Optimistic active replication. Queries and transactions that request abort are decentrally processed at one replica only. The same holds for the read operations of update transactions. Only the write operations of transactions for which the client requests commit are processed at every replica (if the transaction passes certification). So this technique offers considerable load balancing.

3.7.4 Determinism requirements

This subsection discusses the determinism required to guarantee that replicas are always identical.

Determinism can be achieved by constraining the concurrent execution of transactions, thus making sure that (arbitrary) scheduling decisions do not cause violation of the determinism requirement.[10] Decreasing the amount of concurrency can have an adverse effect on transaction processing performance because the resources of the system can be utilized less efficiently.

Pessimistic active replication. Every operation is processed deterministically.

Optimistic active replication. Similarly to the previous factor, only the write operations of transactions that end with a commit are processed deterministically at every replica.

3.7.5 Abort rate

This subsection discusses the probability that a transaction is aborted even though a client submitted a **commit** operation.

These forced aborts happen if the replicated database system cannot guarantee one-copy serializability unless the transaction is aborted. The client that submitted a forcefully aborted transaction will try to resubmit it. So from the client's point of view, the response time for the transaction is slower because it needs to wait longer before the transaction is finally committed. In general, a high abort rate leads to slower average response times.

Due to the locking and deadlock prevention schemes adopted (see section 2.2.3), deadlocks do not occur because of serializability problems local to the replica (i.e., not at the Database Service's level).

[9] One could also say that the load is perfectly balanced in this case because the same load is processed at every replica. Since this is not at all beneficiary for performance, we decided not to call this load balancing.

[10] For example, in the Replication Framework, the Lock Manager, in collaboration with the Scheduler, grants locks in such a way that determinism is guaranteed where needed.

Pessimistic active replication. Forced aborts do not occur because all operations are ABCast to all replicas and then directly submitted to every local Database System. As explained in the previous paragraph, this system does not forcefully abort transactions.

Optimistic active replication. Because transactions start processing independently on one replica, some conflicts between transactions are only detected at certification time. As described in subsection 3.3.1, the certification test forcefully aborts transactions when one-copy serializability can not be guaranteed.

Furthermore, when an update message does pass certification, locally running transactions may be forcefully aborted at the time the Lock Manager grants locks to the update message (see section 3.3, step V).

3.7.6 Discussion

In the previous sections, the two techniques were compared considering one factor at a time only. The question arises which technique has the best performance when all factors are added up.

There is no clear-cut answer to this question, because the answer depends on the characteristics of the transaction load that is processed by the system. In subsection 2.2.4, the transaction load was parameterized from the client's point of view. We now consider these parameters in respect to the replication techniques and the performance influencing factors that were introduced.

Interactivity of transactions In the case of **interactive transactions**, the pessimistic technique uses an amount of ABCast invocations that is linear in respect to the number of operations per transaction. The optimistic technique does a significantly better job because it uses only 0 or 1 ABCast invocation. Because networks tend to be slow compared to CPUs, the pessimistic technique is not considered feasible for interactive transactions [WPS+00b].

In the case of **one-shot transactions**, there is not much difference: both techniques use a constant number of ABCast invocations.

Number of operations per transaction This parameter is orthogonal to both techniques in the case of one-shot transactions. For interactive transactions, it can make a considerable difference as to the amount of ABCast invocations. See the previous parameter.

Percentage of commit transactions If the commit percentage is **low**, many transactions end with an abort operation. In this case, the optimistic technique uses 0 ABCast invocations per transaction, whereas the pessimistic technique uses at least 1.

If the commit percentage is **high**, there is no difference.

Percentage of queries If the query percentage is **low**, there are many update transactions which means that both techniques use about 1 ABCast per transaction. However, there is a major difference between the techniques in respect to the amount of additional processing and the abort rate:

- In the pessimistic technique, there is no additional processing and the abort rate is 0.
- In the optimistic technique, there is additional processing in the form of the certification step. Furthermore, if the transactions have a tendency to access the same database items (i.e., transaction often interfere), the abort rate can be high because of failed certification tests and locking conflicts.

On the other hand, if the query percentage is **high**, the optimistic technique uses much less ABCast invocations than the pessimistic technique. Also, the former balances the query processing load

across the replicas, which can significantly improve performance in respect to the pessimistic replication technique.

Number of transactions submitted per time period Because of the load balancing possibilities of the optimistic technique, it can in some cases handle a higher number of transactions per time period than the pessimistic technique. For a discussion of these cases, see the previous and the next parameters.

Fraction of write operations in update transactions A **low** write fraction means a high fraction of read operations. In this case, the optimistic technique can load balance these reads and performs better than the pessimistic technique.

In the case of a **high** write percentage, there is no difference between the techniques.

There was one factor that was not mentioned when considering the transaction load parameters: the determinism requirements. This because we did not explain yet how exactly the determinism requirements are met for the different techniques (since this is a design level issue). But even if we had this information, it would not be easy to relate the performance implications of determinism to the characteristics of the submitted transactions. We therefore do not pursue this topic in detail, but make a general observation only:

It is likely that the determinism requirements have a more negative influence on the performance of the pessimistic technique than of the optimistic technique. This because in the pessimistic technique every operation needs to be deterministically processed, whereas in the optimistic technique, determinism is only needed for the processing of update messages.

Rules of thumb Concluding this discussion, we give the following rules of thumb to determine which replication technique is best used in a given situation:

- The pessimistic replication technique does not have the drawbacks of added processing overhead and possibly high abort rates. It follows that this technique should be used when the submitted transactions:
  - are one-shot and mostly update transactions with many writes, or
  - are interactive, very short (very few operations per transaction), and mostly update transactions, or
  - are mostly update transactions that often interfere, or
  - may not risk being aborted (for example, real-time systems could require this to ensure liveness[11])
- The optimistic replication technique should be used in all other cases, because it uses less ABCast messages and balances some of the processing load.

[11]The Database Replication Prototype does never ensure liveness because the Lock Manager may choose to delay a transaction indefinitely. The Lock Manager's policy could however be changed in order to ensure liveness in the case of pessimistic replication.

Chapter 4

Database Replication Prototype

The Database Replication Prototype is a replicated database system that is structured according to the Replication Framework. Two replication techniques were implemented within the prototype: pessimistic and optimistic active replication. For comparison reasons, a centralized database system was also created that uses the same components as the replication techniques. In Chapter 5, the prototype is used to compare the implemented replication techniques.

In this chapter we briefly discuss the main implementation choices and considerations. Also we explain how the prototype was tested. For details on the implementation we refer to the source code documentation. This documentation can be found on the web at: `http://lsewww.epfl.ch/~rvandewa`

4.1 Implementing the components of the Replication Framework

For the implementation of the components Database Service and Group Communication Service, existing software was used. Three different implementations of the Replication Manager were developed: one for each replication technique and one for the centralized database system. The components Client and Operations are specific to the Database Replication Prototype, so these were also developed as part of the project.

The implementation language used is Java, because it allows applications to be developed in a reasonably short time. The fact that Java is not a high performance language is not relevant to us: the prototype will only be used to compare the **relative** performance of replication techniques.

In the following subsections, we present the software used to implement the Database Service and the Group Communication Service. We also discuss the limitations of the software and how these were circumvented.

4.1.1 Database Service: POET

The Database Service is implemented by the POET Object Database [POE97]. This is a database with many features, but we only use it to store an array of objects that corresponds to the database items, and for its locking and transaction management. POET satisfies the ACID properties.

POET offers strict two phase locking, but it is not deterministic. The Lock Manager of the Database Replication Prototype solves this problem by constraining the order in which locks are requested from POET (when this is needed for the replication technique).

Recall that the optimistic replication technique needs to access the write operations and the read set of an update transaction when it passes to the **committing** state, so that it can ABCast an update message to all replicas. However, POET does not provide this access. Therefore, the Replication Manager itself

manages a copy of the readset and the writes for all **executing** transactions.

Another shortcoming of POET is that it does not allow read locks to be released before a transaction has been committed (even though this is allowed by the strict 2PL specification). The solution adopted is the following. When the read locks of a transaction need to be released, the transaction is aborted. Immediately after that, a new transaction is started. This new transaction acquires the write locks that where held by the old transaction and also executes the write operations of the old transaction. Note that the abortion of the old transaction and the acquiring of write locks for the new transaction must be done in one atomic step to prevent other transactions from "stealing" the locks.

4.1.2 Group Communication Service: OGS

The Group Communication Service is implemented by the Object Group Service (OGS) [Fel98]. OGS is a prototype that is built on top of CORBA. It is an infrastructure for distributed objects. These objects can communicate using Group Communication primitives and concepts. In the Database Replication Prototype, the replicas play the role of distributed objects and they communicate using the Atomic Broadcast implementation provided by OGS.

The Atomic Broadcast implementation of OGS satisfies the Atomic Broadcast specification (see section 2.2.2), but only under the assumption that a *majority* of the processes do not crash.[1] The majority of n processes is equal to $n/2+1$. When the assumption does not hold, the Group Communicate Service may cease to deliver messages. Because of this, the availability of the Database Replication Prototype is less than optimal: instead of tolerating the crashing of all but one replica, the Database Replication Prototype tolerates the crashing of only a minority of replicas.

A problem encountered when using OGS was that it violated the Order property of Atomic Broadcast in some cases: messages were sometimes delivered out of order at one replica. This problem was fixed by debugging and correcting the source code of OGS.

4.2 Testing the prototype

While successful tests are never a guarantee for correctness, they do increase the confidence one has in a system. Therefore, the Isolation and Atomicity properties were tested as described in this section. The Durability property was not tested.

The system is put under a high load transaction load. Every submitted transaction randomly decreases some database items by some random values and increased the same amount of items by the same values. Thus, every transaction modifies the database state, but the total value stored in the database always stays the same.

If after many committed transactions, the database value is still the same as in the beginning, the system is likely to guarantee Isolation and Atomicity. This because of the two properties were not guaranteed, the total value of the database would change.

The Database Replication Prototype was subjected to many test runs and never exhibited a change in the total database value. Thus, we are confident that the properties are upheld.

[1] More precisely, OGS' Atomic Broadcast is based on the Consensus algorithm by [CT96] which requires a majority of processes to operate.

