Diploma Project

Database Replication on Top of Group Communication

A Simulation Tool

Stefan Pleisch

February 21, 1997

Supervisors:

Prof. Dr. H.-J. Schek
Dr. G. Alonso
B. Kemme

Department of Information Systems
Database Group
Swiss Federal Inst. of Technology (ETHZ)
CH-8092 Zurich

Prof. Dr. A. Schiper
Dr. R. Guerraoui

Computer Science Department
Operating Systems Laboratory
Swiss Federal Inst. of Technology (EPFL)
CH-1015 Ecublens

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Replicated database systems (RDBS) provide two major advantages compared to traditional client-server based databases. They increase the availability of the data, since data items are stored on multiple nodes. Even when node failures occur, the data can still be accessed. The second advantage is the locality of data. In replicated database systems the node closest to the user usually executes a query. This leads to considerably shorter response times for read-only transactions. Updates, however, must be performed at every node. To guarantee the consistency and integrity while allowing transactions to run in parallel and spread across several nodes, existing distributed DBS use locking-based concurrency control protocols to synchronize data access and two-phase locking to guarantee a deterministic behavior of a transaction on all nodes.
However, up to now replication is not considered in update intensive applications.

The distributed systems community has proposed a certain number of group communication primitives [ADM92], that allow a total order of events in the entire system. The atomic broadcast [ChT93], for example, guarantees the all-or-nothing property. This means that either all correct nodes receive a message or none at all. In addition the atomic broadcast imposes a total order on the messages it delivers.
It is not until recently, however, that research has tried to use these primitives also in database systems.

In this work we have designed and implemented a simulation tool which facilitates to study the performance of replication control protocols that use atomic broadcast for synchronization. The simulation tool allows to identify the bottlenecks in such a database and to understand the implications of the broadcast primitive onto database performance.

This paper is organized as follows. In the next chapter we outline the objective of this project. Then we define a model for the atomic broadcast and the replicated database. Based on this model we explain the serialization protocol which has been used in the simulation tool. Chapter 6 then presents the conception of the replicated database simulation tool. After the conception of the RDBS simulation tool, performance measurements of the RDBS are shown. Then, chapter 8 explains some implementation issues.
Finally, the paper closes with some conclusions to this project and suggestions for further work.
The result tables of the performance measurements and the code are contained in the appendices.

In this paper important words are *emphasized*, while names taken from the code are written with font sans serif.

Chapter 2

Objective

The goal of this project is to design and implement a simulation tool in order to test database replication protocols using broadcast primitives for communication.

Since the broadcast primitives are usually studied by the distributed systems community and the database issues by the database community, this diploma project tries also to merge some aspects of these two communities together.
We expect to get some indications about the realizability of such databases as well as about their performance.

Our implementation is based on an atomic broadcast algorithm suggested by Chandra/Toueg [ChT93] and a replication control protocol proposed in [AAE96], that takes into account and relies on atomic broadcast. The general architecture of such a replicated database system can be seen in figure 2.1.

The resulting prototype should cope with the following requirements:

1. it has a modular design and implements the most important components of a replicated database system. It should be easy to exchange single components or add new components to the system. In particular it should be possible to take over some of the components and incorporate them into a real system.
2. provides a highly parametrisied implementation in order to allow exhaustively testing of a variety of configurations.

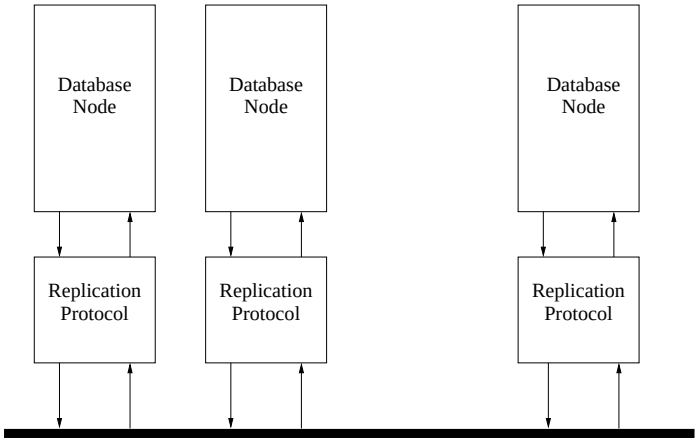

Figure 2.1: Distributed database relying on a replication protocol

3. finally, performance analysis of the system should be possible, for the broadcast primitive as well as for the database system based on the atomic broadcast.

Chapter 3

Atomic Broadcast

This chapter provides a summary of the theory developed in [ChT93]. We first present the model this chapter is based on in section 3.1. Then we present a broadcast algorithm, called *atomic broadcast*, which has been suggested by [ChT93]. It allows to broadcast messages to all processes in a system and guarantees that all correct processes receive all these messages in the same order. We will also see that the atomic broadcast problem can be reduced to the consensus problem using failure detectors (section 3.4).

3.1 The Model

The system we consider consists of a set of N *nodes* which communicate with each other via message-passing across a communication network. A *communication channel* allows to exchange messages between a pair of nodes. We require that this communication channels are *reliable*. Reliable communication channels do not loose any messages, even in the case when the message reception buffer of the destination node is full.
In the following, we will consider process, node and site as equivalent notions.

In a *synchronous model of computation* the transmission delay of messages is borned by $\triangle T$ and we dispose of synchronized clocks. However these requirements are very restrictive. Therefore we look at a *asynchronous computation model*. An asynchronous computation model does

not impose any bound on message delay, neither on computation time of processes. In addition we can not rely on synchronized clocks.

In a real system, failures can occur and communication protocols should be able to handle them. We can distinguish several failure types. In a *crash failure* environment we suppose that a process stops all computation and communication. This is the failure type handled by most of fault-tolerant applications including the one described in this work. It is also the easiest one to handle. In particular, we exclude the situation where the process has an accidental behavior, i.e. returns some completely arbitrary results for instance (*arbitrary-failure*), neither do we consider *omission-failure* or *value-failure*. An omission-failure occurs when a process simply does not respond to requests, whereas in value-failure it returns an incorrect output relatively to the specification. A further failure type that is important to handle are *net-crash-failures*. Ideally, at least one partition should be able to continue its work. However, since we assume reliable communication channels, we do not consider this failure type either.

This project considers three different types of communication, called *broadcast*, *point-to-point* and *send-to-all* communication. When broadcasting a message m, m is sent to all nodes in the system. If not all nodes are in the destination set of a message, we speak of a *multi-cast*. We exhibit point-to-point communication when a message is sent from a process p_i to another process p_j $(j \neq i)$. Send-to-all communication is equivalent to a point-to-point communication from a process p_i to all processes in the system.

3.2 Required Properties

Informally the atomic broadcast primitive has to fulfill two properties, namely:

1. the all-or-nothing property
2. the messages arrive in the same order at all correct nodes

The first property states that either a message m is received by all correct processes or by none at all. It may not occur that some processes receive m whereas other do not.

The consensus algorithm is based on a reliable broadcast. A possible algorithm for this broadcast is presented in the next section.

3.3 Reliable Broadcast

The basic communication primitive of our communication hierarchy is the reliable broadcast. It guarantees that all messages broadcast by correct processes are delivered and that all correct processes deliver the same messages. Spurious messages are thereby ignored. Therefore the reliable broadcast fulfills the all-or-nothing property. The reliable broadcast is defined by the two primitives *r-broadcast(m)* and *r-deliver(m)* (see figure 3.1).

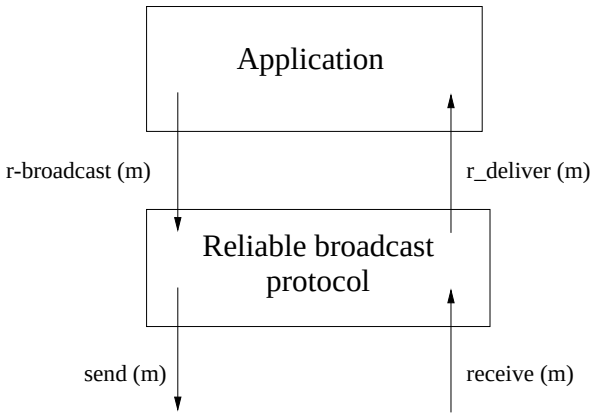

Figure 3.1: R-broadcast protocol stack

Formally, the reliable broadcast protocol has the following properties:

RB1 **Validity**: If a correct process r-broadcasts a message m, then it eventually r-delivers m.

RB2 **Agreement**: If a correct process r-delivers a message m, then all correct processes eventually r-deliver m.

RB3 **Uniform integrity**: For any message m, every process r-delivers m at most once, and only if m was previously r-broadcast by *sender(m)*.

In the absence of property RB1 every communication protocol that never delivers any messages would also comply with the definition of a reliable broadcast.

Here is the outline of a possible implementation of the reliable broadcast:

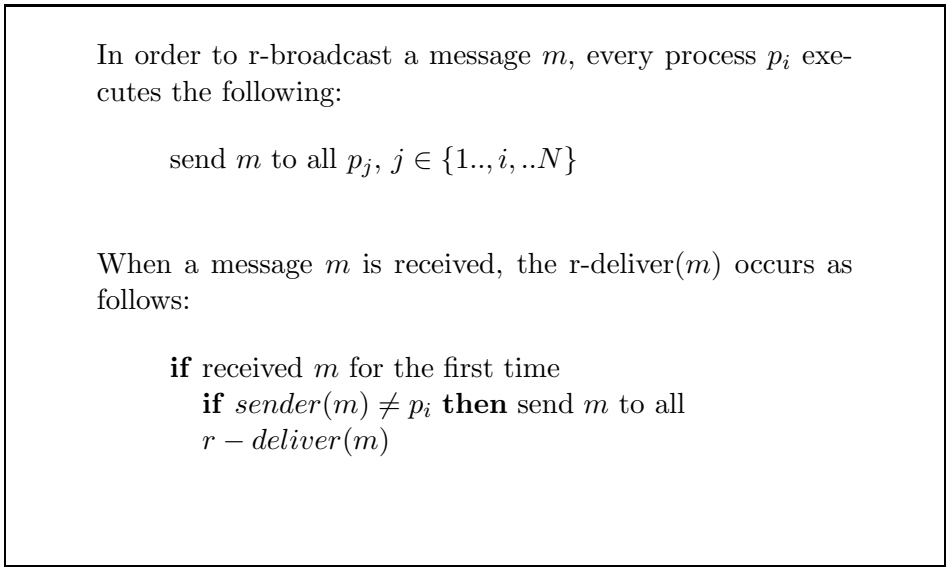

Figure 3.2: Reliable broadcast algorithm [ChT93]

A drawback of the reliable broadcast algorithm is the number of messages sent. Every time a message m is broadcast to N processes, N^2 messages are sent in fact. Depending on the underlying communication facility, this could prove to be a serious bottleneck.

We can also understand now why we need reliable communication channels. Imagine a process p with a full message reception buffer. It will discard all message which arrive during the time Δt its buffer is still full. If Δt is great enough, p will discard every message belonging to the reliable broadcast of a message m. Therefore process p does not receive m, which is clearly a violation of the reliable broadcast properties.

3.4 Consensus Problem

It can be shown that there exists an equivalent problem to the atomic broadcast, namely the consensus problem. This equivalence arises from

the fact that each algorithm can be reduced to the other [ChT93], meaning that the first algorithm is solvable when the second is and vice-versa. In other words a solution for one problem also solves the other one. Therefore, having solved the consensus problem allows also to find an algorithm for the atomic broadcast.

The consensus is a well-known problem, whose properties are specified in subsection 3.4.1. Its solution relies on the notion of failure detectors, which will be presented in subsection 3.4.2.

3.4.1 Specification

The consensus problem is defined as follows:

- N processes n_i that all propose one value v_i
- the processes reach an agreement on a value v, $v \epsilon \{v_i\}$

All correct processes propose a value v_i and must reach an unanimous and irrevocable decision on some value that is related to the proposed values.
Formally the consensus problem is specified by the following 4 properties:

- Termination: every correct process eventually decides some value.
- Uniform integrity: every process decides at most once
- Agreement: No two correct processes decide differently
- Uniform validity: if a process decides v, then v was proposed by some process

However, Fischer/Lynch/Paterson [FLP85] have shown in 1985 that there exists no deterministic algorithm to solve the consensus in an asynchronous system that is subject to even a single crash failure. This is known in literature as the *FLP-impossibility*. Essentially the impossibility for the consensus and the atomic broadcast stem from the difficulty of distinguishing an actually crashed process from a very slow one.

In 1991, Chandra and Toueg suggest therefore an extension to the asynchronous model. They introduce the concept of *failure detectors* [ChT93], which allows to circumvent the FLP-impossibility.

The following subsection outlines the properties of the failure detectors.

3.4.2 Failure Detectors

A failure detector is a model that is attached to every process. It can be wrong. However, if the set of failure detectors satisfies certain properties, the consensus problem can be solved. There are two completeness and four accuracy properties. Each failure detector $\mathcal{D}$ fulfills one property of each of the two classes.

Completeness:

- Strong completeness: Eventually every process that crashes is permanently suspected by every correct process
- Weak completeness: Eventually every process that crashes is permanently suspected by at least one correct process

These properties of completeness, however, are not sufficient by itself to get useful information about failures. The trivial situation where every process permanently suspects every process also satisfies strong completeness, but is clearly not of any practical use.

A failure detector must therefore also satisfy some accuracy property that restricts the mistakes that it can make.

Accuracy:

- Strong accuracy: No correct process is ever suspected
- Weak accuracy: at least one correct process (same for all) is never suspected
- Eventual strong accuracy: there is a time after which no correct process is ever suspected

- Eventual weak accuracy: there is a time after which at least one correct process (same for all) is never suspected

The combination of the different accuracy and completeness properties lead to 8 classes of failure detectors. For the purposes of this project we need only the failure detector $\diamond S$, which satisfies strong completeness and eventual weak accuracy.

Chandra/Toueg [ChT93] propose the algorithm displayed in figure 3.4 to solve the consensus problem. They have shown that the $\diamond S$ failure detector is enough to solve the consensus. The consensus algorithm makes the assumption that not more than f processes fail, where f ¡ $\lceil \frac{N}{2} \rceil$.

3.5 Atomic Broadcast

Finally we discuss how the atomic broadcast can be reduced to consensus. Formally the atomic broadcast is a reliable broadcast with the following additional property:

TO4 (Total Order): If two correct processes p_i and p_j $(i \neq j)$ deliver two messages m and m', then they deliver them both in the same order.

All correct processes will therefore deliver the same sequence of messages.

Chandra/Toueg suggest to use three different tasks for the atomic broadcast presented in figure 3.3. The first one executes the a-broadcast, while the second task waits for messages to be r-delivered. The third task finally periodically starts the consensus when the *R_delivered* list is not empty.

However this could lead to rather difficult synchronization problems for real implementations. If every process only disposes of one port for the communication, the second and third task will sometimes access the same port simultaneously. While the former just expects to receive r-delivered messages from the reliable broadcast, the third task waits for some consensus messages. At this point a real system has to guarantee

Every process p_i executes the following:

$R\_delivered \leftarrow \{\}$
$A\_delivered \leftarrow \{\}$
$k \leftarrow 0$

To execute $a - broadcast(m)$:

$r - broadcast(m)$ $\{task\ 1\}$

$A - deliver(-)$ occurs as follows:

when $r - deliver(m)$ $\{task\ 2\}$
    $R\_delivered \leftarrow R\_delivered \cup \{m\}$
when $R\_delivered - A\_delivered \neq 0$ $\{task\ 3\}$
    $k \leftarrow k + 1$
    $A\_undelivered \leftarrow R\_delivered$ - $A\_delivered$
    $propose(k, A\_undelivered)$
    wait until $decide(k, msgSet^k)$
    $A\_deliver^k \leftarrow msgSet^K$ - $A\_delivered$
    atomically deliver all messages in $A\_deliver^k$
        in some deterministic order
    $A\_delivered \leftarrow A\_delivered \cup A\_deliver^k$

Figure 3.3: Atomic broadcast algorithm
[ChT93]

that all the necessary consensus messages are received by the third task. This is however not always very straightforward. A solution to this problem would be to use two different ports, namely one for consensus messages and one for r-delivered messages.

Every process p executes the following:

procedure *propose*(v_p)
$estimate_p \leftarrow v_p$ {$estimate_p$ *is* p*'s estimate of the decision value*}
$state_p \leftarrow$ *undecided*
$r_p \leftarrow 0$ {r_p *is* p*'s current round number*}
$ts_p \leftarrow 0$ {ts_p *is the last round in which* p *updated* $estimate_p$, *initially* 0}

{*rotate through coordinators until decision is reached*}

while $state_p = undecided$
  $r_p \leftarrow r_p + 1$
  $c_p \leftarrow (r_p \bmod n) + 1$ {*current coordinator*}

  Phase 1: {*All processes* p *send* $estimate_p$ *to the current coordinator*}
    send (p, r_p, $estimate_p$, ts_p) to c_p

  Phase 2: {*Current coordinator gathers* $\lceil \frac{(n+1)}{2} \rceil$ *estimates and proposes a new estimate*}
    if $p = c_p$ **then**
      wait until [for $\lceil \frac{(n+1)}{2} \rceil$ processes q: received (q, r_q, $estimate_q$, ts_q) from q]
      $msgs_p[r_p] \leftarrow$ {(q, r_p, $estimate_q$, ts_q) | p received (q, r_p, $estimate_q$, ts_q) from q}
      $t \leftarrow$ largest ts_q such that (q, r_p, $estimate_q$, ts_q) $\in msgs_p[r_p]$
      $estimate_p \leftarrow$ select one $estimate_q$ such that (q, r_p, $estimate_q$, ts_q) $\in msgs_p[r_p]$
      send (p, r_p, $estimate_p$) to all

  Phase 3: {*All processes wait for the new estimate proposed by the current coordinator*}
    wait until | received (c_p, r_p, $estimate_{c_p}$) from c_p **or** $c_p \in \mathcal{D}_p$ {*Query failure detector*}
    if [received (c_p, r_p, $estimate_{c_p}$) from c_p] **then**
      $estimate_p \leftarrow estimate_{c_p}$
      $ts_p \leftarrow r_p$
      send (p, r_p, *ack*) to c_p
    else send (p, r_p, *nack*) to c_p

  Phase 4: {*Current coordinator waits for* $\lceil \frac{(n+1)}{2} \rceil$ *replies. If they indicate that* $\lceil \frac{(n+1)}{2} \rceil$ *processes adopted its estimate, the coordinator r-broadcasts a decide message*}
    if $p = c_p$ **then**
      wait until [for $\lceil \frac{(n+1)}{2} \rceil$ processes q: received (q, r_p, *ack*) **or** (q, r_p, *nack*)]
      if [for $\lceil \frac{(n+1)}{2} \rceil$ processes q: received (q, r_p, *ack*)] **then**
        r-broadcast(p, r_p, $estimate_p$, *decide*)
endwhile

{*If* p *r-delivers a decide message,* p *decides accordingly*}
when *r-deliver*(q, r_q, $estimate_q$, *decide*)
  if $state_p = undecided$ **then**
    decide($estimate_q$)
    $state_p \leftarrow decided$

Figure 3.4: Consensus algorithm using $\diamond S$
[ChT93]

Chapter 4

Transaction Management on Replicated Database Systems

This chapter first introduces some notions of the formal model of distributed database systems (DDBS). In section 4.2 the transaction model used in this project is presented, first in a centralized database system, then for a replicated database system. Section 4.3 motivates the concurrency control. The notions presented so far will then be used to discuss a particular serialization protocol taken from [AAE96].