Chapter 5

Performance Results and Evaluation

Using the Database Replication Prototype, the performance of replication techniques can be compared. In this chapter, we compare two techniques: pessimistic and optimistic active replication. This chapter first presents the parameters that are fixed for all experiments, those that vary across experiments, and the adopted performance indicators. Following that, it presents the measurement results of various scenarios and compares the two techniques quantitatively.

5.1 Prototype parameters and environment

In this section, we first define the terminology used for experiments. Then we discuss the prototype parameters that are fixed for all scenarios and those that are varied across scenarios. Finally, we describe the hardware/software environment in which the experiments are conducted.

5.1.1 Basic definitions

To talk about the configuration of experiments and the way they are conducted, we introduce the following terms:

scenario A scenario is a particular setting of all prototype parameters that may affect the performance of the replicated database system. Examples of prototype parameters include: the transaction characteristics defined in subsection 2.2.4, the replication technique used and the number of replicas in the system. The prototype parameters are described in the next subsection.

run A run is an execution of the prototype until some fixed number of random transactions has been committed. During the run, the prototype and the transactions are parameterized according to a given scenario. The performance results for a run are obtained by calculating performance indicators using the measurements taken during the run.

performance indicator A performance indicator is a value that characterizes the performance of the system during a run, for example, the mean response time for committed transactions. A performance indicator can be used as a *performance criterion* to compare runs that are parameterized according to different scenarios.

experiment An experiment is characterized by a number of (slightly different) scenarios. The goal of an experiment is to gain insight into the impact of a few prototype parameters on the system's performance. This is done by executing runs that are parameterized according to the scenarios and by comparing the performance indicators of these runs.

5.1.2 Prototype parameters

The prototype parameters configure the Database Replication Prototype. They can be divided in two classes: system parameters and application parameters. The *system* parameters are implementation choices that are not visible to the client. There are two system parameters: the replication technique and the number of replicas in the system.

On the other hand, the *application* parameters depend on the kind of application the client is modeling and are thus visible to the client. These parameters are independent of the implementation. The application parameters are: the transaction parameters defined in subsection 2.2.4, the database size, the database object size and the number of clients in the system.

The rest of this subsection explains which of the prototype parameters are *fixed*, and which are *variable*, respectively.

Fixed prototype parameters

Since many prototype parameters are orthogonal to the compared replication techniques, varying these does reveal characteristics of the prototype that are of interest to us. Moreover, since the possible parameter variations are very numerous, some main searching directions had to be chosen to be able to perform the experiments within the time that was available for the project. These two considerations resulted in the fixation of the following parameters:

number of operations per transaction: 10. This means that each transaction contains 8 data operations. The number of data operations is loosely modelled after [TPC94], where the retrieval of money from an ATM results in a transaction of 3 read operations, 3 write operations and an update of the ATM's retrieval history, which we model as the execution of two additional operations.

As stated in subsection 3.7.6, this parameter makes a considerable difference between the replication techniques in the case of interactive transactions. However, for more than a few operations, pessimistic active replication is not considered a viable approach. We decided that for this reason, we do not deeply investigate the characteristics of this case.[1] Since varying the parameter does not make a difference for one-shot transactions, we thus not vary it at all.

Note that this parameter might influence the amount of interference between transactions. Because of the limited time available, we have chosen not to investigate how this affects the replication techniques.

percentage of commit transactions: 100. This means that clients always request commit.

number of transactions submitted per time period: This parameter can, by design, not be set in the Database Replication Prototype. This because the number of clients is fixed for a given run and because the clients wait for the previous transaction to finish before submitting the next (see also the varied *number of clients* parameter).

However, this number is related to the measured *throughput* of the system: the number of transactions it commits per time period. Let C_r be the set of transactions committed in a run r, then we define:[2]

$$throughput(r) = \frac{\#C_r}{\text{duration of } r}$$

[1] Indeed, as is explained in subsection 5.4.3, doing multiple ABCast invocations per transaction costs so much time that pessimistic active replication with interactive transactions is not viable.

[2] $\#C$ is the cardinality of a set C, i.e., the number of elements C contains.

fraction of write operations in update transactions: 0.5. This is roughly in line with the ATM case described before.

As described in subsection 3.7.6, varying this fraction might reveal differences between the replication techniques as to their load balancing of read operations. However, in the Database Replication Prototype, the execution of operations takes very little time in comparison to ABCast invocations (see subsection 5.4.1). For this reason, we do not expect differences to be measurable, and thus, we do not vary this fraction.

database size: 1000 database objects.

Changing this parameter influences the amount of transaction interference. Because of the limited time available, we have chosen not to investigate how transaction interference affects the replication techniques. Therefore, this size is fixed.

database object size: 1 byte.

This number is chosen to minimize both the memory usage of Operations and the size of communication messages. Because of the prototype character of the project, the memory management of the Database Replication Prototype and of the Group Communication Service is not optimized. Choosing a small object size avoids problems due to excessive memory usage.

Variable prototype parameters

The following parameters are varied across the scenarios. We only give a global impression of the role these parameters play in the experiments. In the sections about the measurement results we exactly describe and motivate the parameters that are varied for a particular experiment.

replication technique: **pessimistic** active replication and **optimistic** active replication.

For analysis and comparison reasons, the performance of a **centralized** database system is also determined. This system is built using the same components and in the same environment as the Database Replication Prototype.

number of replicas: 3 to 5. The case of 2 replicas can not be examined because the Atomic Broadcast implementation of OGS requires at least 3 replicas to be able to deliver messages.

A large number of replicas increases the availability of the system, but might incur overhead because the ABCast primitive needs to synchronize messages across more replicas. On the other hand, this overhead may be compensated because of load balancing.

interactivity of transactions: **interactive** and **one-shot**.

percentage of queries: 0 to 99. We do not to set this percentage to 100 because a read-only system would be implemented in a very different manner than the Database Replication Prototype (for example, the ABCast primitive, lock management and certification are not needed in such a system).

From the system's point of view, this parameter influences the behaviour of the replication techniques in various ways (see subsection 3.7.6).

From the client's point of view, the percentage of queries depends on the kind of application being modelled: an application with few queries, such as an ATM application, or an application with many queries, such as web pages. Observe that most web pages are read by many visitors and changed by only a few editors.

Thus, since this parameter is significant internally as well as externally, it plays an important role in the experiments that compare replication techniques.

number of clients: 1 to 36. These clients are evenly distributed over the replicas.

The number of clients determines the degree of concurrency in the system. To see this, recall that each client submits its transactions sequentially. It follows that the maximum number of transactions concurrently processed by the system is equal to the number of clients. Note that the throughput depends on the number of clients because the degree of concurrency influences resource utilization, and thus, the throughput.

5.1.3 Scenarios and experiments revisited

Using the prototype parameters as defined in the previous subsections, this subsection introduces some notation for scenarios and experiments.

A *scenario* is denoted by a 5-tuple containing one valid value for every variable parameter. The values are ordered in the same order as their respective parameters in the previous subsection. Because the fixed parameters defined in subsection 5.1.2 have the same value in every scenario, they are not explicitly mentioned in the tuple. For example, let some scenario s be defined as follows:

$$s = (\textit{pessimistic}, 3, \textit{interactive}, 50\%, 15)$$

This means that s is a scenario in which the pessimistic replication technique is used in a system with 3 replicas. The transactions are interactive and 50% of them are queries. Finally, the total number of clients in the system is 15.

An *experiment* is denoted by a 5-tuple containing a set of valid values for every variable parameter. The sets are ordered in the same order as their respective variable parameters in subsection 5.1.2. For example, let some experiment e be defined as follows:

$$e = (\{\textit{pessimistic}, \textit{optimistic}\}, \{3, 4, 5\}, \textit{one-shot}, \{0\% \rightarrow 99\%; 10\%\}, 15)$$

We interpret e as the set containing all possible scenarios that can be constructed from the 5-tuple. A scenario can be constructed if its first value is included in the first set of e, its second value in the second set of e, etc. Two example scenarios that can be constructed from e: (*optimistic*, 4, *one-shot*, 20%, 15) and (*optimistic*, 3, *one-shot*, 99%, 15).

About the notation adopted for experiments: For singleton sets, the set brackets are optional. A notation in the form $\{a \rightarrow b; c\}$ denotes the set that contains a, b and all $d = a + nc | n \in \mathbf{N} \wedge d < b$. In the case of experiments with a centralized database system, the number of "replicas" is always 1.

5.1.4 Hardware/software environment

The experiments with the Database Replication Prototype were conducted using 5 Sun Solaris workstations connected by a Local Area Network. Each workstation hosts one replica as well as the clients that connect to this replica.

More precisely, the clients are modelled as separate threads that run in the same process as the replica they connect to. The client/server communication is just the invocation of the `process` method of the replica's Scheduler by the client. This means that the communication between the client and the replicated database system takes almost no time, which is just fine because this communication is orthogonal to the performance comparisons made in this chapter.

The main specifications of the workstations and the network are as follows:

workstations In scenarios with 1 to 4 replicas, 1 to 4 Ultra-60 workstations are used, each with a 360MHz CPU and 256MB of RAM.

In scenarios with 5 replicas, 1 Ultra-30 workstation is used in addition to the 4 Ultra-60's above. The Ultra-30 is equipped with a 248MHz CPU and 128MB of RAM.

network All workstations are connected via an Ethernet-LAN running at 100Mbps.

Since the workstations and the network were used for standard office applications during workdays, experiments were conducted only during night hours and weekends to avoid interference from these applications.

The Database Replication Prototype is compiled and executed using the Java Development Kit 1.1.7B provided by Sun.

5.2 Performance indicators

For the performance comparison of the active replication techniques we use the following main performance indicators: *mean response time per committed transaction* and *throughput*. To gain insight into the tradeoff between CPU utilization and network utilization, the *network and processing delays per committed transaction* are considered. In the case of the optimistic replication technique the *abort rate* is used to assess the impact of abortions due to the fact that this technique is certification based (see also subsection 3.7.5).

This section defines the mentioned indicators, but first it gives some basic definitions.

5.2.1 Basic definitions

From the client's point of view, the time it takes for some Operation o to be processed by a database system is the time between the submission of o and the reception of o's result. The fact that in many cases o is processed at all replicas is of no importance to the client. In the following definitions we therefore only consider the time the database system spent on processing o to obtain the result that the client received.[3]

proc(o) The amount of time that o spent in the components Database Service and Replication Manager from the moment o is submitted by the client, until the moment the client received o's result.

Some factors influencing this time are bookkeeping activities in the Scheduler, the waiting for locks in the Lock Manager and the execution of Operations by the Database Service.

net(o) The amount of time that o spent in the component Group Communication Service from the moment o is submitted by the client, until the moment the client received o's result.

If o is ABCast, this time is the sum of: the time between the ABCast of o at some originating replica p and the delivery of o at some replica q, and the time it takes for the result of o to be sent back from q to p.

If o is not ABCast, this time is zero.

total(o) The amount of time between the moment o is submitted by the client, until the moment the client received o's result.

This value is equal to: $proc(o) + net(o)$.

[3] Recall that in the case of the processing of o on all replicas, the result returned to the client is the first result that arrives at the originating replica.

5.2.2 Mean response time for committed transactions

The response time for a committed transaction is loosely defined as follows: the time between the submission of the first Operation of the transaction by the client and the moment at which the client receives the response that the transaction was committed.

This definition has two implications:

- transactions that end with an abort operation are not considered, and
- transactions that are forcefully aborted are only considered when the client resubmits these (possibly multiple times) until they are committed. The response time is then defined as follows: the time from the submission of the first Operation of the first forcefully aborted transaction until the reception of the "commit succeeded" response.

Let O be the collection of all Operations that are submitted on behalf of T. The response time for a committed transaction T is then formally defined as follows:

$$total(T) = \sum_{o \in O} total(o)$$

In case that T commits the first time it is submitted, O contains exactly those Operations that form T. When T is forcefully aborted one or more times, some Operations of T appear multiple times in O.

Let C_R be a set containing all transactions committed in a given set of runs R. The performance indicator *mean response time for committed transactions* is defined as follows:

$$mean(R) = \frac{\sum_{T \in C_R} total(T)}{\#C_R}$$

To illustrate this definition, Figure 5.1 shows a typical frequency diagram for 100 runs of 2000 submitted transactions each. This is an example only, the full results are presented and analyzed in section 5.4. All runs the figure were executed according to the following scenario: (*pessimistic*, 3, *one-shot*, 50%, 15). The diagram shows the relative frequency distribution of the response times of committed transactions: the higher the line for a given response time, the more often that response time was observed during the runs. The *mean* is plotted as a vertical dotted bar, its value is 1221 *ms*.

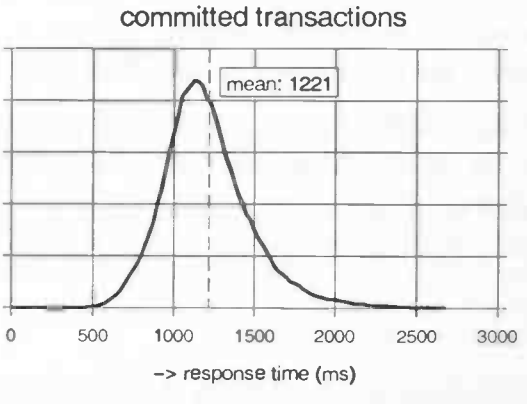

Figure 5.1: Typical distribution of response times for pessimistic replication

5.2.3 Throughput

The mean response time for committed transactions can be directly converted to the *throughput* of the system: the number of transactions it commits per time unit. Let $\#clients(r)$ denote the number of clients in a run r, then:

$$throughput(r) = \#\text{clients}(r)/mean(r)$$

So the throughput takes into account the number of clients. When comparing scenarios that have the same number of clients, we use the mean response time. When the number of clients varies, we use the throughput. Note that the throughput performance indicator plays an important role in industry benchmarks such as [TPC94].

5.2.4 Network and processing delays for committed transactions

Let O be the collection of all Operations that are submitted on behalf of T. Similarly to the response time, the network delay and the processing delay for a committed transaction are defined:

$$\begin{aligned} net(T) &= \sum_{o \in O} net(o) \\ proc(T) &= \sum_{o \in O} net(o) \end{aligned}$$

5.2.5 Abort rate

The *abort rate* in a run r is defined as follows:

$$abortRate(r) = \frac{\text{number of transactions forcefully aborted}}{\text{total number of transactions committed or aborted}}$$

This performance indicator only applies to scenarios featuring the optimistic replication technique, since in the pessimistic replication technique transactions are never forcefully aborted.

5.2.6 Separating queries and update transactions

As described in section 3.3, the optimistic replication technique processes queries on one replica only, whereas update transactions are processed on all replicas. Because of this fundamental difference, we consider queries and update transactions separately in some comparisons.

Let R be a set of runs. $mean(R|q)$ denotes the mean response time for committed queries in R. $mean(R|u)$ for the mean response time for committed update transactions in R.

To illustrate these definitions, Figure 5.2 shows a frequency distribution similar to Figure 5.1. However, the scenario is different: (*optimistic*, 3, *one-shot*, 50%, 15). The leftmost diagram shows all committed transactions, $mean(R) = 858$. In the middle diagram the committed queries are plotted, $mean(R|q) = 187$. The rightmost diagram shows the committed update transactions, $mean(R|u) = 1540$.

Since the scenario specifies 50% queries, one might expect that:

$$mean(R) = (mean(R|q) + mean(R|u))/2 = 864$$

However, this is somewhat different from the actual value of $mean(R)$ (858). This difference is due to two factors. Firstly, the implementation only approximates the 50% value because it uses a random

generator to decide on the interactivity of a submitted transaction. Secondly, some of the submitted transactions have been forcefully aborted, influencing the number of queries and update transactions actually committed.

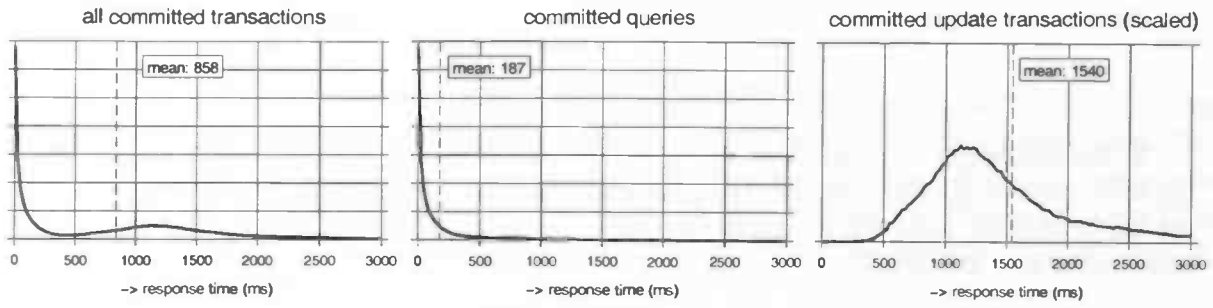

Figure 5.2: Typical distribution of response times for optimistic replication

5.2.7 Motivation for the performance indicators

We have chosen to use the mean transaction response time and the throughput as the main performance indicators. Advantages of these indicators are that they are very commonly used and that they take into account all measured values.

A disadvantage is that they do not include information on the distribution of the measured values. A related problem is that for interactive applications (i.e., those involving impatient end users), the maximum expected response time may be more interesting than the mean response time. We address these issues when discussing the typical frequency distributions for the response times in both replication techniques (see section 5.4.1).

5.3 Data gathering

This section explains how reliable experimental data was gathered using the Database Replication Prototype. It discusses the stability of the prototype during long runs and analyzes the behaviour of the throughput of the system when the concurrency level is varied. (The comparison of replication techniques is treated in section 5.4.)

5.3.1 Reliable performance indicators

As defined in subsection 5.1.1, an experiment is a set of scenarios. The performance indicators for a scenario are obtained by conducting runs according to this scenario.

A way to obtain statistically reliable indicators for a scenario is to conduct one run parameterized according to the scenario. In this run, transactions are submitted and response times are measured until the performance indicators reach a certain degree of statistical stability. This approach has two disadvantages that make it infeasible for experiments conducted using the Database Replication Prototype:

- As the experiments are conducted in an office environment, the performance indicators obtained during a single run can be subject to interference. Such interference can be caused by, for example, users logging in at unexpected times, the operating system behaving unexpectedly, network congestion due to web-site traffic, etc.

- The approach assumes that the system reaches a *steady state* [TPC94] after some number of transactions is processed and remains in that state ever after. The system is in a steady state when submitting any number of additional transactions does not noticeably influence the performance indicators.

  The steady state assumption does not hold for the Database Replication Prototype. More precisely, the performance of the Atomic Broadcast primitive offered by OGS declines over time: the later a message appears in the total order, the slower it is delivered at all replicas (i.e., $net(T)$ gets larger over time). Another problem of OGS is that it crashes after delivering about 3000 Atomic Broadcast messages.[4] Note that these behaviours are due to the prototyping nature of OGS: more mature implementations usually do not crash after sending just a few thousand messages and the delivery time does not vary as a function of the system's lifetime.

  For a quantitative illustration, see Figure 5.3. It shows the evolution of the network and processing delays of transactions during the lifetime of the system. The scenario is: (*pessimistic*, 3, *one-shot*, 50%, 15). $net(T)$ is short for the first 30 transactions, but after that it more or less stabilizes at 1200 *ms*. However, after the 300th transaction the times start to increase steadily, reaching more than 1300 *ms* per transaction. The run was stopped after about 670 ABCast invocations per replica (the system-wide total was fixed at 2000). Aside from a slow start, $proc(T)$ is constant over time.