This chapter only highlights the basic notions used in the following chapters. For a more detailed presentation consider [CGM88, BHG87, San92, AAE96].

4.1 Distributed and Replicated Database Model

A *distributed database (DDB)* is a collection of *data items* [CGM88] that are distributed across several sites. Each data item is given a name, its address in our case, and a value. The *granularity* of data items could be a part of the disk, a record of a file or a field of a record. However we will not consider the granularity further on.

A *distributed database system (DDBS)* is a collection of *sites* connected by a communication network [BHG87]. Figure 4.1 shows such a DDBS. Each site is a centralized database, which stores a portion of the database.

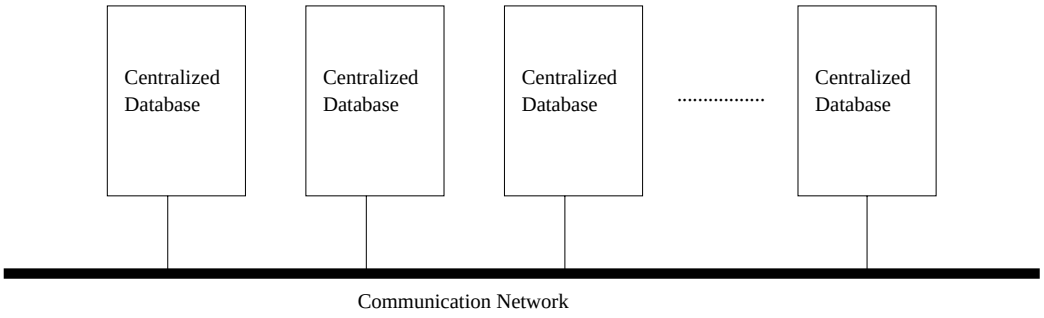

Figure 4.1: Distributed database

It is also referred to as a *node*.

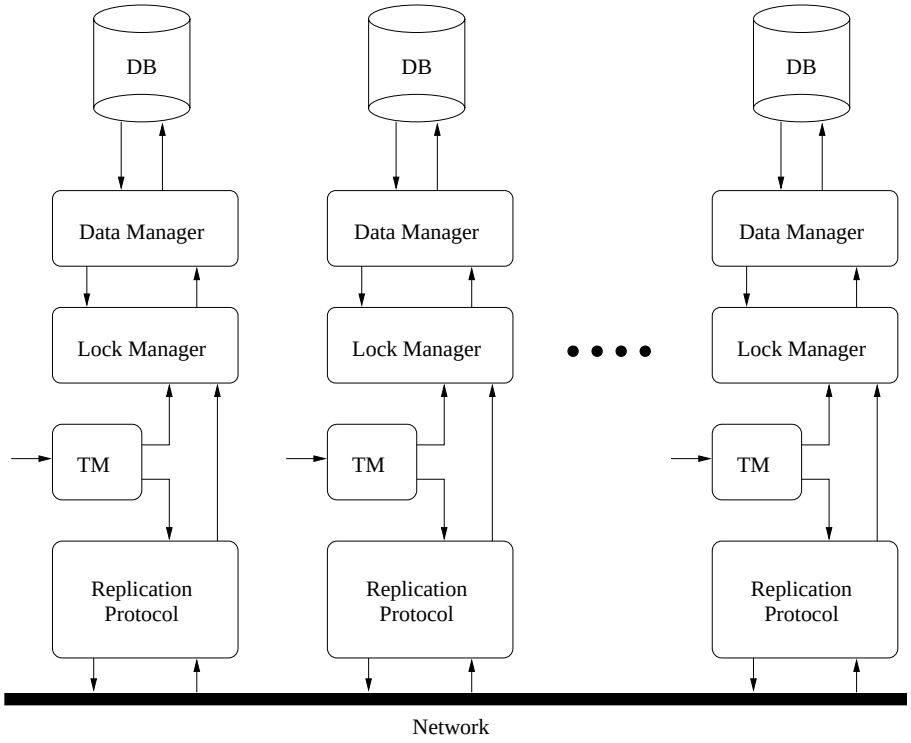

Figure 4.2: Database system architecture
[Alo96]

By a *replicated database (RDB)* we understand a distributed database in which some data items exists on multiple sites. The main reasons to replicate parts of the DB are to increase availability and performance of the DBS. Despite of site failures replicated data items can be retrieved as long as one copy is available on some node. In addition read operations are always executed on the nearest available copy. This leads to better

performance of the DBS.
An abstraction of the replication of data items can be achieved by considering the *logical* in contrary to the *physical* data items. Logical data items x are represented by multiple physical data items $x_1, x_2, ...$ that are stored at different nodes.

Figure 4.2 illustrates a RDBS. Each node contains a centralized DBS, which sits on top of a replication protocol. The transactions interact with the DBS through the *transaction manager (TM)*. The TM determines which site of the DDBS should process the operations of the transactions. *Lock manager (LM)* and *data manager (DM)* are responsible for actually executing the transactions. We will refer to this model and also present a more profound discussion of the different components of the RDBS in the following sections.

4.2 Transaction Model

4.2.1 In Centralized DBSs

Informally, a *transaction* T is an execution of a program that accesses a shared database [BHG87, CGM88]. It gets a consistent database state as input and, provided atomic execution, its output leaves the database again in a consistent state. A transaction is modeled as a set of operations, terminated with a delimiter operation, called *commit* or *abort*. A transaction commits if it terminates normally and its effects on the database are permanent. In contrary a transaction is aborted if it terminates anomalously. In this case all its effects on the database should be undone.
We distinguish two operations: *read* and *write* operations. The set of read operations of a transaction is called the *read-set*, whereas the write operations are called the *write-set*. The set of operations of a transaction is *partially ordered*. In our model we assume any order of the write operations while we impose a total order on reads. In addition all write operations can be executed after the read operations.
Further we only consider transactions which contain at least one write

operation.

4.2.2 In Replicated DBSs

A transaction T_j is issued at one node n_i of the RDBS. We say that n_i is the *initiating node* of transaction T_j. The index j thereby specifies the number of the transaction on that node. A read operation of T_j accessing the logical data item x will be noted $r_j[x]$. In a RDBS read operations should be executed locally in order to improve the performance of the DBS. On the other hand updates (write operations), called $w_j[x]$, are done on every node of the RDBS. $o_j[x]$ addresses either a read or a write operation of transaction T_j.

4.3 Concurrency Control

4.3.1 In Centralized DBSs

Because transactions T_i and T_j are executed concurrently on one node, they might access the same data items at the same time. This problem arises when operations of T_i and T_j *conflict*. Two operations issued by different transactions conflict, if they access the same data item and one of them is a write operation. T_i and T_j have to be executed *atomically* by a DBS in such a case, which means that:

- each transaction accesses shared data without interfering with other transactions
- if a transaction terminates normally, i.e. commits, its effects on the database are permanent. Otherwise (the transaction is aborted) no changes to the database are done at all.

In order to ensure atomicity of transaction execution, we need *concurrency control* and *recovery*. Concurrency control determines the order of conflicting operations $o_i[x]$ and $o_j[x]$, thereby also imposing an order on the corresponding transactions T_i and T_j. Recovery ensures that data is not lost even in case of a failure of the entire database. It will not be

subject of this work.

One way to ensure concurrency control is by using *locks* on data items. In [ElN94] a lock is a variable associated with data items which describes the status of that item with respect to possible operations that can be applied to the item. Different types of locks can be distinguished, namely *exclusive* and *shared* locks. If an operation holds an exclusive lock on a data item, no other operation can access this item. On the other hand a shared lock allows other operations to obtain also shared locks on the corresponding data item.

Concurrency control provides a *serialization order* to the transactions. Serializable execution order of transactions has been widely accepted as a correctness criteria in the database community. Informally the schedule of executing a set of concurrent transactions $\mathcal{T}$ is correct if it is equivalent to some serial order of the same set of transactions $\mathcal{T}$. In other words there must exist a sequence of the transactions $T_i \in \mathcal{T}$ such that executed one after the other, they are executed in the same order than when executed concurrently. If such a sequence of $T_i \in \mathcal{T}$ exists we say that the concurrent execution of $\mathcal{T}$ is *correct*.

4.3.2 In Replicated DBSs

An important issue in RDBS is preserving the *consistency* and *integrity* not only on each centralized DBS, but also over the whole DDBS. Since updates are performed on every node, we need to guarantee that all nodes obey the same serialization order. Interleaved execution of transactions on a RDBS therefore has to behave equivalently to a serial execution of those transactions on a one-copy DBS. This is called a *one-copy serializable* [BHG87] execution.

4.4 A Serialization Protocol

4.4.1 Introduction

In this project we will focus on one particular serialization protocol proposed in [AAE96]. This protocol relies on an atomic broadcast communication primitive with the properties specified in subsections 3.3 and 3.5 (*RB1-3* and *TO4*) in order to guarantee the correctness of the replicated database. It exhibits the advantage of reading one copy and writing all-copies of replicated objects.

The considered database is supposed to be *fully replicated* [AAE96], which means that every site stores a copy of all objects in the database. In addition the serialization protocol makes the assumption of a crash-failure (see section 3.1) environment.

4.4.2 Algorithm

A transaction, T_i, on node n_A executes as follows:

1. A read operation $r_i[x]$ is executed locally by obtaining a read lock on x.
2. A write operation $w_i[y]$ is deferred until T_i is ready to commit.
3. Before T_i commits, it broadcasts its deferred writes $w_i[x_1, ..., x_n]$ to all sites:
   (a) On receiving $w_i[x_1, ..., x_n]$, the lock manager on node n attempts to grant the write locks to T_i. If there are any transactions T_j with read locks on object x_i, the lock manager checks if T_j had already broadcast a commit. If the commit is indeed pending, the lock manager blocks until the commit is delivered. Otherwise T_j is aborted and T_i is granted the write lock.
   (b) After obtaining atomically all its write locks at S, the write operations at S are initiated and performed.

Once all the writes are executed successfully, the node that initiated T_i broadcasts T_i's commit. All node execute T_i's commit operation only after its delivery.

A formal proof of correctness of this protocol is given in [AAE96].

4.4.3 Properties

In a RDBS using this serialization protocol, a transaction can be in 5 different states, displayed in figure 4.3. It is either *running*, *waiting* for a lock, *committed*, *being aborted* or *aborted*. A transaction is committed if it terminated normally at all the nodes in the DBS and all the changes to the database are permanent.

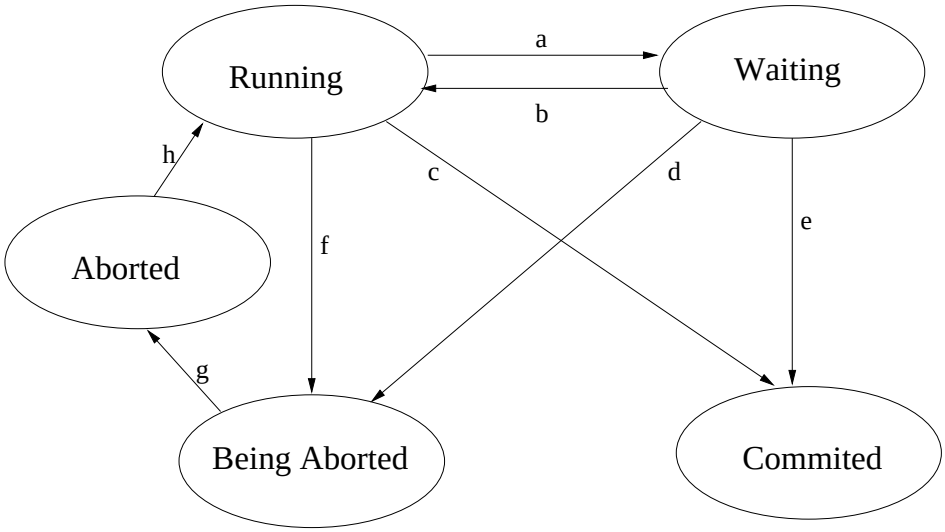

Figure 4.3: States of a Transaction

A transaction is in the being aborted state if it terminated anomalously. All its effects to the database are undone while it is in this state. Then it changes to the aborted state *(edge g)*.

The committed state is a terminal state, which means that a transaction will not leave it again. The other 4 states are transitional states (we suppose that an aborted transaction will be restarted again). Three actions might occur while a transaction T_i is in the waiting state:

- either a lock an operation of T_i is waiting for is released by another transaction $T_j (j \neq i)$ or the write-set of T_i has arrived at the node. The transaction T_i changes to the running state. *(edge b)*
- the commit message T_i has been waiting for has arrived and the transaction T_i commits. *(edge e)*
- the transaction has to be aborted because its execution interleaved with another transaction's execution. *(edge d)*

A transaction in the running state will become waiting if it is either blocked on a lock request *(edge a)*, totally executed and waiting for the transaction's commit or abort from the initiating node or waiting for the write-set. In a DDBS a running transaction can be aborted when a conflict arises on another node of the system. It passes therefore directly from the running state to the being aborted state. The same reasoning applies for the action *c*. A database node is still processing some write operation of the transaction T_i when it already receives the commit message for T_i.

In our implementation, a transaction T is committed on the initiating node as soon as all its operations are done. The locks are released and the lock manager sends the commit request for T to the transaction manager. The transaction manager then broadcasts T's commit to all nodes in the system. The user can be notified about T's commit only after the commit of T has been broadcast and has been received on the initial node again. Otherwise the user might get a incorrect response of the system. This situation arises, when the node crashes after having notified the user of T's commit, but before having broadcast the commit to all nodes. After a certain time the other nodes will then abort T, leaving the user with a committed transaction that, in fact, has been aborted in the DBS.
The advantage of this approach is that the locks hold by T on the initiating node are released as soon as possible.

The cost of one transaction execution is at least equal to the cost of two atomic broadcast operations. The first one is the transaction's write-

set that is broadcast to all nodes whereas the second atomic broadcast incurs from the global commit message. Every abort of a transaction T potentially increases the cost by two atomic broadcasts. If the abort of T occurs after node n_i has broadcast T's write-set, the transaction must be restarted leading to at least two more broadcasts. This implies a significantly higher transaction execution cost.
An advantage of this protocol is that deadlocks between write operations of different transactions do not occur. No global deadlock detection algorithm is therefore needed.
On the other hand conflicts between read and write operations are detected locally for all transactions. In such a case always the read operations are aborted. Since reads are done locally, the cost of redoing them do in general not include the cost of an atomic broadcast. These additional cost occur only if the write-set has already been sent, as seen before.

When the write-set of a transaction T arrives, the locks for these write operations are granted using *strict two-phase locking* [San92, BHG87]. All the locks are acquired in a single atomic step and are released only after the reception of a commit or abort message.

In the current project we use the serialization protocol in combination with the atomic broadcast protocol presented in the previous chapter.

Chapter 5

CSIM17

This chapter discusses some of the most important features of the simulation package *CSIM*. Since the current work is implemented using this package, CSIM's properties have influenced the conception of the simulator to a certain degree. It is therefore necessary to know the basic concepts of CSIM in order to understand certain design choices.
We start with a presentation of CSIM in the first section, thereby considering all the primitives which are important for this project. In the second part we look at the way a simulation can be started.

5.1 The CSIM Library

CSIM is a process-oriented discrete-event simulation package, copyrighted by Microelectronics and Computer Technology, Inc. 1985-1994. Its version CSIM17 can be used with both C and C++ code. It provides a library of routines that allows to build simulation tools very efficiently.
A CSIM program is based on a collection of processes that interact with each other. The CSIM library provides all the necessary synchronization facilities. In addition, a set of tools for performance measurements is already implemented and can be switched on.
Another important feature of CSIM is the notion of simulated time. The simulation can let time pass artificially. This allows the precise modeling of real systems.
In the following subsections CSIM's most important features for this project are briefly discussed.

5.1.1 Process

Processes are the active entities in a CSIM model. They seem to be executed in parallel, even though their code is processed sequentially on a single processor. Each process has its own runtime environment and access to global data. In addition every process can be assigned a priority.
Because the processes are processed sequentially it never occurs that two processes access simultaneously one variable. In this sense every process behaves like a coroutine. Using this knowledge would considerably simplify the simulation tool. However it would not allow to adapt the same conception for a real system. This is the reason why the RDBS simulator in general has been designed as if it would be running on a real system with multiple sites. There are certain shared simulation parameters that are not protected by any semaphores or locks. They are specific to the simulation, however, and would not enter into consideration in a real system. This and efficiency reasons were responsible for not protecting these shared variables.

In all graphics a process p will be represented as shown in figure 5.1.

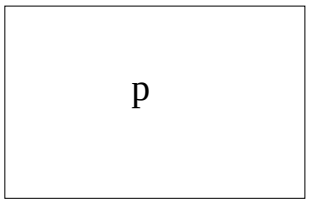

Figure 5.1: Representation of process p

5.1.2 Mailbox

Mailboxes are entities used for synchronization between two processes. A process sends a message m to process p by dropping m into p's mailbox. Process p might be blocked while waiting on a message to arrive in its mailbox. When m finally arrives, p is resumed and continues its execution. A mailbox therefore models a communication with *non-blocking send* and *blocking receive*.

A timed receive operation is also available. After a certain time specified as argument the blocked process resumes its activities.