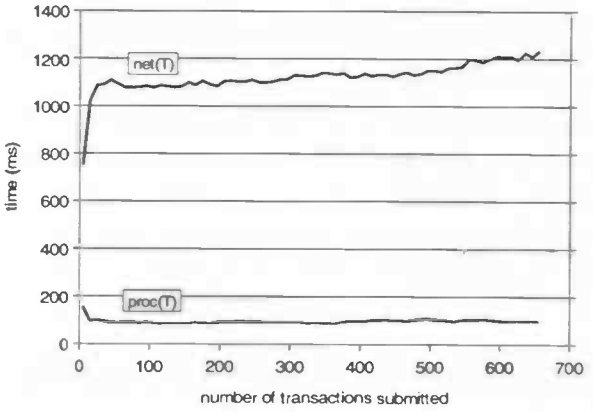

Figure 5.3: Evolution of network and processing delays

  As is clear from the figure, $net(T)$ dominates over $proc(T)$. Recall that the performance indicator mean response time for committed transactions is an average over the sum of $net(T)$ and $proc(T)$. The instability of $net(T)$ thus disturbs the comparison of replication techniques, because these utilize varying amounts of ABCast messages during the lifetime of the system.

Since both disadvantages are particularities of our implementation, they need to be overcome to obtain performance comparisons that are generally applicable. The solution adopted is to measure the performance indicators of a scenario s by performing **multiple short runs** parameterized according to s. Now, random interference is ruled out by spreading the runs over time and by interleaving the runs of different scenarios.

Furthermore, the performance degradation of Atomic Broadcast is overcome by stopping each run when either 2000 messages have been delivered or 8000 transactions have been committed. The latter bound is imposed to prevent runs from taking too long in scenarios with very few ABCast messages.

[4] These problems are likely due to the fact that OGS does not garbage collect certain buffer structures, leading to unbounded memory usage, decreasing performance and crashes. The problems were discovered when the project was already in an advanced stage, so it was not doable to replace OGS by another implementation.

By defining runs this way, we ensure a fair comparison between the techniques: when 25% or more of the committed transactions in a given run require an ABCast message to be sent, all techniques observe exactly the same characteristics of the Atomic Broadcast implementation. If less than 25% require ABCast messages, the impact of the ABCast performance degradation becomes sufficiently small for it not to have a disturbing influence.

Stability of the Scheduler, Lock Manager and Database Service In the previous discussion, we did not consider whether components other than the Group Communication Service reach a steady state. This is important because the number of transactions committed may vary between 2000 (if all transactions require an ABCast message) and 8000 (if less than 25% require ABCast). In Figure 5.4, $proc(T)$ is shown just as in Figure 5.3, but now for a centralized database system. The scenario is: (*centralized*, 1, *one-shot*, 50%, $\{5, 10, 15\}$).

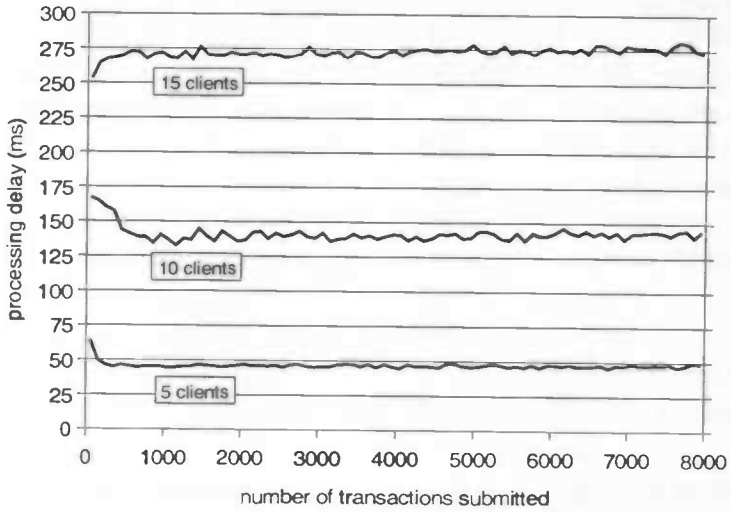

Figure 5.4: Evolution of the processing delay in a centralized database system

The number of clients is varied to approximate the concurrency levels of the two active replication techniques. In the case of 3 replicas and 15 clients, the pessimistic technique has a concurrency level of 15 on every replica, while for the optimistic technique, it varies between 5 and 15 according to the load balancing. The figure shows that for all levels of concurrency, $proc(T)$ is more or less constant over time. The conclusion is that the Scheduler, Lock Manager and Database Service indeed reach a steady state: varying the number of transactions does not disturb the performance indicators.

Number of runs per scenario To decide how many runs need to be conducted to obtain statistically relevant performance indicators for a scenario, we conducted 100 runs per scenario for the following typical experiment. In Figure 5.5 $mean(s)$ is plotted for ($\{$*pessimistic, optimistic*$\}$, 3, *one-shot*, 50%, 15). Every point plotted corresponds to the mean response time for committed transactions, averaged over batches of 5 runs.

In the optimistic scenario, $mean(100\text{ runs}) = 858$. The minimum and maximum observed for batches of 5 runs are 843 and 868, and the standard deviation for all 20 batches is 6.5. Assuming that the mean computed over 100 runs is a good approximation for $mean(s)$, it is highly probable that the mean computed using any 5 runs is within 2% of $mean(s)$. For the pessimistic scenario, the numbers are similar.

The performance indicators in the rest of this chapter are computed over at least 5 runs. Since the scenarios considered are similar to the scenarios in Figure 5.5, the error for the mean in these scenarios

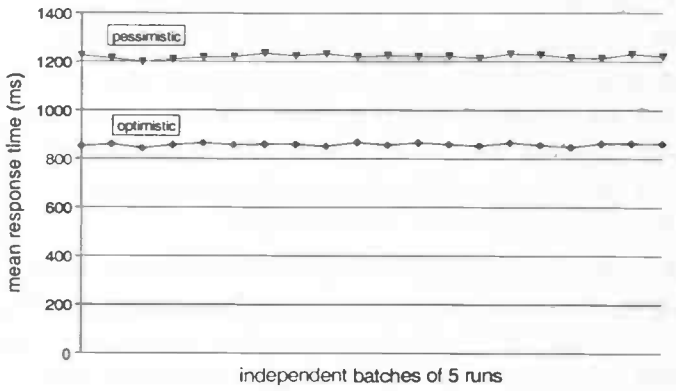

Figure 5.5: Mean response time observed during 100 runs, averaged per 5 runs

is very likely also smaller than 2%. Following a similar procedure, the error for the *abortRate*(s) is determined to be smaller than 5%.

5.3.2 Optimizing throughput

In real world situations, the number of clients is usually determined by the application. Therefore, we do not vary this number when comparing replication techniques in the next section. Still, we must choose an appropriate value for this parameter. In line with [TPC94], we use a value that (more or less) optimizes the throughput. The current subsection explains how this value was determined.

As described in section 5.1.2, the number of clients determines the concurrency level of the Database Replication Prototype. In order to find the concurrency level that results in a high throughput, we conducted the following experiment: (*centralized*, 1, *one-shot*, $\{0\%, 50\%, 99\%\}$, $\{1 \rightarrow 10; 1\} \cup \{12, 15, 21, 36\}$) (for the moment, we only consider the centralized case).

Figure 5.6 shows the throughput as a function of the number of clients. For 3 to 5 clients, throughput is more or less optimal. Then it starts to degrade to about 50% of the maximum, at which it stabilizes (from 15 clients and up).

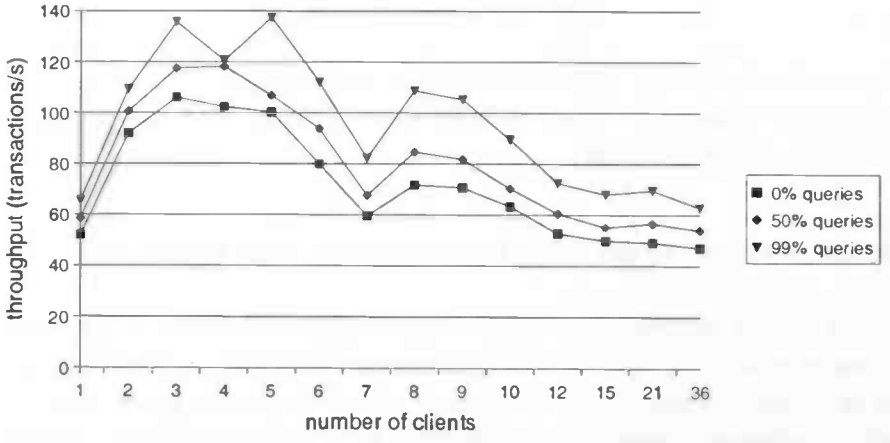

Figure 5.6: Throughput in a centralized database system

The general trend shown is explained as follows: 1 or 2 clients poorly utilize the available resources because the system wastes a considerable amount of CPU time waiting for I/O operations to complete. In the case of 3 to 5 clients, the system can use all resources with limited transaction interference: a

submitted transaction can obtain all its locks without being delayed. When more than 5 clients are used, transactions start interfering, so the probability that a submitted transaction is delayed increases. From an implementation point of view, delaying a transaction means that its Java thread is blocked and later restarted when the requested lock is available. The fact that Java thread operations are very time-expensive explains the steep drop for 6 to about 15 clients, because transaction interference rises significantly during this interval. For more than 15 clients, the transaction interference increases less quickly, leading to a more gentle decline of the throughput.

The similar shape of the three lines indicates that the behaviour of the throughput is independent of the query percentages. However, the throughput for high query percentages is better than for low query percentages. This is explained by the fact that update transactions potentially interfere with both queries and other update transactions, leading to locking conflicts and delays. On the other hand, queries only delay when they interfere with update transactions. So, the higher the query percentage, the less transactions are delayed and the higher the throughput.

We have no explanation for the throughput drops in the scenarios with 7 clients and in the scenario with 4 clients and 99% queries. Maybe they are due to peculiarities in POET's transaction scheduling or Java's thread scheduling mechanisms.

Throughput in replicated scenarios To have a good throughput value in a replicated system, the number of transactions concurrently processed on each replica should be optimal. However, the replication technique chosen influences the load balancing across replicas and thus the number of transactions concurrently processed on each replica.

We decided to set the total number of clients to 15 in scenarios with 3 or 5 replicas and to 16 in scenarios with 4 replicas. This way, the number of transactions concurrently processed varies between, approximately, 3 and 16 (dependent on the replication technique, the query percentage and the number of replicas).

A possible drawback of this choice is that for the pessimistic replication technique, the number of concurrent transactions at each replica is 15 or 16 because all processing is replicated. We do not consider this a serious problem for the following reasons:

- The network time plays a dominant role in this technique (see section 5.3.1), so the performance loss due to the suboptimal concurrency level does not have a large influence on the overall performance.
- The alternative of allowing only one client per replica results in the desired 3 to 5 concurrent transactions on all replicas. However, such a low number of clients per replica would be too distant from real world situations.

5.4 Quantitative comparison of pessimistic and optimistic replication

In this section we present and discuss the performance results obtained for the pessimistic and the optimistic replication techniques. First, we show the frequency distributions of two typical scenarios to get a general idea of the characteristics of each technique. Then, we present the behaviour of the techniques in case the query percentage is varied, because this parameter is important to the application as well as to the replication technique. After this, we discuss the influence of the interactivity of transactions on both replication techniques and explain why we do not consider interactive transactions in other experiments.

Since scalability is an important topic, we extensively discuss the performance of the techniques when the number of replicas is varied. We conclude by giving a summary of the results. We also state the limitations of our approach and discuss some possible improvements of the optimistic technique.

5.4.1 Frequency distribution of response times

We discuss the general characteristics of the two techniques by showing the frequency distribution of the response times for two typical scenarios. The experiment conducted is: ({*pessimistic, optimistic*}, 3, *one-shot*, 50%, 15). The submitted transactions contain random operations that uniformly access all database objects. Hereafter, the scenario that features the pessimistic (optimistic) technique is called s_p (s_o).

Figure 5.7 shows the frequency distribution for query response times and update transaction response times in s_p. For the frequency distributions in s_o, see Figure 5.8. Next, we discuss and compare several aspects of the distributions.

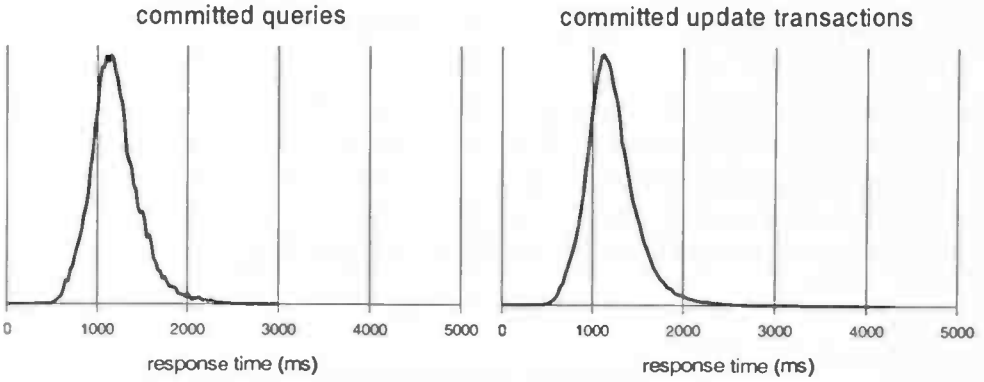

Figure 5.7: Frequency distribution of response times for pessimistic replication

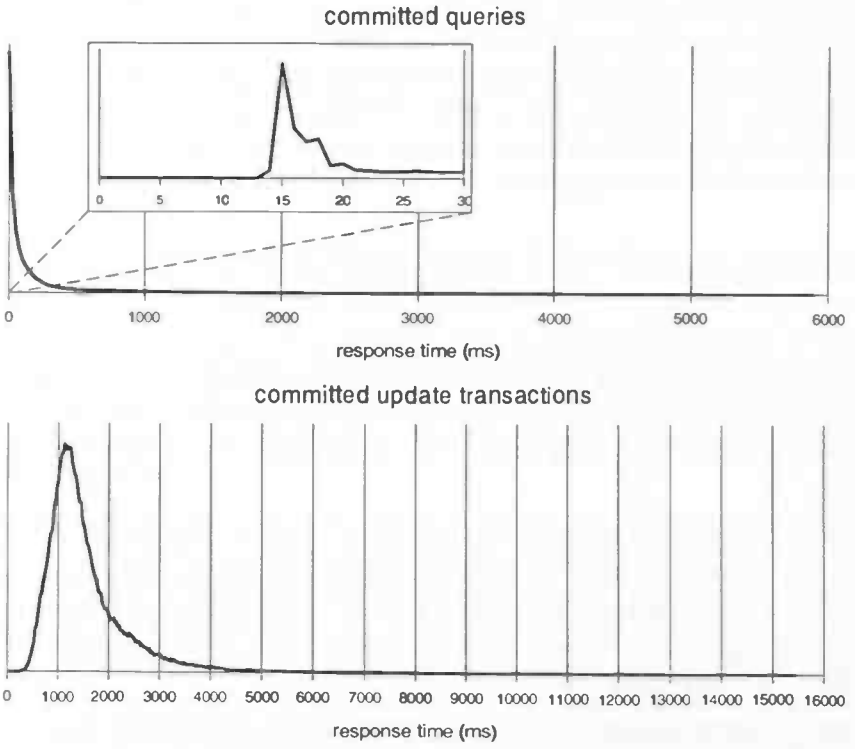

Figure 5.8: Frequency distribution of response times for optimistic replication

General shape The frequency distributions, except for the one showing the queries in s_o, have the same shape. This is explained by the fact that in the corresponding scenarios, one Atomic Broadcast invocation is performed for each transaction. Recall that the time required for sending an ABCast message is dominant over all other transaction processing (see section 5.3.1). Thus, the shape is formed by the Atomic Broadcast primitive.

Atomic Broadcast Figure 5.9 separately shows the distribution of the network delays (i.e., ABCast delays). Clearly, it has the same Gaussian shape as seen in Figures 5.7 and 5.8.

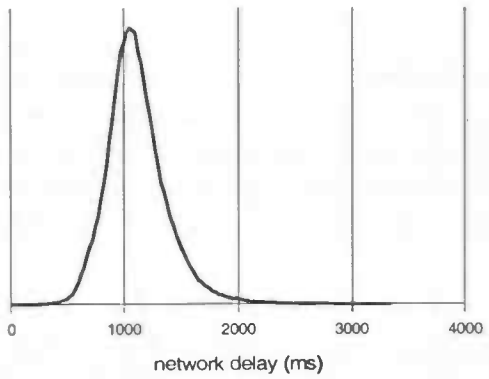

Figure 5.9: Frequency distribution of network delays

The shape resembles the Gaussian distribution for the following reason. The Atomic Broadcast primitive is built on top of a stack of network protocols that each manage message queues. Each message passes through these queues and risks being delayed in every queue. The most likely case is that a message is delayed by some of these queues and not delayed by others. It is less likely that a message is queued almost nowhere or almost everywhere. When many messages are sent, this results in a Gaussian distribution. Because a message requires a minimum time to pass through the protocol stack and be transmitted over the network, even when it is never queued, the response time does not get below a certain minimum value. The minimum value observed was 131 *ms*.

Queries in optimistic replication In s_o, the Atomic Broadcast primitive is not used for queries. The result is that most queries take no more than 15 to 20 *ms*, as can be seen in the close-up. Some queries take longer: they are delayed because of a locking conflict with some other transaction. Queries that take very long are probably delayed by a write lock held by a **committing** update transaction: such a lock is not released until the update transaction has been delivered and certified, which may take a few seconds. Also, some queries have been forcefully aborted and resubmitted, extending their response time.

Extreme values For update transactions, the maximum response time observed is 4313 *ms* in s_p and 15458 *ms* in s_o. For queries, the maxima are 3021 *ms* and 5907 *ms*, respectively. In s_p, the maxima are lower than in s_o because transactions are never forcefully aborted in s_p whereas in s_o, the abort rate is 6.9%. For the same reason, the distributions are skewed more in s_o than in s_p (i.e., in s_o the "tails of the distributions" are longer and contain more transactions).

The difference between the techniques is larger for update transactions than for queries. This is explained by the fact that if an update transaction is forcefully aborted, multiple expensive ABCast invocations are needed which severely slow down the transaction. For queries, resubmissions are less expensive.

Queries vs. update transactions Clearly, queries and update transactions behave very differently in s_o. In s_p however, the distributions are nearly the same for both types. The conclusion is that in scenarios featuring the pessimistic technique, a transaction's response time does almost not depend on its type.[5]

5.4.2 Percentage of queries

As noted in section 5.1.2, the percentage of queries parameter is important when comparing replication techniques. The parameter characterizes the application (i.e., many queries or many update transactions) and has a significant effect on the techniques (i.e., a varying number of ABCast messages per transaction). For these reasons we vary the query percentage in all following experiments. We do not show the frequency distributions but limit ourselves to the two performance indicators mean response time and abort rate. In order to shorten the text somewhat, we use the terms PR and OR for pessimistic replication and optimistic replication, respectively.

This subsection discusses how the query percentage affects the two techniques in the case of 3 replicas. Figure 5.10 shows $mean(s)$ for the scenarios in the experiment ($\{$*pessimistic, optimistic*$\}$, 3, *one-shot*, $\{0\%, 25\%\} \cup \{50\% \rightarrow 99\%; 10\%\}$, 15). Queries and update transactions are displayed in separate diagrams because there is a considerable difference between the way the techniques handle these. The lines for queries start at 25% because queries are not submitted in scenarios with a query percentage of 0%.