In the following chapters a mailbox *mb* will be represented in graphs like:

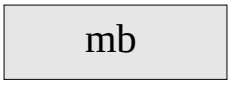

Figure 5.2: Representation of mailbox *mb*

A problem with mailboxes arises from the fact that a process is suspended when accessing an empty mailbox. Only when a message is dropped in the corresponding mailbox, the blocked process is activated and continues execution. Suppose that a process p receives messages from several mailboxes. If p checks periodically the mailboxes, it blocks on every empty mailbox. During the time p is suspended the messages in the other mailboxes can not be treated. This is clearly a situation which can not be accepted. A first solution provides the CSIM primitive *msg_cnt*, which allows to check the number of messages in a mailbox, without the risk of getting suspended. However, periodically check the mailboxes for their number of messages consumes a considerable amount of real CPU time, especially when the simulation is voluminous. Tests have shown that this solution is not applicable to this project. We will discuss a better solution in the following subsection.

5.1.3 Event

Events are other means for synchronization. They can be *set* (*signaled*) or *cleared*. A process waits on a cleared event and can resume processing as soon as the event is set (signaled). CSIM offers the possibility to test simultaneously a set of events. The signaled event with the lowest index is then selected first.

Using events in combination with mailboxes provides a neat solution to the problem of synchronizing access to multiple mailboxes. Every time a message is dropped in a mailbox, the corresponding event in the event-set is signaled and the process retrieves the messages from this mailbox. If all the mailboxes are empty, the process is suspended while waiting on

any of the events to be signaled. On one hand this consumes much less CPU time than periodically checking the message count, on the other hand the process can handle messages dropped in any mailbox.

5.1.4 Facility

A facility can be used to model a resource, for example a CPU, accessed by a process. It basically consists of a critical section, where only one process can be at a time. Processes are queued before accessing the facility in the order of process priority and of arrival time. In addition some basic primitives for performance measurements are provided by CSIM when using a facility. In particular, the facility utilization rate and the maximal, mean or minimal length of the facility's waiting queue can be retrieved.

5.1.5 Table

A table can be used to collect data and produce some statistics. In this context a table contains a statistical summary of all values which have been recorded.

5.2 Starting a Simulation

The CSIM library provides a special primitive called *sim*. This primitive can be used to start all the processes. By calling *sim* from the function *main*, command line parameters can be considered by the simulation.

Simulation time can pass by two statements. First, when a process uses a facility for a certain time. Second, a process can call a hold statement. It is then suspended for the time specified as argument to this statement.

Chapter 6

Conception Of The Replicated Database System Simulator

This chapter discusses the conception of the replicated database system (RDBS) simulation tool. The implementation will be based on the simulation package CSIM presented in chapter 5.

After defining the basic assumptions about the considered database, we focus on general issues of the architecture of the simulation tool. Whereas section 6.1 considers only the architecture on one node, we extent the simulation tool to model the whole RDBS on N nodes in section 6.2. We then proceed with a more detailed explanation of every component in this architecture. We begin with the simulation of the network in section 6.3 and then discuss the communication module in section 6.4. In a further step we proceed to the conception of the database system, beginning with the transaction manager followed by the interface adapter, the lock manager, the data manager, and the physical database.

6.1 Architecture on one Node

This section focuses on the architecture of the replicated database simulator on one node. The next section will then consider the entire simulator, i.e. the generalization to N nodes.

The architecture of the RDBS simulator has to fulfill the following constraints:

1. be the most adequate for the implementation of the selected serialization protocol
2. allow the exchange of one or more modules of the simulation tool with modules supporting other serialization or broadcast protocols

The second constraint addresses the fact that this simulation tool might be used for performance measures of various serialization protocols. It is therefore important that parts of the simulator can quite easily be replaced. However the more general an architecture is, the less efficient it might become for a particular protocol. In order to comply with both constraints we decided to use an object-oriented approach to this problem. By specifying a well-defined interface for every part of the system, the implementation of a part can be replaced with another implementation, provided that the interfaces are equivalent.
This considerations lead to the architecture depicted in figure 6.1.

Each white box in figure 6.1 specifies a *component* of the simulation tool. In order to simplify the representation, the communication module groups some other components together. It will be divided up into these components later.
While the communication module is responsible for the implementation of the atomic broadcast, the transaction manager starts the execution of the transactions and decides whether the operations are executed on the local node (read operations) or broadcast to all nodes (write operations). The interface adapter extracts the content of the broadcast messages and hands it over to the lock manager. The lock manager schedules the execution of the operations by the data managers. Each data manager accesses its own database.

Due to the object-oriented approach, every component is modeled by an object. These objects all export the primitive loop. When the simulation is started, a process initializes an object and calls its loop function. As the name of the function indicates, loop basically contains an infinite loop. In this sense the component gets its own thread of control. In the further discussion when we refer to components as processes we implicitly mean the objects, whose primitive loop has been called by a process.

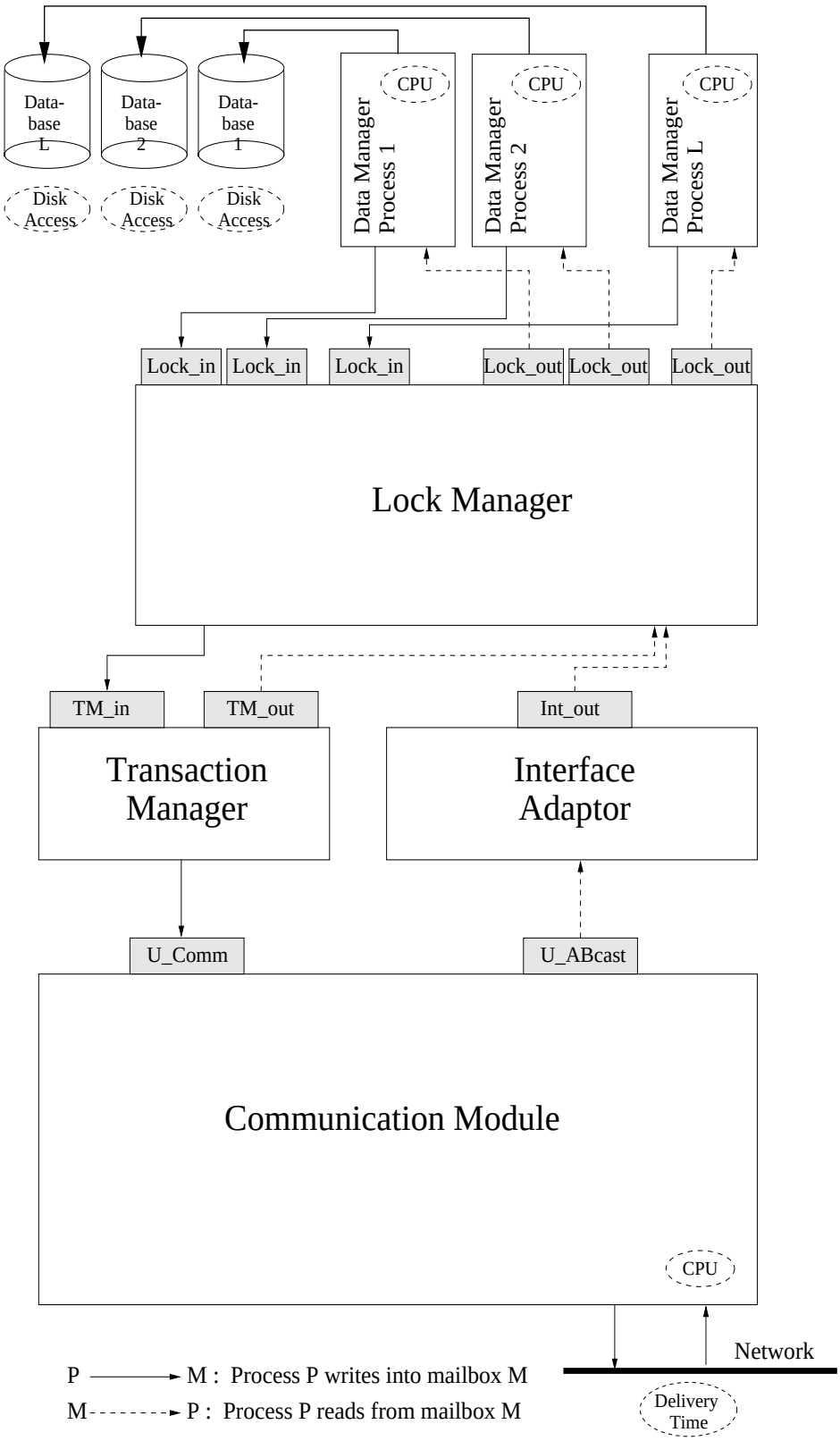

Figure 6.1: Architecture of the RDBS simulator on one node

Processes communicate by exchanging messages. The message exchange takes place at well defined interfaces, also called mailboxes, by simple send and receive primitives.
The name conventions for the mailboxes are the following. A mailbox which ends with '*in*' is a receiving mailbox of the component it belongs to, whereas '*out*' indicates a sending mailbox into which the output of the corresponding component is dropped. The names of the communication module's mailboxes are slightly different. U_Comm, where the U stands for upper, is the receiving mailbox, while the mailbox U_ABcast is the output mailbox of the communication module.

The ellipsoids in figure 6.1 specify the locations where simulation time passes. This happens for every disk access, for the CPU of the data managers, for the CPU used for sending and receiving messages and for the network, which models the delivery time of a message.

Most of the components of the RDBS simulation tool have been designed in order to run also on a real distributed system. Exceptions are most of the parameters used for performance measurements and the simulation of the network, which would not be needed in a real system anyway.

6.2 General Architecture of the RDBS Simulator

While the previous section dealt with architecture issues on one database node, this section provides a more general view of the simulator.

Figure 6.1 presents the components of the database simulator on one node. This architecture is the same for all nodes in the system. Since the simulation tool will be realized on one machine by using different processes, we can only speak of *virtual nodes* in this context. This virtual nodes simulate the database nodes in real RDBSs. In the following the term node will be used instead of virtual node.
In order to distinguish the components of one node from the others, every node is identified with a unique node ID. Upon initialization of the

system, every component gets the ID of its node. It then only communicates with components having the same node identifier. An exception is the network, which exists in a unique version and is accessed by all the nodes. Special treatment is also required for the data manager processes. Since every node has multiple data manager processes, they need to be identified differently. Section 8.7 addresses this subject.

6.3 Network

The nodes in a RDBS can communicate with each other through a network, which links all the nodes together. This project focuses on one particular type of network presented in subsection 6.3.1. Subsection 6.3.2 then explains how such a network could be simulated.

6.3.1 The Ethernet and the TCP/IP Protocol Family

As already discussed in section 3 the protocol for the atomic broadcast requires reliable communication channels. We therefore distinguish between the model of the network and the model of the communication protocol used on this network.
In a first step we now consider only the physical network. We decided to model the local area network (LAN) called *Ethernet*. It consists of a central communication bus. Every node is linked to this bus and inspects the messages in transit on the bus. The access to the bus is controlled in the following way: when a node is emitting a message, it checks whether there is a collision with another message. If two messages are emitted at about the same time, then both nodes detect the collision, stop emission, wait for an arbitrary delay and retry to send the message. Otherwise the node continues to send its message, which can then be received by all the nodes it was sent to.

The Ethernet itself, however, does not provide reliable communication, because it is not guaranteed that processes read all messages. For instance, they could discard messages from the internal buffer because of lack of buffer space. An additional communication protocol is needed to ensure this property. The only way to get these reliable channels on an

Ethernet is to use the *TCP* protocol. TCP defines the notion of virtual channels between two nodes and provides error checking and flow control on this channels.

These assumptions on the underlying network define the parameters of our simulation tool.

6.3.2 Network Model

As we have seen in the previous section we need to simulate the facilities provided by TCP running on an Ethernet. Our network layer therefore must define a primitive which exhibits reliable communication between two nodes, thereby at the same time respecting the access constraints to the communication bus.

The network is modeled by an object, which simply receives messages from the communication modules and sends them to the destination nodes. It does not know anything about the nature of the messages, except their destination node.
Every message is delayed for an exponentially distributed delay in order to model the time a message needs to be delivered from the sender to the receiver. In addition, before sending a message, the sending process waits for an exponentially distributed delay. This delay simulates the time needed to pack a message, to access the network and to emit the message. The necessary flow control is also taken into account in this parameter.
The receiving node also incurs a certain delay for error checking and flow control. This delay is also exponentially distributed.

Evidently there exists more precise simulations of real networks. One critique of the current simulation is that it does not precisely simulate what happens when the network is busy. In a real Ethernet the communication process waits and then retries after some time. The communication module tries to model this behavior with an exponentially distributed delay before sending the message.
The current model could also be extended to simulate the behavior of

the communication on a wide area network (WAN). A major difference to the current system would then be that multiple messages might be sent at the same time, due to efficient routing.
Another extension would be to integrate a way to delay messages for different time intervals.

6.4 Communication Module

The purpose of the communication module is to provide an atomic broadcast operation to the upper layers of the replicated database system, thereby using the network defined in the previous section. This includes on one hand the reliable broadcast communication primitive and on the other hand an ordered delivery of the messages respecting the all-or-nothing property. For the purpose of this project we have selected the atomic broadcast suggested in [ChT93] and explained in chapter 3. First subsection 6.4.1 looks at the general architecture of the communication module, before we proceed with discussing the conception of every layer in detail.

6.4.1 Architecture

The communication module is divided into 2 layers, the *communication layer* and the *ABcast layer*. The former provides the facilities for the reliable broadcast, whereas the latter contains the consensus algorithm and the atomic broadcast communication primitive.

In general, a major challenge for the architecture of any communication protocol stack is that each layer should be able to send and receive messages to and from its neighbor layers at any time. Since this ideal situation hardly ever occurs in a real system, due to resource and time constraints, the time the system is ready for sending and receiving messages should be as great as possible. This means that the system should never block when receiving a message. A possible solution for this problem is to implement every layer as a process. Each process disposes of two mailboxes, one for the communication with the upper layer and the

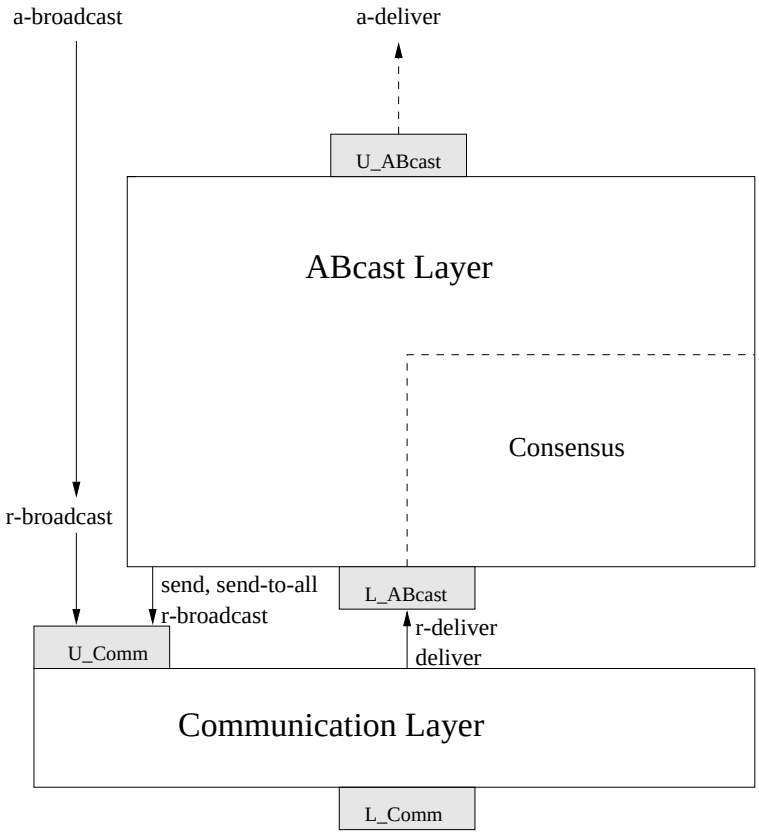

Figure 6.2: Architecture of the Communication Module

other for communicating with the lower layer.

The atomic broadcast is described by two processes, one for atomically broadcasting messages and one for receiving messages from the communication layer. The need for two processes incurs from the fact that the ABcast layer blocks when waiting for some consensus messages. Therefore it is necessary to have two independent processes for the sending and receiving of messages. However, since the sending process does nothing else than forwarding every message from the upper layer to the communication layer, it has been merged with the communication layer. This allows to save process-switches and therefore to increase the efficiency of the communication module.

Figure 6.2 shows the architecture of the communication module. Every message that has to be atomically broadcast is dropped into the mailbox

U_Comm, where it is retrieved by the communication layer and sent via r-broadcast over the network using the TCP simulation. The network drops the messages into the mailbox L_Comm of the receiving nodes. The communication layer then r-delivers them to the mailbox L_ABcast. Finally the messages are a-delivered in mailbox U_ABcast according to the properties of the atomic broadcast. The ABcast layer uses communication layer primitives for the consensus. The user interface of the communication module is defined by the input mailbox U_Comm as well as the output mailbox API_out.

Note that the consensus is part of the ABcast layer. Since it is tightly linked to the atomic broadcast, this architecture is reasonable.

6.4.2 Communication Layer

This layer basically implements five communication primitives: *r-broadcast* and *r-deliver*, *send*, *send-to-all* and *deliver*. R-broadcast and r-deliver define the reliable broadcast, send, send-to-all and deliver are used by the ABcast layer for additional messages needed to achieve the consensus. Send sends a point-to-point message, send-to-all sends a message to all nodes and deliver delivers the message to the upper layer.

Conceptually, the communication layer is implemented as follows:

- the upper layer has called a communication primitive (dropped a message in U_Comm):

```
for every message currently in mailbox U$\_$Comm do
   if message type is 'r-broadcast'
      for every node in the DBS
           send message to it
       end for;
   else if message type is 'send'
       send message to the destination node
   else if message type is 'send-to-all'
```

```
            for every node in the DBS
                send message to it
            end for;
        end if;
    end for;
    wait for a new call to occur;
```

- the network has dropped a message in L_Comm:

```
    for every message m in L$\_$Comm do
        if message m is a 'r-broadcast' message
            if m received for the first time
                if n was not initiator of m
                    for every node in the DBS
                        send m to it;
                    end for;
                end if;
                drop m in L$\_$ABcast
            end if;
        else if message type is 'send' or 'send-to-all'
            drop the message in L$\_$ABcast
        end if;
    end for;
    wait for a new event to occur;
```

6.4.3 ABcast Layer

The ABcast layer implements the atomic broadcast protocol presented in chapter 3. The primitive a-deliver basically defines the behavior of this layer. The facilities of the ABcast layer are ensured by two objects. The first object implements the consensus. Its principal primitive is the function propose which allows to reach a consensus decision among all correct nodes. The second object implements the reception part of the atomic broadcast algorithm. It starts the consensus when necessary and,

once the consensus has reached a decision on a group of messages, it a-delivers this group to the upper layer in a deterministic order. Figure 6.3 illustrates the activities of the ABcast layer.