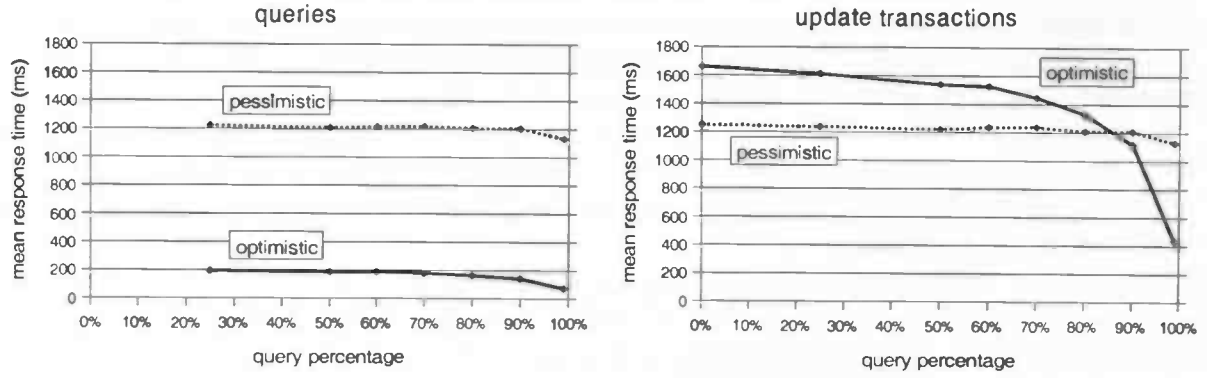

Figure 5.10: Mean response time per technique for varying query percentages

Queries The diagram on the left shows that queries are handled about 6 times faster in OR than in PR, independent of the query percentage. Recall that in OR queries are only processed locally, while in PR queries are Atomic Broadcast to all replicas. The slowness of the ABCast primitive of OGS is the main cause for the observed difference.

In both techniques, the response time goes down as the query percentage increases. The reason for this is that the transaction interference decreases (since queries do not interfere with queries). However, in relative terms the decrease is much larger for OR. To see this in the extreme case, compare the response time for 99% queries to that of 25% queries: OR gains about 50% while PR gains about 5%. The reason for this is the load balancing capability of OR: the processing load on every replica decreases as the query percentage increases, leading to lower response times. Though some queries were forcefully aborted in OR (see next paragraph) this does not have a significant impact because resubmitting and reprocessing a query is not very expensive.

[5] This is orthogonal to the query percentage parameter because this parameter affects the amount of transaction interference and thus the response time of all transactions.

Update transactions The diagram on the right shows that in most cases, PR handles update transactions 20% to 30% faster than OR. The difference is relatively small because in both techniques a slow ABCast is used per update transaction. It is due to the fact that for low query percentages the abort rate (in OR) is noticable: from 13% for 0% queries to 5% for 60% queries (see the curve for 3 replicas in Figure 5.15). Aborting update transactions is expensive because it is resubmitted and ABCast again, leading to two or more ABCast invocations (or more) for the same transaction.

For query percentages beyond about 85%, OR does better than PR. In these cases, relatively few ABCast messages are sent in OR. The load on OGS is thus very low, resulting in a much better performance of the Atomic Broadcast primitive and lower overall response times.

5.4.3 Interactivity of transactions

In most experiments we consider one-shot transactions only. This section shows how the replication techniques behave in the case of interactive transactions. Also, it explains that interactive transactions are (1) not feasible in pessimistic replication and (2) exhibit the same behaviour as one-shot transactions in optimistic replication. Though it is not shown in this section, the conclusions also apply when the number of replicas is varied.

Optimistic replication Figure 5.11 shows $mean(s)$ for the experiment (*optimistic*, 3, {*one-shot, interactive*}, $\{0\% \rightarrow 99\%; 25\%\}$, 15).[6] The mean response time for interactive transactions is always approximately 200 *ms* larger than the mean for one-shot transactions. The reason is that for interactive transactions, the client repeatedly invokes the originating replica for every operation, while for update transactions, the client invokes the replica only once. Since (1) this overhead is some constant time for all transactions and (2) all other processing is the same in both scenarios, the difference between the mean response times in both scenarios is constant. The figure shows that excluding this difference, the techniques behave the same in both situations (the response times decline in the same way when the number of queries is varied).

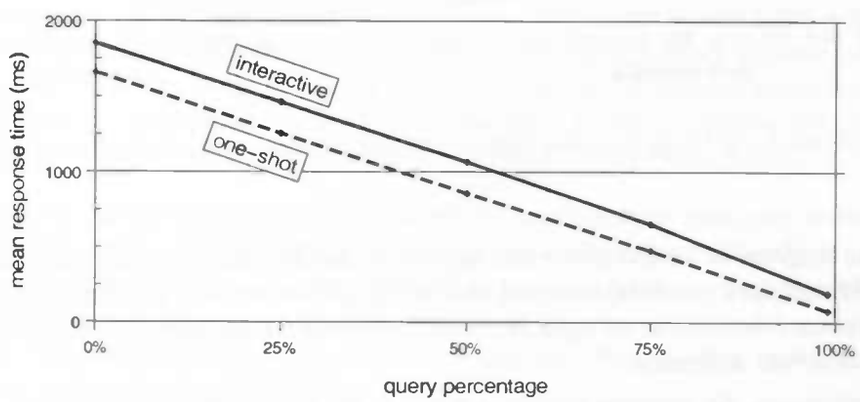

Figure 5.11: Interactive vs. one-shot transactions in optimistic replication

Pessimistic replication Figure 5.12 shows $mean(s)$ for the experiment (*pessimistic*, 3, {*one-shot, interactive*}, $\{0\% \rightarrow 99\%; 25\%\}$, 15). The mean response time for interactive transactions is almost 8 times higher than the mean for one-shot transactions. This is explained by the fact that 10 (slow) ABCast messages are sent per interactive transaction instead of 1 per one-shot transaction.

[6] Because it does not influence the conclusions, we do not separate queries and update transactions in this section. Also note that the linear character of the curves is accidental.

When we consider interactive transactions and compare the two techniques, the mean response time in PR is at least 4 times the mean in OR.

For one-shot transactions, both techniques have comparable response times: for few queries, PR performs better, whereas for many queries, OR wins.

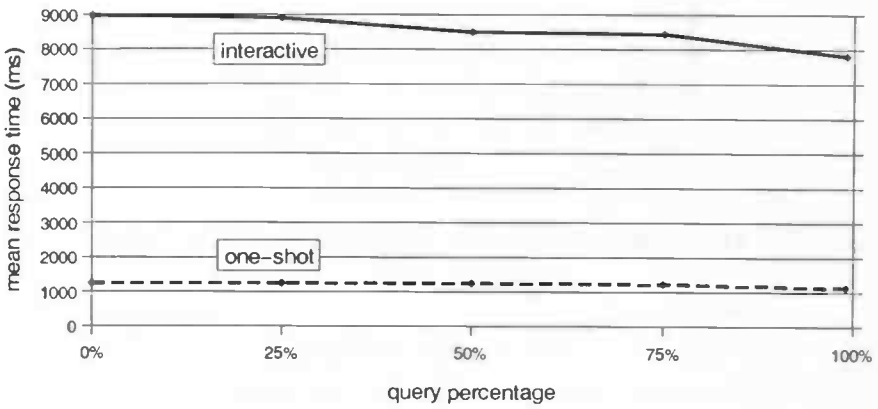

Figure 5.12: Interactive vs. one-shot transactions in pessimistic replication

Conclusion For interactive transactions, optimistic replication outperforms pessimistic replication by far. This large difference makes the pessimistic technique infeasible for all scenarios that feature interactive transactions. For one-shot transactions, the response times of both techniques are similar. Therefore we focus on this transaction type in other experiments.

5.4.4 Scalability

This section examines the scalability properties of the two replication techniques, i.e., the behaviour of the techniques when the number of replicas is varied. In both replication techniques, adding replicas increases the availability of the system. However, because of increased communication overhead, adding replicas may hurt performance. In this section we quantify this tradeoff. We show that in most scenarios, performance decreases, but that in some scenarios, the optimistic technique scales nicely: adding replicas then results in sustained or increased performance.

The experiments presented in this subsection are similar to the experiment discussed in subsection 5.4.3. The difference is that here, the number of replicas is varied as well. For a discussion of the behaviours that are independent of the number of replicas we refer to subsection 5.4.3. The number of clients is 15 or 16 (depending on the number of replicas in a scenario) because the implementation requires that there is an equal amount of clients on every replica. To compensate for this variation we use the throughput as the performance indicator.

Pessimistic replication Figure 5.13 shows the results for the experiment (*pessimistic*, $\{3, 4, 5\}$, *one-shot*, $\{0\%, 25\%\} \cup \{50\% \rightarrow 99\%; 10\%\}$, 15 or 16). Queries and update transactions are mixed because their response times (and their throughputs) are the same (subsection 5.4.1). The interpretation of the figure is straightforward: pessimistic replication does not scale well because the throughput decreases significantly as replicas are added. This is explained by the fact that the Atomic Broadcast primitive offered by OGS gets slower when a message must be sent to a larger group of processes.

Optimistic replication Figure 5.14 shows the results for the experiment (*optimistic*, $\{3, 4, 5\}$, *one-shot*, $\{0\%, 25\%\} \cup \{50\% \rightarrow 99\%; 10\%\}$, 15 or 16). Next, we treat the two diagrams separately.

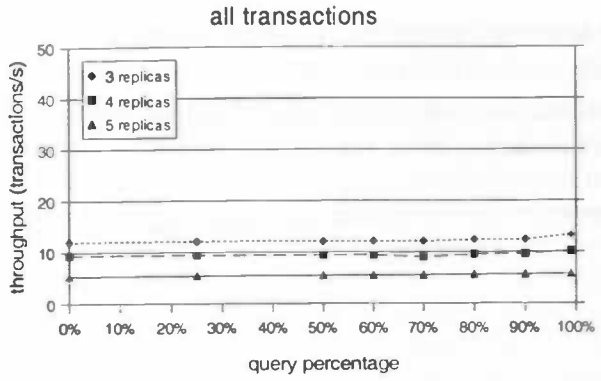

Figure 5.13: Throughput of pessimistic replication for varying numbers of replicas