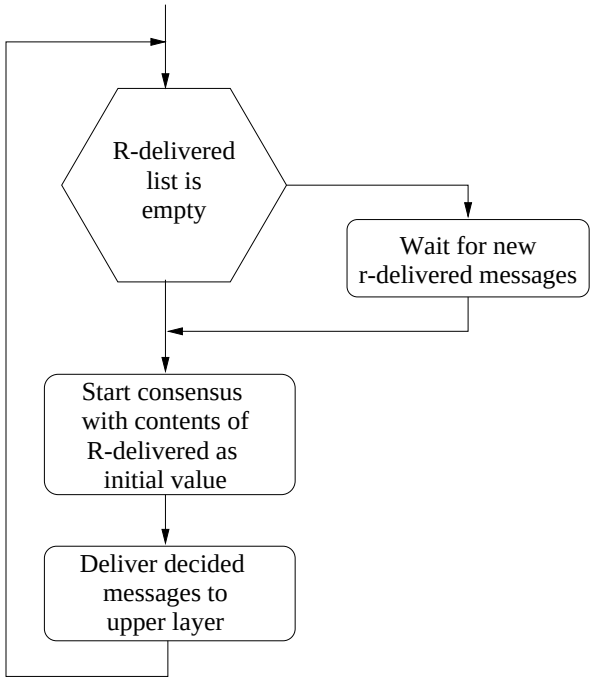

Figure 6.3: Flow chart of activities of the ABcast layer

Both objects access the lower mailbox L_ABcast. An adequate message handling procedure guarantees the correct delivery of the messages to one of the two processes.

For the realization of a-deliver it is suggested in [ChT93] to use two tasks (see section 3.5). In the current architecture of the simulator, only one task is assigned for this purpose. There are multiple reasons for this architectural choice. First of all, since the two processes would access shared variables, those variables would need to be protected. In particular, concurrent access to the R_delivered list (see algorithm in figure 3.3) would have to be excluded with semaphores or some other synchronization facility. Another drawback of the use of two processes is that they would need two different ports (mailboxes) for their communication. This implies that every consensus message needs to be tagged such that the communication layer is able to deliver it to the correct mailbox. This violates the layered architecture, because the lower layer needs knowledge

about the type of messages (consensus messages or others) it delivers. In order to avoid this problem, a third process could be introduced, which would then charge itself with the task of delivering the messages to the corresponding processes.
Since all the messages are sent over the same network, the only gain we get from using two different tasks is the parallel processing of the received messages. The time needed for message handling is very short, however, and therefore neglected in the simulation.

6.5 Transaction Manager

The transaction manager (TM) is responsible for handling the execution of the transactions. The transactions are supposed to have the characteristics presented in section 4.2.
According to the serialization protocol discussed in section 4.4, read operations are executed sequentially and only on the local node. Write operations of a transaction, on the other hand, are all broadcast in one message to all nodes in the system, once all read operations have been performed.

A parameter, set by the user, determines how many transactions are processed at the same time by the TM. At the start of the execution, this amount of transactions is started. Then, only when a transaction commits, a new one is started.

In addition, some mechanism has to be provided in order to distinguish operations of an aborted transaction from the operations that are sent when the transaction is restarted. There are several approaches possible. For instance every time a transaction is restarted, a new number is assigned to this transaction. Another possibility is the introduction of round numbers. This round number is increased every time a transaction is aborted.

Figure 6.4 illustrates the activities of the transaction manager. For each read operation of a transaction the TM sends a read request to the lock manager. Once all reads are executed, the transaction's write-set is broadcast by sending it to the communication module.

The replies from the lock manager can either be commits or aborts. If a commit has been received, it is broadcast to all other nodes. Then the transaction is removed from the list of transactions and a new one is read from the input file.
If a transaction has been aborted then different measures have to be taken according to the state the transaction processing is in:

- if the transaction's write-set has already been broadcast, then an abort message for this transaction is also broadcast. In addition, the transaction's round number is incremented and the transaction is restarted by sending the first read operation again to the local lock manager. If the transaction had contained no read operations, it would not have been aborted.
- if the transaction's write-set has not yet been broadcast, increment the round number of the transaction. Then restart the transaction processing by sending the first read operation to the local lock manager again.

The transaction manager will never receive an abort for the write-set of a transaction. This is due to the fact that only read operations of *local* transactions are aborted. Once the write-set has acquired its locks *on its initiating node* and the transaction has not been aborted before, it will commit.

6.6 Interface Adaptor

The only purpose of this layer is to receive the a-delivered messages from the communication module and to convert them to another object which is then understood by the transaction manager. A-delivered messages can be write-sets, commits or aborts of transactions.

6.7 Lock Manager

The lock manager (LM) is responsible for the scheduling of the operation execution. It decides whether a transaction has to be aborted or is ready to commit. In addition, it distributes the operations to the data manager processes.

First the general structure of the LM is presented. Then a subsection is devoted for the discussion of every object in this component.

6.7.1 General Structure

The lock manager basically consists of three objects: *lock manager* object, *lock table* and *transaction handler* (see figure 6.5). The lock table is used to manage the lock requests while the transaction handler stores all transactions whose operations are in process. These two objects are accessed by the lock manager object.

6.7.2 Transaction Handler

Implemented as an object the transaction handler provides the necessary primitives for keeping track of the transactions currently in process in the LM. In addition it can be queried about the status of a transaction, i.e. whether a transaction already committed or aborted.

6.7.3 Lock Table

The lock table is implemented as a hash table of size n. Each entry in the hash table points to a list of lock entries. Each lock entry is identified by the address of the data item whose lock requests it handles. It contains two lists, one linking all operations that got a lock granted and the other one linking all waiting operations. If these two lists are empty, the lock entry is removed. Figure 6.6 illustrates such a lock table.

Table 6.1 shows when a lock on a certain data item can be granted. A lock is granted when a 'Y' stands at the corresponding position in

the table. It cannot be granted with a 'N'. The columns contain the locks currently granted while the rows show the type of incoming lock requests. The label *Done* indicates that the execution of an operation has been finished. *Write-set arrived* refers to the write-set of the transaction whose operation currently holds a lock on the data item.
A read lock is a shared lock, it can be granted to multiple read operations simultaneously. On the other hand, write locks are exclusive locks. There can only be one write lock at a time on a data item.

	Read Lock			Write Lock
Request for a	Done		Not Done	
	Write-set arrived	!(Write-set arrived)		
Read Lock	Y	Y	Y	N
Write Lock	N	Y	N	N

Table 6.1: Which lock can be granted at which moment

When a read lock is requested for a data item x that holds already a shared lock, it can be granted. If this data item holds an exclusive lock, however, the lock request has to wait. The situation is more complicated when an exclusive lock is requested by operation $w_j[x]$. If the corresponding data item x already holds an exclusive lock, the new lock request has to wait. Otherwise suppose that a read operation $r_i[x](i \neq j)$ holds a shared lock on x. It depends now on the status of the transaction T_i whether the lock can be granted. We distinguish two cases:

- The write-set of the transaction T_i has arrived before the write-set of T_j. Therefore all the conflicting write operations of T_i will be processed before the ones of T_j and T_i will be before T_j in the serialization order. It is therefore not necessary to abort T_i. As soon as all the operations of T_i are processed and have released their locks, $w_j[x]$ will be granted the exclusive lock on x (provided it was not aborted in the meantime and it is the next operation waiting for the lock).
- The write-set of T_i is still pending. In this case no serial order of the corresponding transactions exists and T_i has to be aborted. Since no write operations of T_i are concerned, the lock manager does not have to undo updates to the database. It has to wait for the data

manager to finish processing of $r_i[x]$ though. Provided that T_j has not been aborted in the meantime, $w_j[x]$ gets the exclusive lock on x as soon as the shared lock has been released and if it is the next operation waiting for the lock.

An improvement of the lock handling can be achieved when waiting write locks overtake waiting read locks. Suppose that an exclusive lock is hold on data item x and that read operation $r_i[x]$ is waiting. When the write operation $w_j[x]$ tries to get an exclusive lock on x, it also has to wait. When the current exclusive lock on x is released, $r_i[x]$ gets its shared lock. However, since a write operation is also waiting for the lock, transaction T_i gets aborted. This is clearly not a satisfactory situation. In order to avoid T_i's abort, we allow that $w_j[x]$ overtakes $r_i[x]$. Then the write operation is executed before the read operation and there is no need for aborting T_i (at least not in this isolated view). The disadvantage of this approach is, however, that read operations can be delayed for a very long time.

6.7.4 Lock Manager

The entire concurrency control mechanism is controlled by the lock manager. We will describe the behavior of the lock manager by the actions that take place when communicating with other components.

Synchronization With The Data Managers

For the purpose of this discussion we abstract from the different data manager processes and consider them as one logical object, called data manager.

Every time a processed operation is received from the DM, the lock manager has to check first whether the transaction has been aborted in the meantime. If this is the case write operations have to be undone while read operations simply release their locks.

A flow chart of the actions to be taken when an operation commit is received from a data manager process are depicted in figure 6.7.

If the transaction T has not been aborted yet, then the operation is marked as done. If its write-set has already arrived, we check if all the operations of the current transaction are done now. In the other case the operation, since it has to be a read, is committed to the transaction manager. When all the operations are done, the lock manager has to distinguish locally initiated transactions from remote ones. The former will namely be committed to the TM, while the latter still might have to wait for their commit to be broadcast by their initiating nodes. However, if it already arrived, the locks of T are released and T is removed from the transaction list. Committed transactions delete all their locks and are removed from the transaction handler.
If T has been aborted in the meantime, the lock hold by the operation is released. For write operations, the updates are undone.

The lock manager has the control over the execution of the operations. It keeps track of all currently executable operations. Every time a data manager potentially gets available to process an operation, it is handed over the next executable operation. In particular, this is the case when the lock manager gets back a processed operation from one of the data manager processes. Also when the lock manager gets a write-set or a single read operation from another process, it tries to execute the next operation. Only if a data manager is available and some operation is executable, it will succeed.

Synchronization With The Interface Adaptor

The actions on receiving a request from the interface adaptor are shown in figure 6.8.

When the write-set of transaction T is received, the LM first checks, whether the transaction T is already in the transaction list. If it is not, the write locks for T are atomically requested. Then all transactions, whose operations conflict with T's operations, are aborted and T is stored in the transaction list.
On the other hand, if T is already in the transaction list, the LM checks

whether T has been aborted in the meantime. If the transaction has been aborted, the write-set is ignored. Otherwise the write locks for T are requested, the transactions of conflicting operations aborted and T stored in the transaction list.

Finally the lock manager executes the next operation, if one is executable and a data manager process is available.

Synchronization With The Transaction Manager

The lock manager receives read requests of transaction T from the transaction manager. It requests a shared lock for the read operation and adds a new entry for T to the transaction list. Then, if a data manager process is available and an operation is executable, this operation is processed.

Coordinating Interactions

Whereas the previous subsubsections discussed each interaction with another components separately, we focus now on the coordination of these interactions.

The lock manager schedules requests of other components by priority. Highest priority have messages from the interface adaptor. Therefore, write-sets will request their locks before read operations. This again avoids potential conflicts. In addition the lock manager can react quickly on an eventual abort message, before wasting any more resources onto processing the operations of the aborted transaction. The second highest priority have the read requests. The lowest priority is assigned to events indicating the arrival of a processed operation from a DM processes.
Whether another scheduling would be better in terms of performance has not been tested.

6.8 Data Manager

This section first explains the conception of a data manager process. In a second subsection we discuss the simulation parameters selected in order to model the behavior of real data managers.

6.8.1 Behavior of a Data Manager

The data manager (DM) consists of several processes, each having access to the entire database. Every data manager accepts requests to execute operations from the lock manager. Depending on whether it is a read or a write operations, the appropriate actions are taken. Write operations access the corresponding data item and assign some random value to it. Read operations get the value of their data item. After having processed an operation, the data manager returns the response to the lock manager.

6.8.2 Simulation Parameters

A data manager process contains three simulation parameters: the *disk access time*, the *CPU time* and the *disk access rate*. These three parameters will be explained in the following.

Disk Access Time

In order to compute the disk access time, we need the following parameters [ElN94]:

1. **Seek time (T_{seek})**: Specifies the time needed to position the read/write head on the correct track of the disk. We assume in this context that our database is stored on movable-head disks. This time varies according to the distance between the current and the new track on the disk. The typical range of average seek time is 10 to 60 ms.
2. **Rotational delay (T_{rot})**: This parameter specifies the time needed for the block on the track to rotate under the write head. This depends on the rotational speed of the disk as well as on the length

of the track. Elmasri/Navathe [ElN94] propose a value of $T_{rot} = 8.33$ msec.

3. **Block transfer time (T_{bt})**: Time to read a block of data from the disk. This parameter depends on the block size and the rotational speed.

According to [ElN94] the disk access time is computed in the following way:

$T_{da} = T_{seek} + T_{rot} + T_{bt}$

Obviously this values just provide an approximation of the real situation. However they are sufficient to get a general impression on the cost of disk accesses in our database system.

CPU Time

This parameter specifies the time needed to access the data item in main memory and to execute the update or the read operation.

Disk Access Rate

The current version of the RDBS simulation tool does not simulate the interaction between main memory and disk. Since in a real DBS a data item does not have to be fetched from the disk if it is already in main memory, not every access to it incurs the cost of a disk access. The rate at which the disk has to be accessed is called disk access rate. This parameter allows to model the interaction between the main memory and the disk in a real DBS to some extent. The data manager process accesses the disk at the rate specified with this parameter.

6.9 Physical Database

The databases on each node are simulated as arrays of integers. Since we just want to model the accesses on databases, this is sufficient for our

purposes.

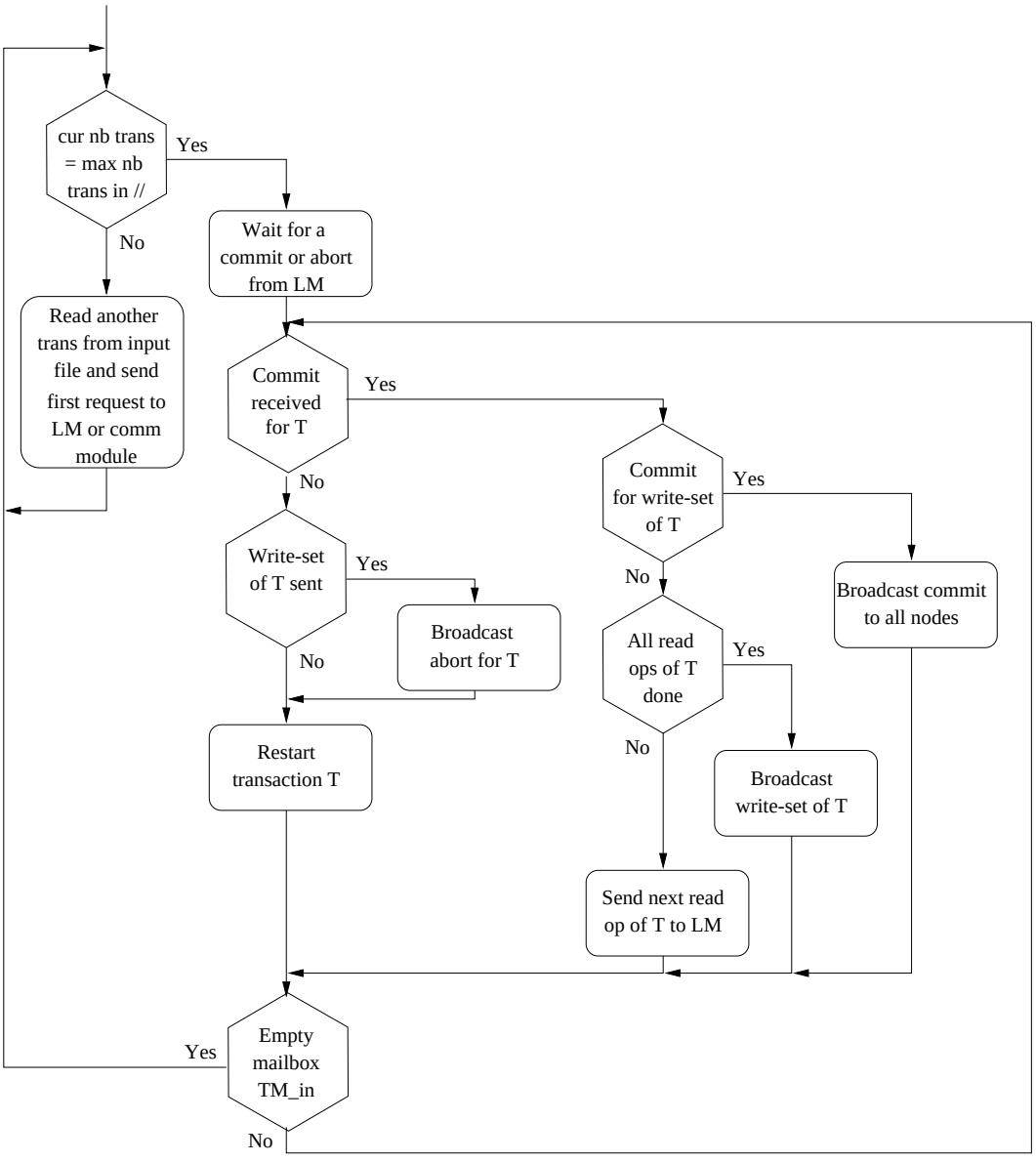

Figure 6.4: Flow chart of activities of the transaction manager

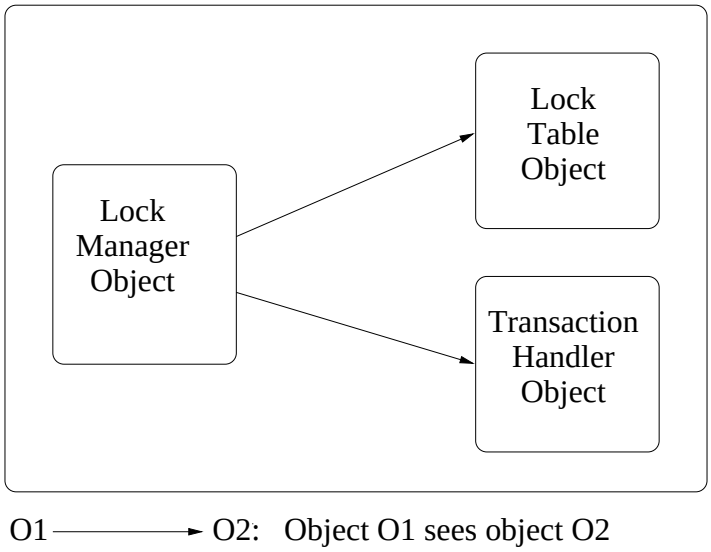

Figure 6.5: Objects defined in the lock manager

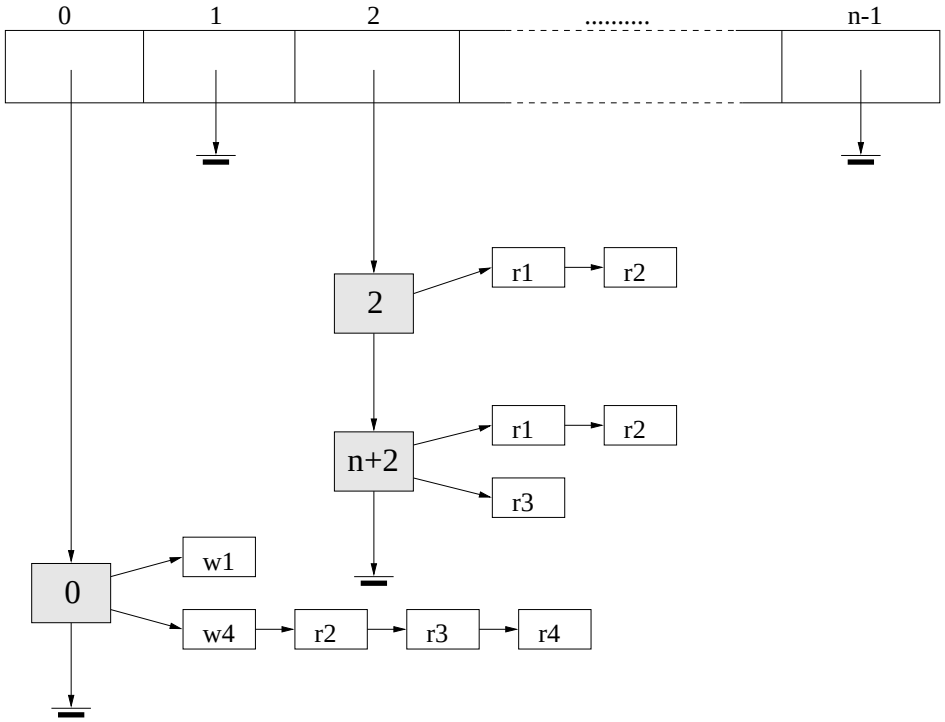

Figure 6.6: Lock Table Structure

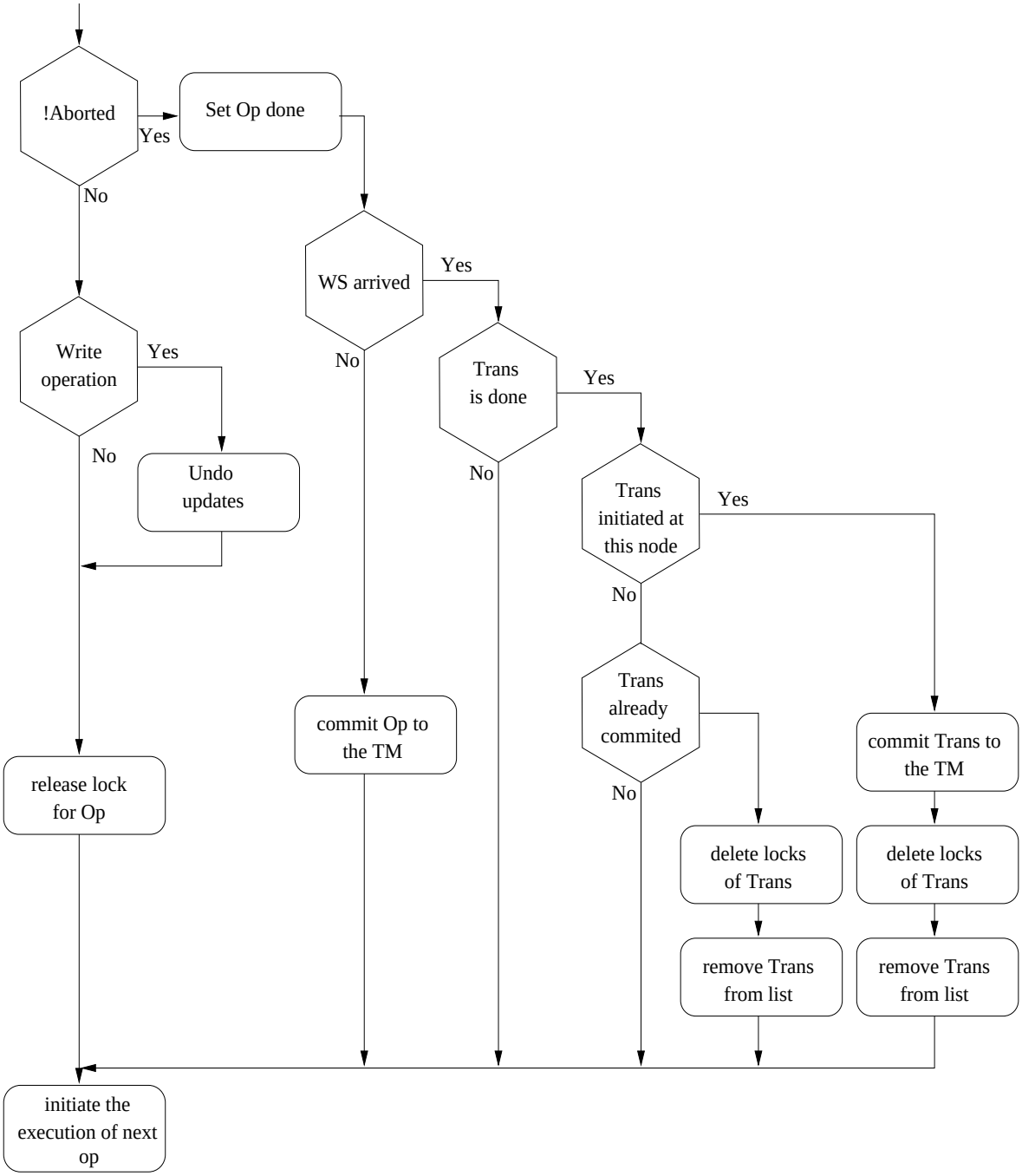

Figure 6.7: Reception of an operation commit from a DM process

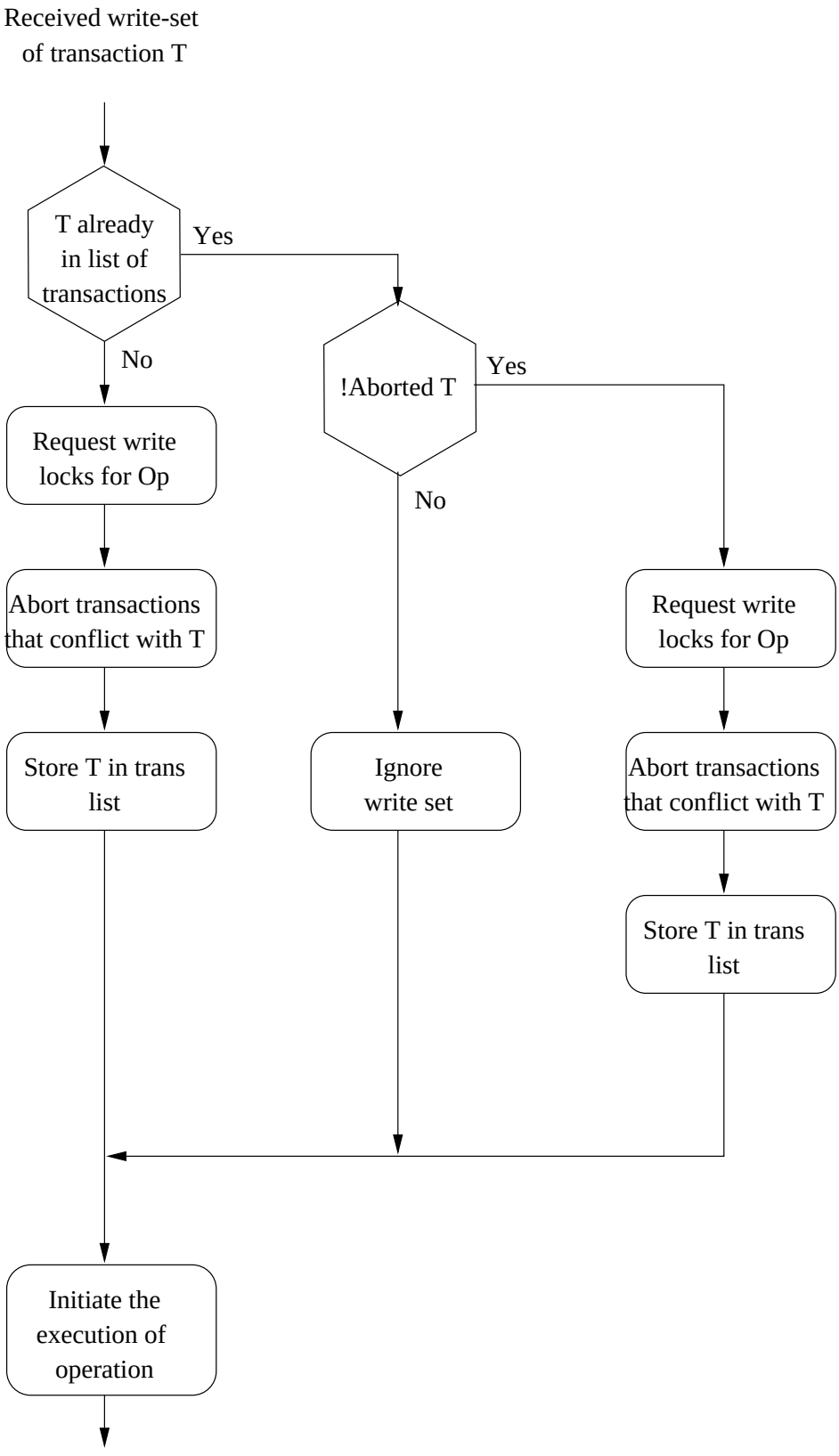

Figure 6.8: Reception of a write-set of a transaction T

Chapter 7

Performance Measurements

The performance measurements in this chapter give some indications about the behavior of a real RDBS.

The simulator has been tested when working in normal conditions. This means that node failures are not simulated.

We perform the testing in two steps: First we only measure the performance of the atomic broadcast primitive (section 7.1). In this case it is necessary to add additional features to the system in order to retrieve the corresponding measurements. Then we try to analyze the influence of the atomic broadcast on the performance of the RDBS and vice-versa. The perception of eventual bottlenecks in this system is also a goal in this second step.

7.1 Atomic Broadcast

We first build an environment which allows to measure the performance of the communication module, i.e. of the atomic broadcast algorithm. Then this section presents the values that are assigned to the different simulation parameters. These values can be specified in an input file. Subsection 7.1.3 finally illustrates the results of the atomic broadcast's simulation and gives a brief discussion of these results.
We will refer to the atomic broadcast simply as broadcast in the following.

7.1.1 Additional Features for Testing

Since we want to measure the performance of the broadcast isolated from the other components of the RDBS, some additional features have to be added to the system. In particular, a sending and a receiving process on every node, called *sender* and *receiver*, now interface with the communication module. The sender invokes the broadcasts at a certain rate, whereas the receiver receives the messages and discards them. We call this broadcast simulator simply simulator, since it can not be mixed up with the RDBS simulator in this section. Figure 7.1 shows the architecture of this testing environment.

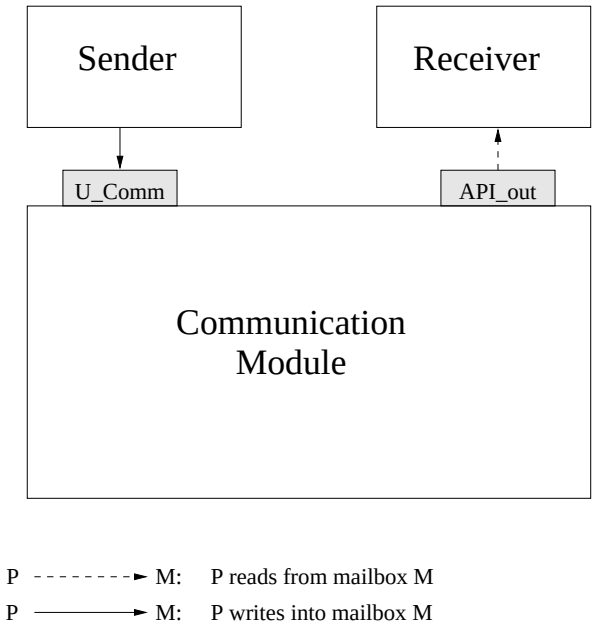

Figure 7.1: Architecture of the testing environment

Since the senders do not all start at the same time, they generally do not execute their statements simultaneously.

7.1.2 Simulation Parameters

From chapter 6 we know that the atomic broadcast simulation tool consumes simulated time at 3 places:

- sim_Message_Delivery_Time: message delivery time, which is the time needed by the message to travel from the sender node n_i to the

receiving node n_j. It is set to 0.1ms.

- sim_Send_Preparation_Time: CPU time used by the communication layer for packing and sending a message m and for executing the necessary flow control. Its value is 0.2ms.
- sim_Receive_Preparation_Time: CPU time needed to retrieve m from the network, compute the checksum and provide flow control. For our simulation measurements it is set to the same value as sim_Send_Preparation_Tim

The values for sim_Network_Delivery_Delay, sim_Send_Preparation_Time and sim_Receive_Prepara- tion_Time have been taken from [Str95] and seem reasonable in our context.

The rate with which the sender sends messages can be specified by using two parameters: sim_Msg_Delay and sim_Delay_Var. The sender of every node broadcasts a message every t ms, where t is uniformally distributed in the interval [sim_Msg_Delay - Δt, sim_Msg_Delay + Δt]. Δt is the sim_Msg_Delay multiplied with sim_Delay_Var. We will call the average size of the interval 0 to t, during which one broadcast is performed *message sending delay.*

For the following tests, the other simulation parameters have the following values:

sim_Number_Of_Nodes	variable
sim_Simulation_Time	50000.0
sim_Timeout_Value	variable
sim_Network_Delivery_Delay	0.1
sim_Send_Preparation_Time	0.2
sim_Receive_Preparation_Time	0.2
sim_Msg_Delay	variable
sim_Delay_Var	0.1

Table 7.1: Simulation parameter values for the atomic broadcast

A value called *variable* specifies that these parameters have been changed during the tests. Especially it might be necessary to adapt the timeout value in the consensus (sim_Timeout_Value), depending on the number of nodes in the system and the message sending delay.

We simulate the behavior of the system during 50000ms. Measurements are only extracted after a certain time (500ms), allowing the system to pass from the startup phase into normal working performance. This prevents that the startup phase of the simulation influences the measurements.

7.1.3 Discussion of the Simulation Results

The measurements presented are generally averages from sets of 5 simulation runs. It was not possible to run more tests because of time constraints. However, our results still allow to derive some conclusions.

Tests have shown that the broadcast algorithm runs only for a limited number of nodes. If the system contains too many nodes and the broadcast rate is too high, the system does not deliver any messages at all. This is due to message contention in the network. We therefore had to adapt the simulation domain to a number of nodes between 2 and 15 in order to get reasonable results. In addition the message sending delay also had to be selected in order to obtain a reasonable behavior of the system. It varies in the interval from 100ms to 5000ms.

Figure 7.2 illustrates the results of a first set of simulation results respecting these constraints. It represents the *message response time* of the atomic broadcast primitive depending on the message sending delay. The response time of a message m thereby specifies the time from broadcasting m until m's delivery to the layer on top of the communication module. The values displayed in the graphs are average values.

Note that no messages are delivered in a system with 10 or 15 nodes at a message sending delay of 100ms. The broadcast algorithm is not able to handle the broadcasts of systems with such a number of nodes. This is the reason why the corresponding graphs start at a message sending delay of 500ms.

The graph for 15 nodes shows the general behavior of the simulator in terms of message response time. For a low message sending delay, the

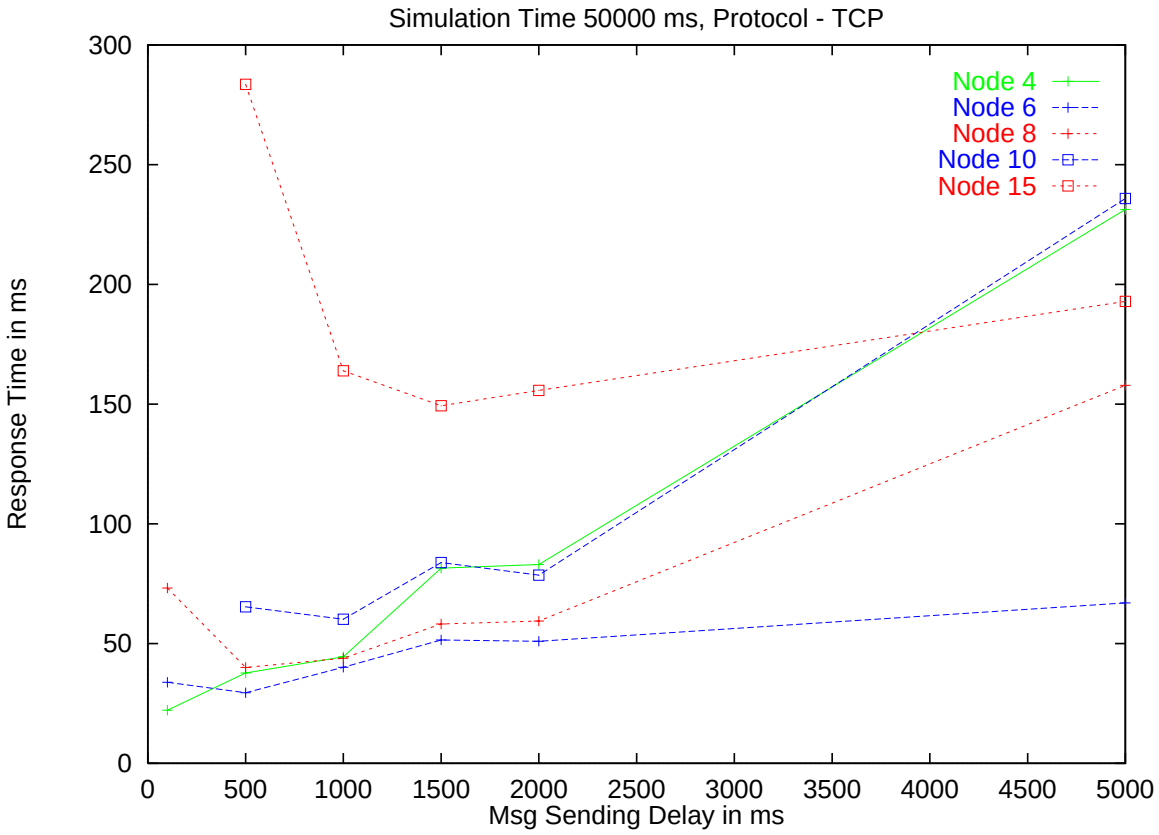

Figure 7.2: Average response time of atomic broadcast on TCP

response time is high. As already mentioned, this is due to network contention. On the other hand the response time also increases for message sending delays greater than a certain threshold. This time, however, there is a different reason to this behavior. Since two broadcasts executed on the same node are separated by a considerable delay, the consensus occurs over a small set of messages. This leads to a higher average response time. Table 7.2 allows to verify that the number of messages delivered per decision made by the consensus actually decreases with increasing message sending delay. The average number of messages delivered together can be retrieved from the 9^{th} column.