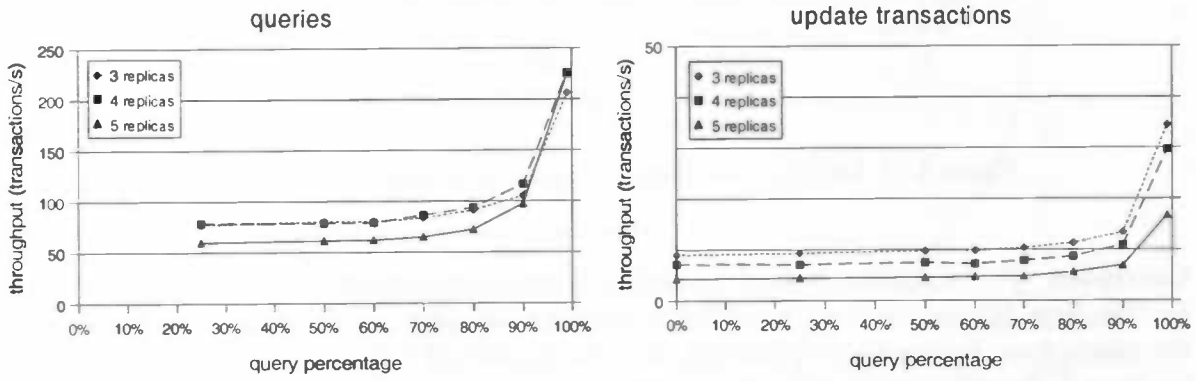

Figure 5.14: Throughput of optimistic replication for varying numbers of replicas

queries For five replicas, the throughput is lower than for 3 or 4 replicas since the fifth replica runs on a significantly slower machine (see subsection 5.1.4). The scenarios with 3 and 4 replicas exhibit approximately the same performance. The reason is that for high query percentages, about 5 and 4 transactions are concurrently processed in the cases of 3 and 4 replicas, respectively. As can be seen in Figure 5.6, throughput is approximately the same for these amounts of clients. Thus, for queries the number of replicas can be scaled to 4 without hurting performance.

update transactions As the number of replicas increases, the throughput gets lower without exception. This is mainly because update transactions depend on the slow Atomic Broadcast primitive. Like for the pessimistic technique, the time an ABCast message takes goes up when the number of replicas increases. A secondary negative influence is the abort rate which grows when replicas are added (the abort rate is treated in the next paragraph). Thus, as far as update transactions are concerned, scaling costs performance.

Abort rate Figure 5.15 shows the abort rate for the optimistic replication experiment described in the previous paragraph. We distinguish two trends: (1) the abort rate goes down when the query percentage increases and (2) the abort rate goes up when the number of replicas increases. Next we explain both trends.

query percentage Forceful aborts are only caused by update transactions. According to the result of the certification test, an update transaction is either committed or forcefully aborted because of a conflict with another update transaction. When it is committed, it potentially causes the forceful

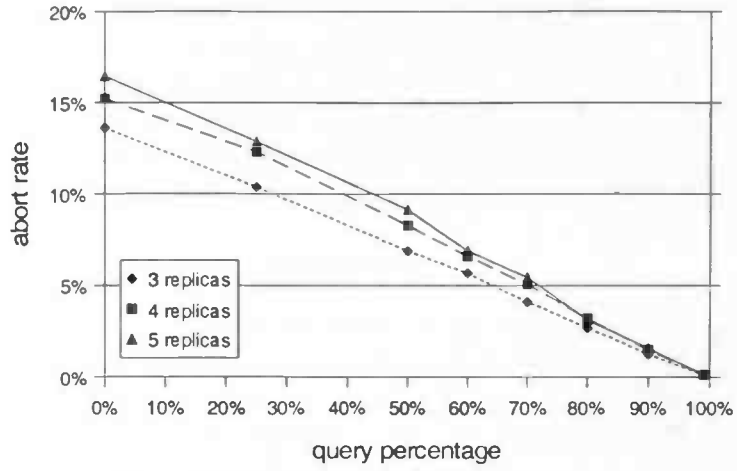

Figure 5.15: Abort rate

abort of locally **executing** transactions. When the query percentage increases, less and less update transactions are processed and certified during a run, leading to a lower abort rate.

number of replicas As the number of replicas grows, processing becomes more and more decentralized. We explain why this decentralization leads to a higher abort rate using the following example. First recall that conflicts between transactions processed at different replicas are detected afterwards, at the certification test. Then imagine that 5 update transactions are processed on, and fairly distributed over, a set of either 3 or 5 replicas:

- In the case of 3 replicas, two replicas each process two transactions and one replica processes one. When each transaction is certified, it is checked if it conflicts with some other committed transaction. Now imagine that one of the two transactions originating from the same replica is certified and committed. Sometime later, the second transaction originating from the same replica is certified. When it is compared to the first transaction, it never conflicts because because both transactions originate from the same replica. On this replica, the Lock Manager prevented the transactions from conflicting. The same holds for the other pair of transactions processed at the same replica. Of course these transactions may still conflict with the 3 transactions originating from other replicas.
- In the case of 5 replicas, every transaction runs on a different replica. Because there is no synchronization between the transactions before they are certified, they could conflict with any 4 of the other transactions.

The conclusion is that for 3 replicas, conflicts are somewhat less likely to occur than for 5 replicas, as is illustrated by Figure 5.15. As stated in section 5.3.1, the differences for individual points might be off by up to 5%. For example, in scenario (*optimistic*, 4, *one-shot*, 0%, 16) the abort rate lies within 14.25% and 15.75%. Since the measured points show a consistent behaviour, i.e., the lines between them do not cross, we think the figure does illustrate the effect described.

5.4.5 The big picture

In the previous subsections, we looked at the performance tradeoff between the replication techniques from different angles and explained the behaviours encountered. In this section we combine the results and present a brief overview of the tradeoff between the centralized approach, the pessimistic technique and the optimistic technique.

Figure 5.16 compares the throughput in a centralized database system, in pessimistic replication and in optimistic replication. The two replicated systems contain 4 replicas (machines). The centralized database system runs on one machine. To the system, 16 clients are connected. This means 4 clients per replica in the replicated case and 16 clients on one machine in the centralized case. The query percentage is varied from 0% to 99% because the main benefit of optimistic replication is that queries are processed on one replica only. All transactions are counted, i.e., queries as well as update transactions.

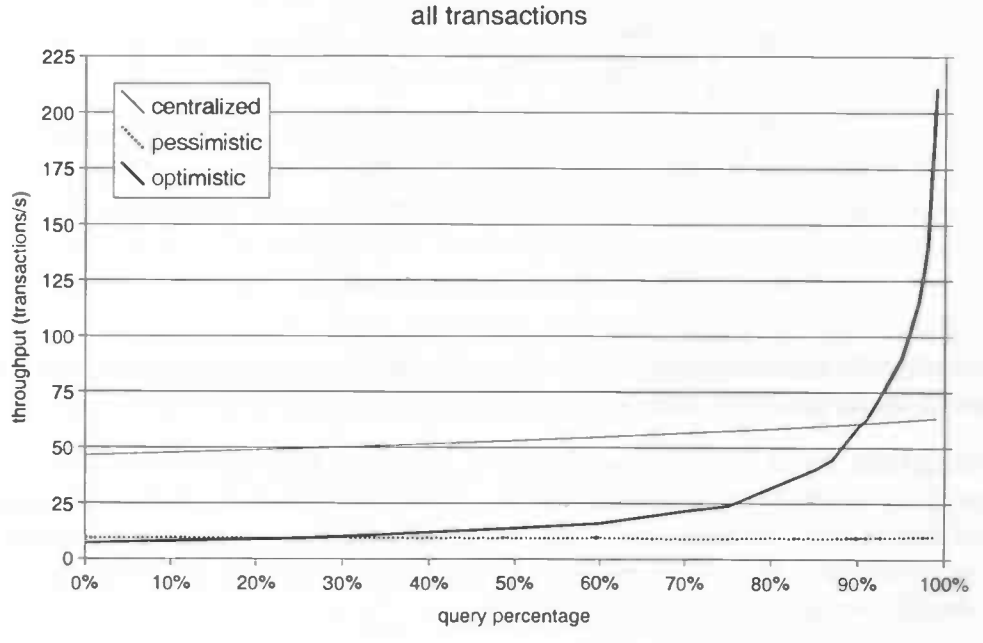

Figure 5.16: Throughput for centralized and replicated scenarios

The figure shows that the performance of pessimistic replication is very poor and that it is only better than active replication when the transaction load contains few queries. From 25% queries and up, optimistic replication outperforms pessimistic replication while offering the same degree of availability. From 93% queries and up, optimistic replication starts performing much better than a traditional centralized database system, at the same time guaranteeing higher availability than this traditional system.

Conclusion In slow, large-scale networks (e.g., the Internet), optimistic replication outperforms pessimistic replication for most transaction loads. Furthermore, in situations with very few update transactions, optimistic replication can achieve true load balancing: it performs better than a centralized database system subjected to the same load.

However, when update transactions form at least 10% of the load, optimistic replication does not scale well from a performance point of view. The reason is that the communication overhead grows significantly when the number of replicas increases. Another problem of optimistic replication is that the abort rate increases when replicas are added, even when the overall load on the system remains constant. This problem is typical for optimistic replication and does not depend on the speed of the network.

5.4.6 Limitations and improvements

First, this section summarizes to what extent the results obtained match the characteristics predicted during the qualitative comparison (see section 3.7). Then it discusses the limitations of the approach taken, considering (1) the characteristics of the transaction processing load and (2) the way the Database Replication Prototype was implemented.

Qualitative vs. quantitative comparisons The measurement results presented in this chapter confirm most of the performance characteristics predicted for both techniques in subsection 3.7. However, we have not been able to quantify the impact of two factors: the amount of processing in addition to the processing performed on a centralized system, and the determinism requirements. The reason for this is the slowness of the Atomic Broadcast implementation: all other processing is largely hidden behind it. To measure the two factors, the Atomic Broadcast implementation should be replaced by a much faster one.

Characteristics of the transaction processing load In the experiments, we constrained the transaction load quite severely to look at the most interesting aspects within the available time. However, the results presented are likely to change when the following limitations are relaxed: low transaction interference, small number of operations per transaction and small database size. Hereafter, we discuss why the results would be different and we give directions on how the optimistic technique could be improved to accommodate the modified transaction characteristics. We take the perspective of the optimistic technique because this technique is the most promising.

low transaction interference Transaction interference is low in our experiments, as reflected by the relatively low abort rate (see Figure 5.15). This is an oversimplification because in most applications, a small number of database items (the *hot spot*) is accessed very frequently while a large number of database items is rarely accessed. In OR, transactions that access the hot spot will likely have a high abort rate and thus progress only slowly. PR may perform better in such cases.