This table further displays the utilization of the network and the utilization of the CPU for sending and receiving messages. The 6^{th} column shows the average utilization of this CPU while the next column indicates the utilization on node 1. Note that the CPU utilization on node 1 is slightly higher. We will explain this behavior in the following subsection. The final two columns in the table show the average number of

messages sent per node and the average number of messages a-delivered per node.
The same behavior as for the system with 15 nodes is also illustrated by the graphs for systems of 6,8 and 10 nodes. A system with 4 nodes would probably behave in the same way for message sending delays less than 100ms. However, these tests have not been performed yet. The result tables for these configurations are depicted in appendix A.
It is reasonable to suspect that the broadcast protocol used in this implementation has a certain bandwidth in message sending delays, where it runs in an optimal configuration. Outside this bandwidth, its performance decreases considerably.

Msg send delay [ms]	Response t min [ms]	Response t max [ms]	Response t mean [ms]	Net utilization	CPU utilization	CPU_1 utilization	Nb msgs deliv max	Nb msgs deliv mean	Nb msgs sent	Nb msgs rcvd
500	26.3	760.6	283.6	0.9738	0.255	0.2828	14.8	4.6108	1505.2	1497
1000	18.7	934.9	163.9	0.756	0.1964	0.2382	15	1.639	757.2	755.8
1500	19.1	1139.9	149.3	0.5366	0.1396	0.1724	12	1.4612	508	507.2
2000	17.8	1548.9	155.8	0.4216	0.1098	0.1356	11.4	1.3298	380.6	380.4
5000	20.0	3343.6	192.9	0.1906	0.049	0.0616	6.4	1.1294	156.8	156

Table 7.2: Performance Measurements for 15 Nodes on TCP

We now consider the network load for the same set of simulations. The results are depicted in figure 7.3. As expected, the net load decreases with increasing message sending delay for a particular system. In addition, the smaller the number of nodes in the system is, the less the network is used. Note the degenerate case where the network load is equal to one. In this situation, the network is used all the time.
It has to be the goal of a RDBS designer to let the system run in its optimal bandwidth.

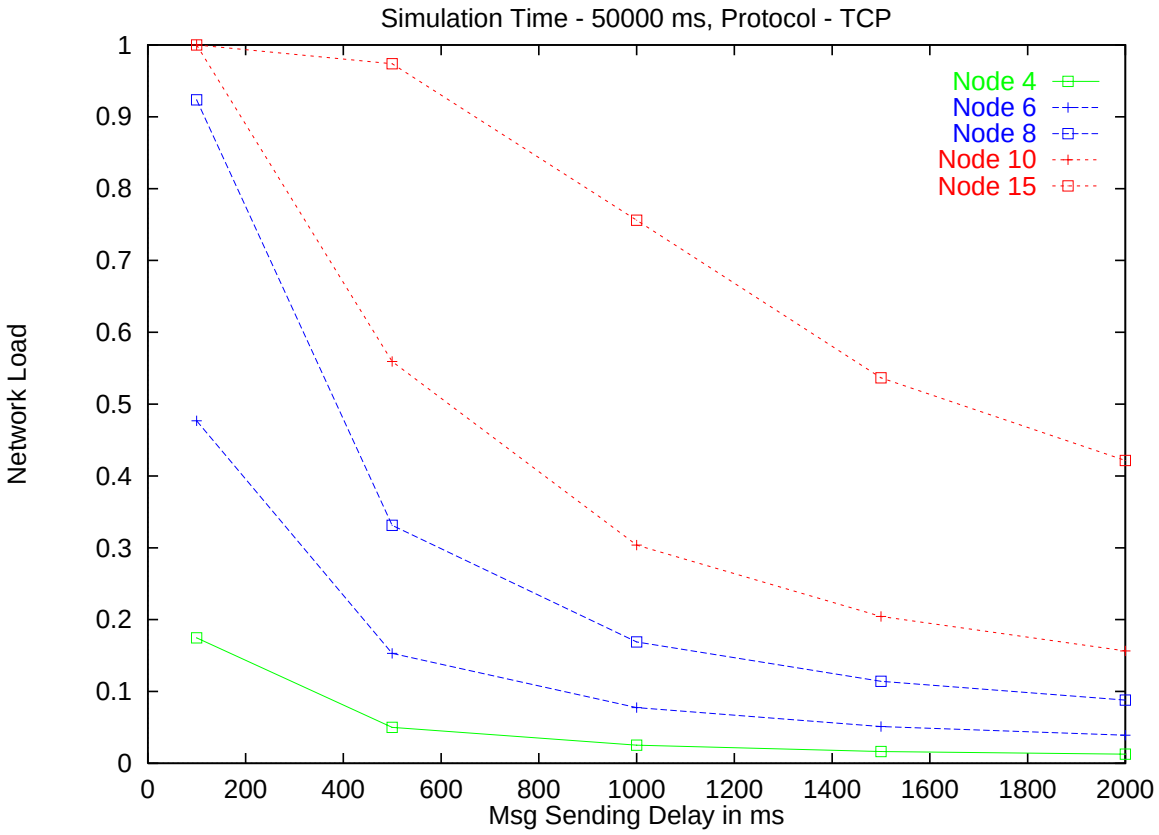

Figure 7.3: Average buffer length in the network using TCP

7.1.4 Stability of the Atomic Broadcast

While performing the simulations we noted that for certain configurations the system performance degenerates. This happens especially for message sending delays less than 500ms.
As an example we represent a system with 6 nodes (figure 7.4). On the x-axis we represent the elapsed simulation time, while the y-axis shows the total number of consensi performed on one node up to this point. In order to elucidate the results, the consensus time-out has been set to 20000ms. In one simulation run, node 1 is selected as initial coordinator every time a consensus is started (according to the atomic broadcast algorithm in figure 3.3. In the other case, the initial coordinator changes every time the consensus is started. Since every node in the system still needs to know the initial coordinator, the coordinator is selected as the node, whose ID is equal to the consensus number modulo the number of nodes.

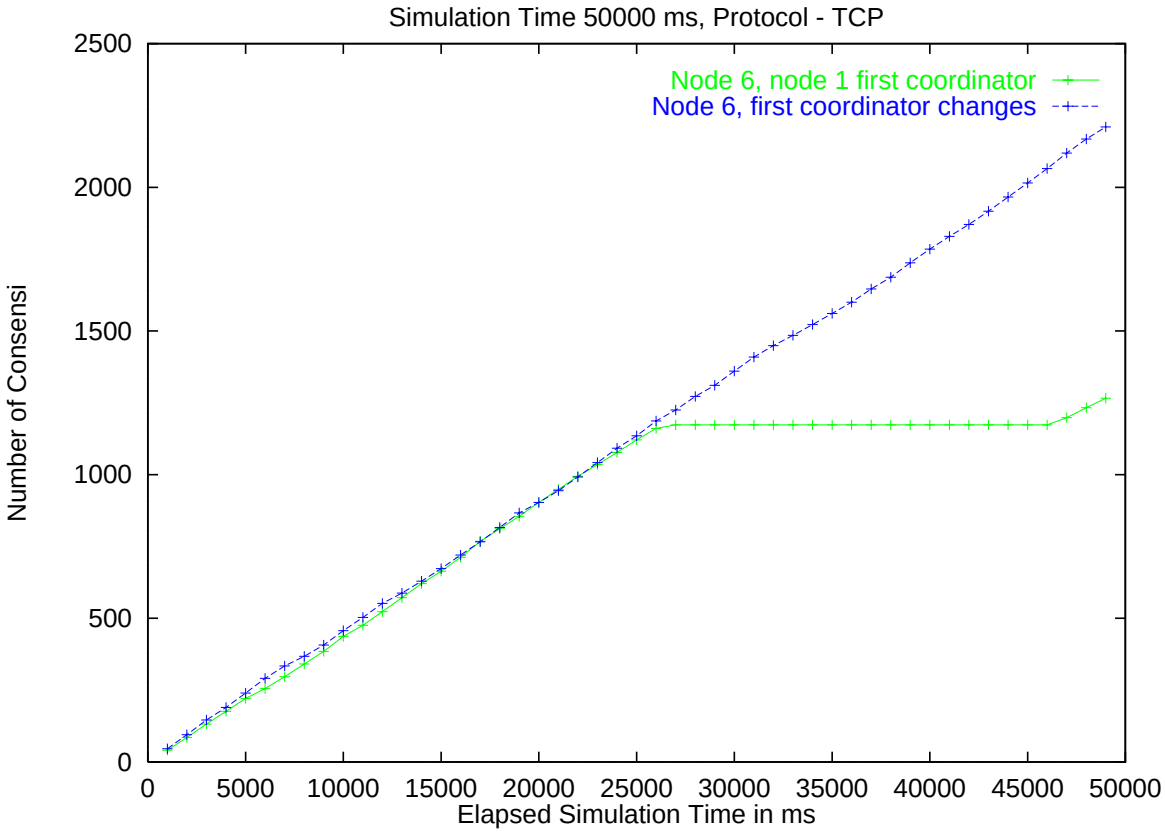

Figure 7.4: Different initial coordinators for the consensus on TCP

The resulting graphs show that for the first case the consensus does not succeed to reach a decision between 25000ms and 45000ms. After this period the other nodes suspect node 1 and a new coordinator is selected. This coordinator (node 2) then succeeds to reach a decision. However, during this time no message has been delivered, which leaves the system in a very poor performance with respect to average message response time. One reason why node 1 could not reach a decision might be that it simply got too many messages. Since in the consensus all messages are sent to the coordinator or from the coordinator to all nodes, the corresponding node might get overwhelmed with messages. When we inspect the tables with the performance measurements we indeed note that the CPU utilization at node 1 is higher than the average utilization.
On the other hand the second graph is more regular. Rotating the initial coordinator seems prevent one node from getting too many messages. The performance in this system is therefore better. However further tests

have shown that even a rotation of the initial coordinator for the consensus does not prevent the system from degenerating in some particular cases.

Atomic Broadcast on UDP

At this point it might be interesting to compare the performance of the atomic broadcast using TCP with the same broadcast algorithm, but using UDP communication instead. Even though the atomic broadcast might not be correct any more in a theoretical context, we get an indication of the cost of reliable communication channels.
There are some differences between communication on TCP and communicating using UDP. In particular, UDP provides a broadcast primitive, which allows to broadcast a message to all nodes at the same cost as a point-to-point message to the node which is furthermost from the sending node.

We have performed the same simulation for a system with 15 nodes as for the atomic broadcast running on TCP. Figure 7.5 shows the results.

We believe that the UDP broadcast response time graph has the same general shape as the TCP broadcast graphs. They are simply shifted towards the origin. An indication for this assumption is given by the graph of the system with 15 nodes. It reaches its optimal configuration at a message sending delay of about 500ms. The response time at this point is on the average about 70ms. In contrary, the broadcast on TCP has an average message response time for this configuration of about 280ms. This is 4 times higher. Therefore the second system can handle broadcasts which arrive in small intervals more efficiently then a system built on TCP.

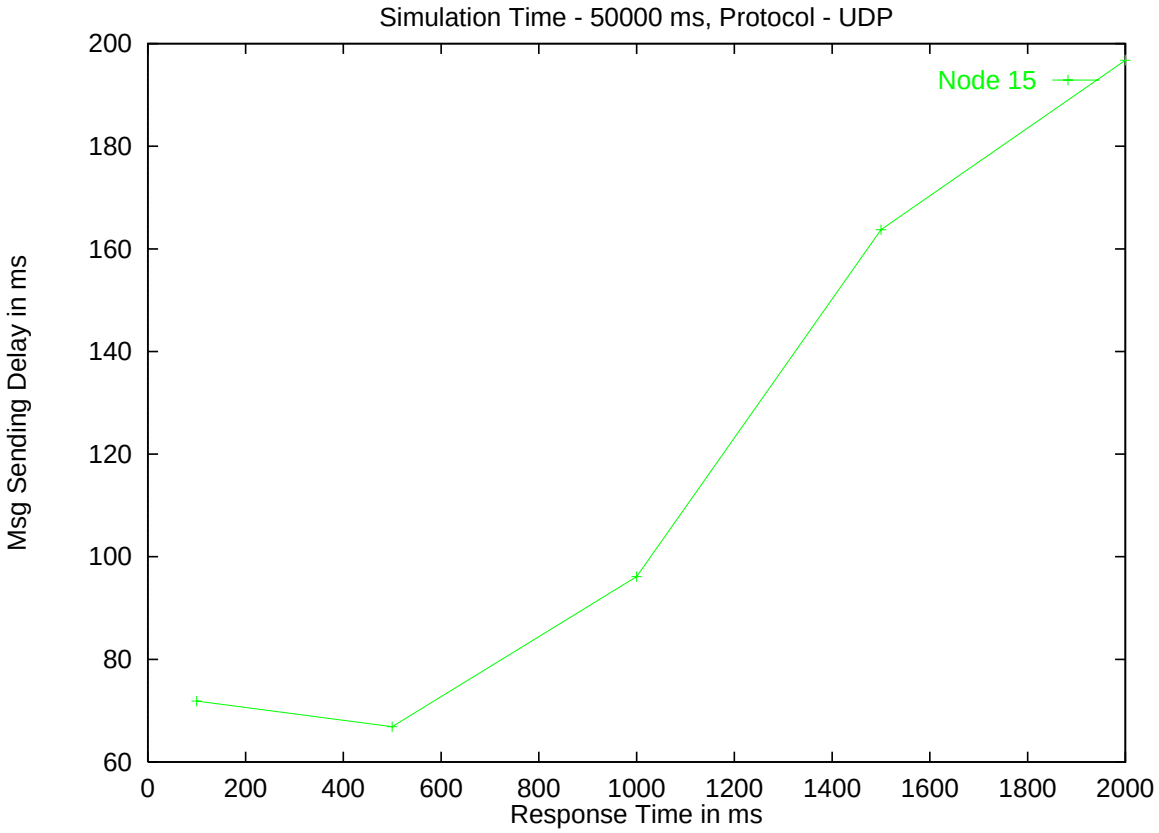

Figure 7.5: Average response time of atomic broadcast on UDP

7.2 Replicated Database System

Having measured the performance of the communication module, we would like to asses the impact of the communication module to the whole RDBS system. We try to illustrate the behavior of the RDBS simulation tool, referred to as simulator in the following, and give indications about its performance in some general cases.

As in the previous section we start with a presentation of the simulation parameters and their values and finally discuss the simulation results.

7.2.1 Simulation Parameters

In addition to the simulation parameters already discussed in subsection 7.1.2, which let simulation time pass, the RDBS simulator allows to model elapsed time with the following parameters:

- sim_Data_Access: average access time to the disk (section 6.8). A value of 40ms [ElN94] seemed reasonable for our purpose.
- sim_DM_Operation_Time: the time needed to perform an operation in main memory including the lock overhead. It has been set to 0.7ms.

Table 7.3 defines the values that have been taken for the different simulation parameters. The purpose of the first 8 parameters has already been explained before. The number of nodes in the system has been fixed to 10 for the following performance measurements. sim_Max_Size_Database allows to specify the size of a physical database. It is kept small to be able to analyze lock contention. There exists as many databases in the RDBS as data managers (DM). The number of DM processes is defined by sim_Number_DM_Processes. It has been selected such that the system has a reasonable disk utilization rate. In a real system, the disk utilization can be decreased by simply adding more disks to the DBS. It is basically a question of cost. In addition the RDBS simulation tool allows to set the size of the lock table in the lock manager (sim_Lock_Table_Size).

Finally, the user can also specify the number of operations in a transaction (sim_Number_Ops_Per_Transaction) and the number of transactions processed in parallel by each transaction manager (sim_Nb_Trans_Proc).

sim_Number_Of_Nodes	5
sim_Simulation_Time	200000.0
sim_Timeout_Value	variable
sim_Network_Delivery_Delay	0.1
sim_Send_Preparation_Time	0.2
sim_Receive_Preparation_Time	0.2
sim_Data_Access	40
sim_DM_Operation_Time	0.7
sim_Max_Size_Database	2000
sim_Number_DM_Processes	5
sim_Lock_Table_Size	200
sim_Number_Ops_Per_Transaction	12
sim_Nb_Trans_Proc	variable

Table 7.3: Simulation parameter values for the RDBS simulation tool

In addition to these parameters, the disk access rate has been set to 20%. This means that every fifth time a data manager has to access the disk, thereby incuring the additional cost of sim_Data_Accessms.

7.2.2 Simulation Results

The results presented in this subsection give a first impression of the behavior and the performance of the RDBS simulation tool. We realize, however, that the set of tests performed is not exhaustive at all and that there still exists interesting cases which need to be simulated.

Table 7.5 gives an overview of the results of our simulations of a system with 5 nodes. An explanation of the simulation parameters is given in table 7.4. We made several simulation runs with different values for the number of transactions processed in parallel by the transaction manager of one node. A higher value for '*Nb trans in*' implies that more transactions are processed in parallel in the RDBS. In particular, by increasing this parameter by 5, the total number of transactions in the system at any moment increases by (5*N). This leads to more conflicts between operations of different transactions, which increases on one hand the number

Nb trans in //	Number of transactions per node, processed in parallel by the transaction manager
Resp t ABcast	Time needed by a message from the A-broadcast to its A-delivery
Net utilization	Utilization of the network. A network used all the time has a value of 1 in this place
CPU utilization	Utilization of the communication coprocessor
Resp t DB	Response time of a transaction. This is the time from starting the transaction until it commits on the initiating node, including restarts due to aborts.
DM CPU utilization	Utilization of the CPU in a DM process.
Disk utilization	Utilization of the disk attached to a DM process.
T waited for lock	Time an operation waits after requesting a lock until it gets this lock granted.
T waited for exec	Time an operation waits after requesting a lock until it gets actually executed by a DM process.
Nb aborts per trans	Number of times a transaction gets aborted.
Avg nb trans aborted	Average number of transactions per node that got aborted.
Avg nb trans proc	Average number of transactions per node processed.
Throughput	Number of transactions processed per second.

Table 7.4: Explanation of simulated parameters

of aborted transactions, and on the other hand the time an operation has to wait for a lock. Both consequences increase the transaction response time. This behavior can be verified in table 7.5.

By inspecting the results for 2, 5, 10 and 15 transaction processed in parallel on each node we can note that the network utilization increases, as well as the data manager's CPU and the disk utilization. The average message response time of the atomic broadcast also increases. However, the net utilization remains stable when the number of transactions processed in parallel is increased from 20 to 25 and the average message response time even decreases. In contrary, the utilization of the DMs' CPUs and the disk are increasing. Since the number of aborted transactions is increasing, a lot of transactions are aborted while their read operations are executed. This leads to the conclusion that in this case the serialization protocol becomes the bottleneck. The operations have to wait a longer time for their locks and also for an available DM process. However, the reason for this bottleneck could be the high response times

of the atomic broadcast. When read locks have to be hold for a long time while waiting for the write-set to arrive, the probability of conflicts with other transactions increases.

Another scenario is shown in table 7.6. It illustrates the results of the same measurements in a system with 10 nodes. We note that this time the network is the bottleneck. Its utilization is almost 1, which means that it is sending messages all the time. On the other hand, the utilization of the data managers CPUs and the disks is low and remains stable for 10,15 and 20 transactions processed in parallel on each node. These resources could handle more requests.

Figure 7.6 illustrates the response times of a system with 5 nodes for 5 and 10 transactions processed in parallel on each node. It displays the average *transaction response time* as well as the average message response time. The transaction response time is the time that has elapsed between starting a transaction and committing it on the initiating node, including restarts due to aborts.
We first consider only the two graphs for 5 transactions processed in parallel. Up to 20000ms the system is in its startup phase. After 20000ms it remains stable with respect to its response times. The average response time for the atomic broadcast is 50ms, whereas a transaction needs on an average about 300ms until it commits. The additional overhead for transactions comes from the fact, that all read operations are first executed sequentially. Even though they are processed locally, they still might get blocked on a data item holding already an exclusive lock. In addition, transactions might get aborted, which again increases the cost considerably.
For a system processing 10 transactions in parallel on each node, the average message response times as well as the transaction response time increase. However, the transaction response time increases to a greater extent than the time needed for an atomic broadcast. Since there are more operations in the system, the conflict rate in the lock manager increases. This leads to operations getting delayed longer as well as a higher number of transaction aborts. However, longer operation delays rather decrease network traffic, since write-sets are sent later. On the

other hand the number of broadcasts increases with an increasing number of transaction aborts. This could lead to higher response times for atomic broadcast messages. By inspecting figure 7.6 we suppose that the impact of an increased operation delay on the system is greater than the the influence of the aborts of transactions.

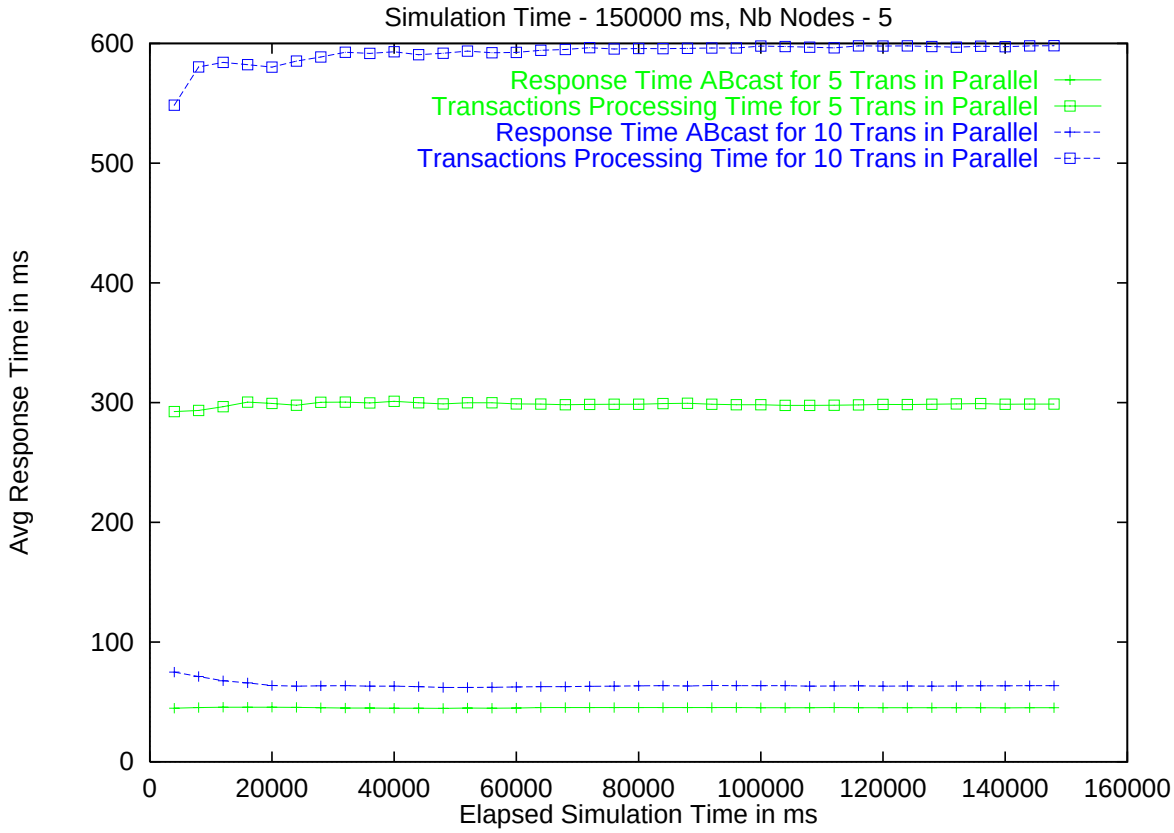

Figure 7.6: Average response time measured periodically during the simulation for a system with 5 nodes

Nb trans in //	2	5	10	15	20	25
Avg nb trans proc	1415	7676	7679	7673	7629	7667
Avg nb trans aborted	115	306	515	604	792	866
Nb aborts per trans mean	0.058	0.145	0.297	0.476	0.613	0.709
Nb aborts per trans max	3	6	11	14	19	23
T waited for exec mean [ms]	9	40	84	114	137	151
T waited for exec max [ms]	344	531	1228	1349	1611	2019
T waited for lock mean [ms]	1	2	6	9	13	14
T waited for lock max [ms]	291	484	995	1349	1592	1752
Disk utilization	0.696	0.837	0.859	0.860	0.870	0.894
DM CPU utilization	0.061	0.073	0.075	0.076	0.077	0.078
Resp t DB mean [ms]	141	298	595	914	1228	1496
Resp t DB max [ms]	868	2463	8360	10749	17277	22213
Resp t DB min [ms]	18	26	48	113	92	124
CPU utilization	0.602	0.624	0.627	0.632	0.634	0.634
Net utilization	0.787	0.811	0.812	0.815	0.817	0.817
Resp t ABcast mean [ms]	32	44	62	84	95	85
Resp t ABcast max [ms]	92	134	210	285	360	388
Resp t ABcast min [ms]	4	3	3	4	4	3

Table 7.5: Simulation results for the RDBS simulation tool for a simulation time of 100000ms in a system with 5 nodes

Nb trans in //	2	5	10	15	20
Resp t ABcast min [ms]	17	28	11	11	7
Resp t ABcast max [ms]	505	839	1393	1457	1648
Resp t ABcast mean [ms]	258	458	663	808	951
Net utilization	0.993	0.996	0.995	0.991	0.983
CPU utilization	0.392	0.395	0.397	0.395	0.392
Resp t DB min [ms]	65	238	315	428	426
Resp t DB max [ms]	3925	15147	44158	75699	77898
Resp t DB mean [ms]	542	1381	3035	4845	6913
DM CPU utilization	0.033	0.037	0.038	0.038	0.038
Disk utilization	0.381	0.424	0.436	0.434	0.435
T waited for lock max [ms]	1280	3027	5262	9943	12778
T waited for lock mean [ms]	6	24	64	108	156
T waited for exec max [ms]	1286	3189	6373	10736	12778
T waited for exec mean [ms]	39	115	224	317	406
Nb aborts per trans max	6	14	23	29	26
Nb aborts per trans mean	0.139	0.345	0.670	0.914	1.194
Avg nb trans aborted	53	100	132	150	155

Table 7.6: Simulation results for the RDBS simulation tool running for 100000ms in a system with 10 nodes

Chapter 8

Implementation of the Replicated Database System Simulator

This chapter looks at some of the implementation issues in this project. Since the code is fairly well documented, the discussion is rather general. Even though not all the coding details are explained, the most important implementation features will be highlighted and discussed. For all the details refer to the code given in appendix C.
We start with some general issues of the implementation. One section is then devoted to the implementation of every component of the RDBS simulation tool. Finally, section 8.9 explains how the system is set up and section 8.10 indicates the limitations of the simulation tool.

8.1 General Issues

As already mentioned in chapter 6 the replicated database system simulation tool has been implemented using an object-oriented approach. Every component in the system is defined as an object with a well-specified interface.

Figure 8.1 illustrates the file dependencies in this RDBS simulation tool. In general every component is implemented in two files. One of them contains the interface, while the others consists of the implementation of the primitives defined in the interface.

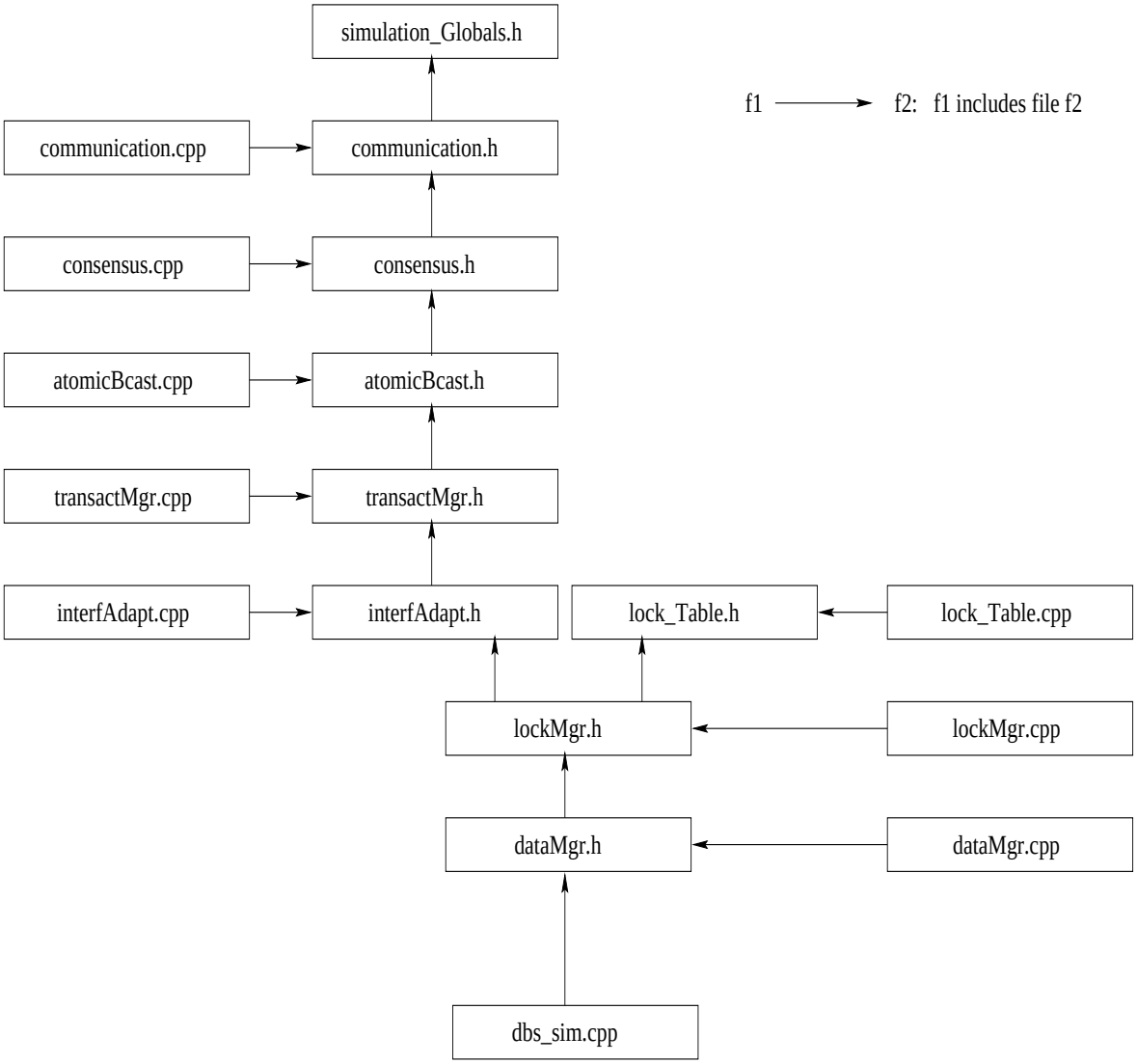

Figure 8.1: File dependencies of the RDBS simulation tool

Due to the limited duration of this project, the termination problem of the database simulator has not been addressed yet. This does not prove to be a problem though. Since simulations are usually performed during a certain period of time when the system is running in normal conditions, the termination protocol does not influence the results. In contrary it could even falsify the simulation results and lead to incorrect performance measures. Therefore the simulator is stopped when the simulation time has elapsed without terminating properly every CSIM process running. They are terminated when the main program stops.

8.2 Network

One process simulates the network. It interfaces with the other parts of the system by means of mailboxes. In particular the N nodes of the RDBS exchange messages through $N + 1$ mailboxes. There is one input mailbox for all RDBS nodes, called *network input mailbox* (Net_MB). It is the same for all nodes and is used by the nodes to send a message. The communication from the network to the different nodes is also managed by mailboxes. Every node n_i has its own receiving mailbox, called L_Comm, where the network drops all messages sent to n_i. This situation is illustrated by figure 8.2.

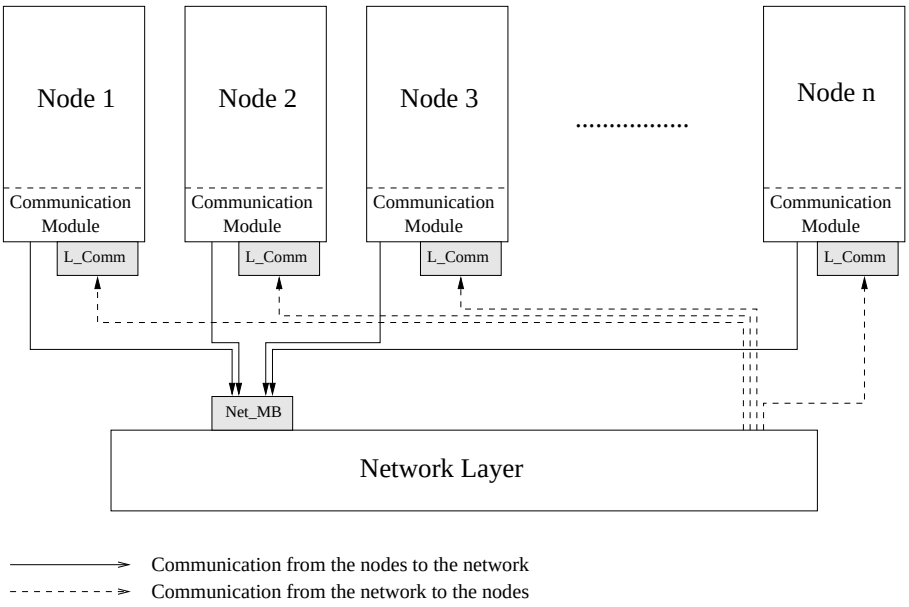

Figure 8.2: Integration of the Network Layer

When a node n_i wants to send a message m to node n_j, n_i puts m into the mailbox Net_MB (We suppose in this context that a node can also send a message to itself). Net_MB therefore contains a list of messages, ordered by emission time. The network process extracts the messages one by one from the mailbox. It then delays them for a certain time T_{del} in order to simulate the time the message needs to travel from the sending node to the receiving node. During this time, the network can not be used for sending other messages.
Therefore, as soon as m is at the front of the list of messages queued in

Net_MB, the network process waits for time T_{del} and then drops m into n_j's L_Comm mailbox.
Since in reality T_{del} depends on the distance between the nodes n_1 and n_2, it is in our simulation exponentially distributed with some medium value specified by the user.

The access to the Ethernet is simulated by the unique network input mailbox and by a delay imposed in the communication module. The message delivery time is modeled by T_{del}. In order to comply also to the TCP protocol, the network layer does not provide a broadcast primitive. In contrary, only point-to-point communication is supported, thereby simulating the notion of virtual channels.
It is important to see that the network, in this conception, does not now anything about the type of the messages. It simply sends a message from one node to the other.

The implementation, however, violates this abstraction from the point of view of the network. This violation is due to the fact that broadcast messages are not entirely copied, but only their pointers. Not copying the entire messages saves a lot of memory.
A broadcast message m therefore consists of N pointers to m, referred to as $ptm_1, ptm_2, ...$ The network then receives N messages, i.e. pointers to m. It has to distribute them to all the nodes in the system. Usually this is done by inspecting the destination entry in a message, which identifies the destination node of this message. However, a broadcast message ptm_i can not use this entry to specify its destination, since all ptm_is would then be sent to the same destination.
In order to solve this problem the network needs to know about the type of messages sent. Every time it detects a broadcast message m, it will end up receiving N copies of this message. It therefore maintains a circular counter, which counts to $(N-1)$ and then restarts again at 0. A broadcast message is sent to the node, whose ID is equal to the counter value. Then the counter is increased by one. Since one node sends all the broadcast message copies atomically, every node will receive the message. The network maintains such a counter for every node in the system.

Send-to-all messages are treated in the same way.

The network is implemented in the files communication.h, communication.cpp.

8.3 Communication Module

This section discusses the implementation of the atomic broadcast algorithm. First some general problems are considered before discussing each layer of the communication module.