Since the pessimistic technique is not sensitive to abort rates, an idea is to mix the best of both techniques: transactions that access the hot spot are ABCast from the beginning of the transaction, like in PR, and other transactions are processed locally first and then ABCast, like in OR. A problem is that the the read and write sets of a transaction (and the hot spot of the system) need to be known in advance to be able to choose the approach. A variation that solves this is to use the pessimistic approach only for transactions that have been forcefully aborted some minimum number of times.

number of operations per transaction If a transaction contains more operations, the processing it takes longer. In OR, the probability increases that some remote update transaction forces the abort of a locally processing transaction. This would increase the abort rate and probably change the tradeoff between the replication techniques.

A solution to this is to have multiple versions [BHG87] of the database items on each replica. When an update transaction commits, its values are written to new versions of the items while the old versions are kept. Now, locally executing queries do not need to be aborted when they interfere with an update transaction that is committed. They can continue to lock and read the old values of the data items changed by the update transaction without violating serializability or getting in the way of the update transaction because of a locking conflict.

small database size The small item and database size in our experiments allow the Database Service to cache all items in memory, speeding up the read operations. While caching the whole database in memory may become more and more feasible considering the ever increasing memory sizes, contemporary systems still store less frequently accessed data on disk only. If reads take significantly longer, transactions submitted in OR take longer to be processed locally, leading to higher abort rates as described for the previous item.

Implementation Important limitations of the implementation are the slowness of sending an ABCast message and the slowness of thread handling in Java. Next, we discuss each limitation in turn.

Atomic Broadcast primitive The slow performance of OGS has influenced our results very much. This does not invalidate the results because the slowness of OGS matches that of an efficient Atomic Broadcast implementation in large-scale networks.

However, in Local Area Networks, the performance of some ABCast implementations is two orders of magnitude better than that of OGS (i.e., delivery in about 10 *ms* to large numbers of replicas instead of 1 *s* to only a few replicas). With such an efficient implementation, database processing will likely become the slowest factor in the system, leading to very different experimental results. PR would perform much better because it heavily relies on Atomic Broadcast but doesn't have any processing overhead or abort rates. But also OR would benefit in the case of update transactions. Also, the scalability of both techniques is likely to be affected in a positive way.

thread primitives Java's thread synchronization primitives are used to implement the blocking of transactions when a lock cannot be granted. These primitives are very expensive in Java 1.1, leading to quite slow behaviour when transactions are delayed and continue when the lock becomes available. A possibility is to switch to Java 2 and see if this changes things. This has not been done during the project because OGS is not compatible with Java 2.

In OR, an alternative is to forcefully abort transactions that cannot get a lock. When the lock becomes available soon after the forceful abort, this will work. However, when a lock is unavailable for an extended period of time, a lot of processing time may be wasted when the transaction is retried again and again.

Chapter 6

Conclusion

A replicated database system is a database system that is distributed over multiple machines, each containing identical copies of the database. The goal of replication is to improve the availability (by redundancy) and performance (by load balancing) of the database system.

Database replication is a topic that is being actively investigated, despite the fact that the subject has been around for more than 20 years. For performance reasons, commercial replicated database systems usually implement lazy replication techniques that are not correct. To users, this can be a problem because data retrieved from the database may exhibit inconsistencies. To counter this, eager replication techniques have been developed, which ensure that all replicated machines exhibit a consistent and correct behaviour to applications. The drawback of eager techniques is that until now, they lacked performance.

In this report, we have considered active replication techniques, which promise to offer better performance than traditional eager techniques. When the database is modified, active replication techniques use the Atomic Broadcast communication primitive to distribute the modifications and ensure consistency across replicas. We have focused on two active replication techniques, pessimistic active replication and optimistic active replication. These techniques were compared qualitatively as well as quantitatively.

6.1 Contributions

The main contributions of this Masters thesis can be split into four parts:

Replication Framework The Replication Framework is an object oriented framework for implementing replication techniques. By defining components that are common for all techniques, new techniques can be added easily. Furthermore, the performance and the requirements of the techniques can be compared on equal grounds.

The framework defines five components: Database Service, Group Communication Service, Replication Manager, Client and Operations. The Replication Manager is the component that needs to be adapted for every technique. The other components stay the same across techniques.

Systematic analysis of pessimistic and optimistic replication The differences between pessimistic and optimistic active replication are analyzed in detail and the performance characteristics of both techniques are compared in a qualitative manner. These qualitative comparisons are independent of a particular implementation.

Database Replication Prototype The Database Replication Prototype is a proof of concept implementation of pessimistic and optimistic active replication. The prototype conforms to the Replication

Framework and is therefore an extensible basis to which additional techniques can be added. Furthermore, it offers measurement facilities that support the quantitative comparison of the replication techniques.

In the Database Replication Prototype, the Database Service is implemented by the commercial POET centralized database system. The Group Communication Service is implemented by OGS, an Object Group Service built on top of CORBA. The prototype proves that the two replication techniques can be made to work using off the shelf components (POET and OGS) within a period 7 months.

Performance Results The performance of the Database Replication Prototype was measured for pessimistic and optimistic replication. Various performance indicators were considered: the mean response time, the throughput and the abort rate. By conducting many experiments with varying transaction characteristics, the behaviour of the techniques was measured in various situations.

The results show that in large networks, optimistic active replication outperforms pessimistic active replication, especially when the transaction processing load contains many queries (that is, many read-only transactions). Furthermore, optimistic active replication is shown to be partly scalable: in certain cases, it can handle a higher load than a centralized database system, even when the network is slow.

6.2 Related work

The DRAGON project [DRA98] tries to create eager replication techniques that reach performance levels comparable to those achieved by lazy replication techniques. Two main directions currently pursued in the context of the project are:

- The Postgres-R database system: a replicated relational database system [KA00]. The goal is to integrate replication inside the existing PostgreSQL database system and make the system scale well in Local Area Networks. In this approach, the replication technique can directly access the internal structures maintained by the storage manager, transaction manager and lock manager of the host database system.

  The advantage of Postgres-R is that it does need to duplicate any internal database structures. The drawback is that the database engine cannot be replaced easily. Contrast this with the Database Replication Prototype which sees the Database Service as a black box. This results in duplication of structures, i.e., locking tables and the read and write operations of executing transactions. However, the prototype is independent of the database system that happens implement the Database Service.

- A simulator for eager replication techniques (work in progress). This simulator determines the performance of replication techniques for different transaction loads, hardware configurations and software configuration. The use of a simulator instead of an actual setup such as the Database Replication Prototype has several advantages:

  - It produces results faster.
  - It can be configured according to various hardware/software configurations.
  - It allows replication techniques to be added easily because only the essential characteristics of a replication technique need to be modeled. Details that do not affect performance can be left out.

The main drawback of using a simulator is that it might be based on wrong assumptions, leading to performance results that are impossible to achieve in a physical system. This can be countered by comparing and tuning simulator results to the results obtained in actual systems (for example, the Database Replication Prototype).

6.3 Future work and research

Next follow several suggestions to improve the Database Replication Prototype and to gain additional insight into the various replication techniques. The suggestions are listed by increasing difficulty of realizing them.

Replacing OGS by another implementation When OGS is replaced by an implementation of the Group Communication Service that offers better performance, results can be obtained that are relevant for Local Area Networks. It would be interesting to compare these results to those of Postgres-R, to assess how much Postgres-R gains from its integration with the database service.

More replication techniques The modular structure of the Database Replication Prototype allows new techniques to be added and compared easily. One could add more replication techniques based on Atomic Broadcast, but it is also possible to add other types of eager techniques. In principle, nothing prevents lazy techniques from being added, but comparing these will be difficult because they tend to fulfill different specifications.

Additional performance indicators To pinpoint more clearly what is happening inside the system during transaction processing, additional performance indicators could be measured. Examples of performance indicators that would be interesting to add: execution time of operations, number of delays due to locking conflicts, and average amount of time transactions are delayed for locks. Furthermore, it would be useful if all measured data could be linked back to individual transactions. This way, the performance indicators can be correlated. For example, one could examine the relation between the amount of time a transaction is delayed and the risk that a transaction is forcefully aborted. A problem of collecting lots of measurement data is that this may slow down the system's performance.

Changing the interface to the Database Service The Replication Manager could be more efficient if it had direct access to certain low level data structures in the Database Service. The solution adopted by Postgres-R is to integrate replication into a particular database system. Because this makes it impossible to change the database system, we propose a different solution: change the specification of the Database Service and add access to exactly those low level structures needed by replication techniques. This way, the implementation of the Database Service remains interchangeable, but duplication is avoided. Before specifying the interface, one would have to do extensive research to determine which low level data structures and operations are needed by the various replication techniques.

Bibliography

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

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DMS98] X. Défago, K. R. Mazouni, and A. Schiper. Highly available trading system: Experiments with CORBA. In N. Davies, K. Raymond, and J. Seitz, editors, *Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 91–104, The Lake District, UK, September 1998. Springer-Verlag.

[DRA98] Information & Communcations Systems Research Group, ETH Zürich and Laboratoire de Systèmes d'Exploitation (LSE), EPF Lausanne. *DRAGON: Database Replication Based on Group Communication*, May 1998. http://www.inf.ethz.ch/department/IS/iks/research/dragon.html.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[Fel98] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.

[GS97] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[KA] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*. To appear.

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

[PGS99] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. Technical Report SSC/1999/008, École Polytechnique Fédérale de Lausanne, Switzerland, March 1999.

[POE97] POET Software Corporation. *POET Java SDK Programmer's Guide*, 1997. Version 1.0.

[SR96] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[TPC94] Transaction Processing Performance Council. *TPC Benchmark A*, 1994. Revision 2.0.

[UDS00] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proceedings of the 9th IEEE International Conference on Computer Communications and Networks (IC3N 2000)*, October 2000.

[WPS99] M. Wiesmann, F. Pedone, and A. Schiper. A systematic classification of replicated database protocols based on atomic broadcast. In *Proceedings of the 3rd Europeean Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira Island (Portugal), April 23–28, 1999. BROADCAST Esprit WG 22455.

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

[WPS+00b] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of* 20^{th} *International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, R.O.C., April 2000. IEEE Computer Society Los Alamitos California.