8.3.1 Message Structure

Until now not much has been said about the structure of a message. Every message sent by the communication module is inherited from the base class Cl_Message. Conceptually, this is the only class known to the network layer. It includes some basic information about the message, as for instance the identifier of the sending process. The communication layer has to deal with two different types of messages: *broadcast* and *point-to-point* messages. Both message types (Cl_Bcast_Msg and Cl_PtP_Msg) are subclasses of the base class Cl_Message. To distinguish these two classes, the base type Cl_Message contains a tag.
Every layer on top of the communication layer can inherit new message types from the existing ones and redefine or add additional members. The consensus defines its own broadcast primitive (Cl_Cons_Bcast_Msg) and also Cl_Cons_PtP_Msg and Cl_Cons_StA_Msg. The last two primitives are used for point-to-point communication and for send-to-all communication. All three primitives contain, compared to their superclasses, additional information specific to the consensus. In particular they are all tagged with the consensus round number and the current consensus number. The atomic broadcast has its own message type (Cl_ABcast_Msg), too. It is this message type which is exported to the users of the communication module. The inheritance structure of the different message classes is illustrated in figure 8.3. An edge from class cl_2 to class cl_1 signifies that cl_2 inherits from cl_1. In contrary to usual representations, the

base class Cl_Message is at the bottom of the figure. This choice has been made in order to show the correspondence to the layered architecture of the communication module.

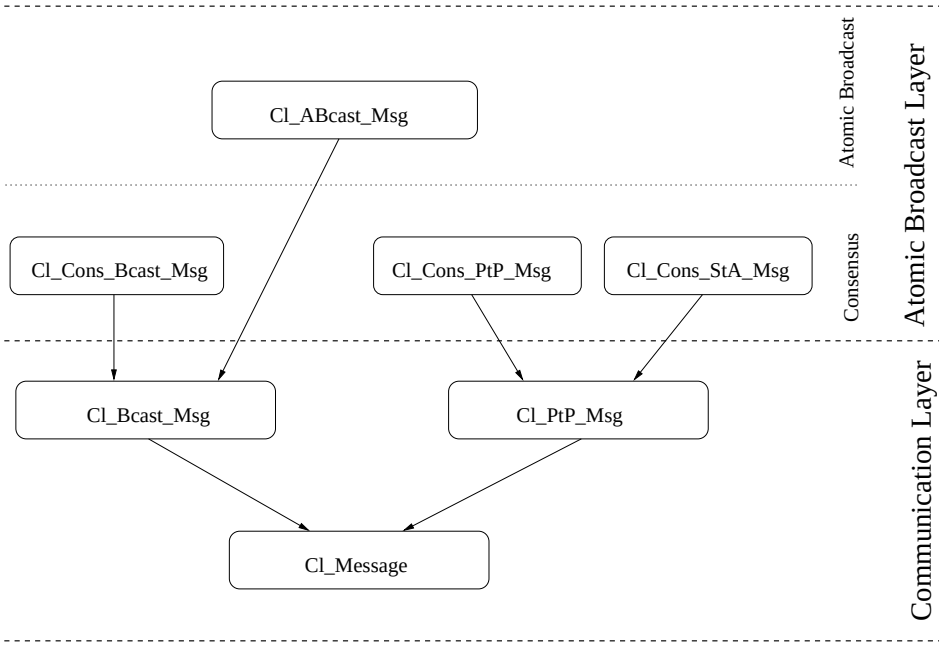

Figure 8.3: Inheritance structure for atomic broadcast messages

Since in upper layers messages have to be copied, every message furnishes its own virtual function make_copy. This functions returns with an exact copy of the corresponding message. With this primitive the advantages of real object-oriented design can be exhibited.

8.3.2 Message Destruction

An important problem that has to be addressed in the communication module is the destruction of unused messages. Since a huge number of messages are created during a simulation, it is necessary to free all storage once the messages are not used any more. In order to keep the number of messages in the system small, we decided not to copy the entire message N times in a broadcast, but only the pointer to the message. This approach reduces storage requirements for broadcasts almost

by a factor of N. However it requires a rigorous handling of messages in terms of changes to message parameters and destruction of messages. The latter implies the introduction of some garbage collection scheme. Every broadcast message m is therefore tagged with a *reference counter* which keeps track of the number of pointers referencing m. Every time a message is copied (i.e. its pointer is copied), the reference count is increased by one. On the other, hand upon every attempt to delete the message, the same counter is decreased by one. Once it is equal to 0, the message is deleted. A major problem to the garbage collection scheme using reference counters is that circular lists are never detected as unused, since their reference count is not equal to 0. This case, however, does not occur in the current project. In the following, when we speak about the destruction of a broadcast message, we actually mean these activities.
The same approach holds also for the destruction of send-to-all messages.

Evidently, the a-delivered messages are referenced only once and can be deleted by the user of the communication module the moment it is necessary.

8.3.3 Communication Layer

The communication layer is also implemented as a class, called Cl_Communication. Like the network, it basically provides the function loop, which handles all activities of the current layer. In every pass through the loop, it waits on an event-set. Every time an event is set, the call to wait_any returns the number of the event. It is important that the event is cleared at this point. Otherwise a message could already have been dropped into the mailbox before we clear the event. The signal on the event would then get lost and the process would not know that there is a message in this mailbox. All the messages in the corresponding mailbox are then retrieved and processed:

- every message from the upper layer is forwarded to the network. Broadcast and send-to-all messages are converted into N point-to-point messages, one for every destination node.

- if a broadcast message m has been received, we first check whether it has already been received. If it is the first time we received m, it is delivered to the upper layer by dropping it into the mailbox L_ABcast. In order to notify the upper layer of m's arrival, the event ABcast_Evts is set. In addition m is sent to all other nodes. In order to prevent the protocol from entering an infinite loop, this action is only performed if the current node has not been the initiator of m. If the broadcast has already been received, no actions are taken. In any case the message is deleted if we hold the last reference on it. Point-to-point messages are simply delivered to the upper layer. They are not deleted at this point since the upper layer now holds a reference on them.

A point that is worth to be mentioned is how we determine that a broadcast message has been received the first time. In the current version a very simple approach has been chosen. A counter simply tags every broadcast with a unique number. This number increases with every broadcast message sent on the same node. Every node stores the number b_k of the last broadcast message received from node k. When node n_i receives broadcast message m from node n_j the following actions are taken:

- if the number b of the broadcast message m is less than or equal to the number b_j, m has not been received the first time. Therefore ignore it.
- otherwise update the number to $b_j := b$ and process the message according to the algorithm presented above.

This approach relies on two assumptions. First, the communication channels have to be first-in-first-out (FIFO), meaning that the order in which messages are emitted into a channel is preserved. This is guaranteed by TCP. The second assumption states that the total number of messages sent at one node is limited to a parameter, called BCAST_NBR_INTERVAL. By adding some sort of sliding window protocol to this approach, the number of messages a node can broadcast would become infinite. Such a number of messages, however, is not needed at the current stage of the

simulator, since simulations are not made over such a long period of time.

In fact all communication, whether it is send, send-to-all or r-broadcast is done using the function send. This function simply drops all the messages into the network mailbox. The fact that every message is finally sent by this function allows to extract measurements about the number of messages sent by the system and about some other system parameters.

The communication layer is implemented in files communication.h and communication.cpp.

8.3.4 Consensus

CI_Consensus exports the primitive propose that is called every time the atomic broadcast wants to reach a decision over a set of messages. It returns the decided set of messages. The current consensus number and the proposed estimate are thereby the input parameters.

The implementation of the function propose follows exactly the algorithm given in figure 3.4. However, since the a-deliver is only implemented using one process, messages received while executing this function might not belong to the consensus. Upon reception they are passed to a message handler installed in the consensus object at initialization. This message handler then treats the messages accordingly. In this approach the consensus does not have to know anything about messages that do not concern it, apart from actually identify them as messages not belonging to the consensus.
It is important to distinguish messages of different consensi and within one consensus, messages of different rounds. Every message of a previous round or even of a previous consensus is deleted following the deletion procedure explained in subsection 8.3.2.
The time-out value of the timed receive statement in phase 3 has been made an user input parameter. It has been noted that this parameter has a considerable influence onto performance of the atomic broadcast

protocol and therefore of the whole RDBS simulator.

As mentioned before, the consensus defines its own message types. All this new message types are derived from the classes implemented in the communication layer. They contain some additional information compared to their superclasses. The class Cl_Cons_StA_Msg is a special case of a point-to-point message. It differs only by its value in the message destination member. While point-to-point message indicate their receiver at this place, the send-to-all message puts the negative value To_All into it. It is by this value that the communication layer can distinguish the two message types.

The files consensus.h and consensus.cpp implement the consensus.

8.3.5 ABcast Layer

The atomic broadcast layer defines the object Cl_Atomic_Broadcast which is initialized with the total number of nodes in the system and a node ID. It also exports the well-known primitive loop. In addition the message class Cl_ABcast_Msg and a type Cl_ABcast_Msg_Handler are specified at this place. The latter is a subclass of the message handler defined in the consensus. It is initiated with a pointer to the R_delivered list, which contains all the messages received from the lower layers but which have not been decided yet.

Cl_ABcast_Msg is a subclass of the type Cl_Bcast_Msg. It only adds additional members which allow to measure the response time of the communication module.

The class Cl_Atomic_Broadcast, finally, installs an object of the message handler class in the consensus object. When its primitive loop is called, it enters an infinite loop. In every pass it first checks whether the R_delivered list is empty. If there is no message in this list, it waits for incoming messages. When a message is received, the message type is determined and the message is treated accordingly. Any consensus message is ignored (deleted if reference counter is equal to 0) while a copy

of the A-broadcast messages is appended to the R_delivered list. The received message then is deleted. The fact that we store a copy of the received message in the R_delivered makes upper layers independent from whether copies of messages are sent or simply copies of pointers.

The function propose of the consensus is then called with a copy of the messages in the R_delivered list as initial estimate. The estimate is an object of type CI_Decision_Value. It defines a list of messages, ordered by increasing initiating node ID and message number. It is important to see that every message in the R_delivered list is only referenced once and belongs to the atomic broadcast object on the corresponding node. When this function returns with a decision, all the messages in this decision list are deleted from the R_delivered list and written into the U_ABcast mailbox. Since all the messages in the decision list are ordered, they are also delivered at every node in the same order.

The atomic broadcast is implemented in the files atomicBcast.h and atomicBcast.cpp.

8.4 Transaction Manager

The classes CI_Commit_Abort_Msg, CI_DB_Msg, CI_Commit_Abort, CI_Transaction, CI_Trans- action_Mgmt and CI_Op are defined in the transaction manager. The transaction manager is implemented in the class CI_Transaction_Mgr. We will briefly look at the purpose of these classes.

In addition a record defines an operation. The first field distinguishes read from write operations, while the second field determines the address of the accessed data item. The last field then is assigned for the update value of the corresponding data item in case of a write operations. In order to simplify the implementation, write operations do not specify an update value. This value is randomly determined in the data managers at the moment it is written to the DB.

Every transaction has a unique identifier, composed of the identifier (process ID) of the initiating node and a number that is unique on this node.

It will be called *transaction identifier* in the following. In addition, each transaction holds a round number. When a transaction is aborted, it is restarted by the transaction manager with a new round number. If operations of the previous round are still in the system, they can be distinguished from the operations of the new execution and treated accordingly. This round number has nothing to do with the round number in the consensus.

Cl_Commit_Abort_Message

This class is an immediate subclass of the message type exported by the atomic broadcast. It is used by the transaction manager to broadcast commits or aborts for a transaction. In order to identify the transaction the commit or abort belongs to, a transaction identifier in the class. A tag allows to distinguish commits, aborts and write-sets of transactions. In addition, the current round number is indicated.

Cl_DB_Msg

Cl_DB_Msg, an immediate subclass of the previous class, is used to broadcast transaction's write-sets to all nodes. Upon initialization it sets the tag of its superclass to normal, thereby indicating a write-set of a message. An array is used to store the operations of the transaction.

Cl_Op

Objects of this type circulate from the transaction manager to the lock manager and between the lock manager and the data manager. Every such object specifies one particular operation. While the communication between transaction and lock manager only consists of objects for read operations, the lock manager and the data manager exchange objects of both operation types.
Every operation is identified by the transaction identifier. In order to recognize out-of-date objects the round number is sent with every object. In addition an object of class Cl_Op also contains the data item's

address and the type of the operation. Further, some flags indicate the state of the operation execution. In particular the Done_Flag is set when the operation has been processed, the Exec_Flag becomes true when the operation is executable and finally the Undo_Flag indicates whether an operation execution is in the undo mode, after its transaction has been aborted.

Cl_Commit_Abort

In order to make abstraction of the broadcasting operation in the lock manager, all the commit or abort messages received by the interface adaptor are converted into objects of type CL_Commit_Abort. Besides the transaction identifier and the operation ID it also contains the round number. The parameter specifying the operation ID indicates the number of the considered operation within a transaction in the case of read operations. For write-sets it is set to WRITE_SET_ID. All commits or aborts sent from the lock manager to the transaction manager also have this type.

Cl_Transaction

Objects of type Cl_Transaction are stored in the transaction lists of the TM and the LM and are sent from the interface adaptor to the lock manager. Cl_Transaction is an immediate subtype of class Cl_Commit_Abort. In addition it contains an array of the operations and a done flag attached to all operations. Two flags, Committed and Aborted, indicate the state of the transaction. If both flags are unset, the transaction is either running or waiting (see figure 4.3).

Cl_Transaction_Mgmt

This class is used to keep track of the transaction currently in process in the transaction manager. It provides primitives to add, remove, find and count transactions that are objects of type Cl_Transaction. All the transactions are stored in a list. A certain number of queries can be invoked on the list of transactions. In particular we can query whether a

transaction has been aborted, committed or whether all operations of a transaction are done. Since this list contains only transactions initiated on the current node, any transaction can be identified using simply its number. This transaction number is given as a parameter to the query functions. These functions then call get_Particular_Trans in order to get a pointer to the corresponding transaction. However the transaction still remains in the list. Its storage is only deleted when the function remove_Trans is called with the corresponding transaction number as a parameter.
The function process_Next_Read sends the next read operation to the lock manager. If they are all processed, it returns NO_READ_OP_LEFT. Write-sets are processed by the function process_Write_Ops, which a-broadcasts the write operations of a transaction in the same message. Finally, the primitive sent_Write_Set returns true if the write-set of the transaction has already been a-broadcast.

Cl_Transaction_Mgr

Containing the implementation of the transaction manager, Cl_Transaction_Mgr also defines a primitive loop. This function is responsible for processing the transactions. Thereby, always sim_Number_Trans_Proc transactions are processed in parallel. This parameter is defined in a simulation parameter input file.
If the number of transactions currently in process is less than this input parameter, new transactions are read from the transaction input file. Their execution is immediately started. A special case occurs if a transaction in the input file contains only read operations. Since we explicitly excluded this case, the last read operation is changed to a write operation in the function get_Ops. This simplifies the generation of an input file considerably and provides a certain degree of input data verification.
If sim_Number_Trans_Proc transactions are in the transaction list, the transaction manager waits for any incoming message from the lock manager. If it receives a commit for a read operation, the next read operation is sent to the lock manager. If no read operations are left, the write-set is a-broadcast. A commit message for a transaction's write-set implies the a-broadcasting of the commit as well as a call to remove_Trans of the

transaction management.
Since abort messages only occur for read operations, the TM checks whether the write-set of the corresponding operation has already been sent. This is the case when all the read operations are tagged as done. The TM then a-broadcasts the abort message to all other nodes, increments the round number and resets all done flags of the transaction. After this activities the first read operation is sent again to the lock manager.

All the data structures presented in this section are implemented in the files transactMgr.h and transactMgr.cpp.

8.5 Interface Adaption Layer

The interface adaptor is implemented by an object called Cl_Interf_Adapt. Every interface adaptor is identified by its node ID. Like all components the interface adaptor also exports the function loop. Once the interface adaptor is started, it blocks on a message reception event. When a signal occurs on this event the interface adaptor resumes its execution and receives messages in the communication module's output mailbox U_ABcast. A message is either a commit message, an abort message or a transaction's write-set message. The interface adaptor then converts them to objects of type Cl_Commit_Abort or descendant type Cl_Transaction. After having dropped them in its output mailbox Int_out, it signals the corresponding event in the lock manager.

Files: interfAdapt.h, interfAdapt.cpp

8.6 Lock Manager

In all the explanations in this section it is important to note the difference between an operation that is *executable* and an operation that is *executed*. In the former case the meaning is that the operation is ready to be executed, i.e. there is no other conflicting operation waiting for the lock. However no data manager process has started the execution of the operation yet. In the second case, a DM process is charged to execute

the operation.

8.6.1 Transaction Handler

The transaction handler is implemented in class Cl_Trans_Handling. It stores all the transactions, whose operations are currently executed on this node, in a list. Primitives allow to insert, remove or consult the transactions in this list. Transactions are thereby identified with the transaction number.

One might imagine data structures with faster retrieval than a simple list in order to speed up the DBS. For instance, instead of a list, a hash table could be used.

8.6.2 Lock Table

The implementation of the lock table is based onto two classes: Cl_Lock_Table and Cl_Lock- _Table_Entry. In addition to these two classes, another class, Cl_Abort_List is defined. It consists of a list of records, containing fields for the transaction identifier, the operation ID and round number. These information is needed in order to store all the transactions to be aborted in this list while processing a write-lock request.

Another important data structure in the lock table is the list called Exec_Op_List. It contains pointers to the executable operations. These pointers are actually the addresses of the data items accessed by the executable operations. They allow to find the corresponding lock table entry. The executable operation is then the next operation waiting for the corresponding lock.

Cl_Lock_Table_Entry

As implies already its name, Cl_Lock_Table_Entry models the lock onto a single data item. The data item address is used to identify the lock. Every lock object contains two lists, Op_List_Granted and Op_list_Wait. The

former links all the operations to whom the lock has been granted and that are currently in process or are already done but not yet committed or aborted. On the other hand, the Op_List_Wait contains the operations still waiting on the lock, either because of a conflict or because no data manager process is available. A predefined list structure has been used for the implementation of these two classes.
Upon creation of an object of type Cl_Lock_Table_Entry, the address of the data item to protect is passed in the argument as well as a pointer to the executable operations list. Every lock which contains an operation that is executable, i.e. that does not conflict with another operation, has its data item address written into this list. The operations will then be executed in the order they are stored in the list. Even though the order of operation execution (non-conflicting operations) in the lock manager is not important, this approach prevents starvation of a operation. A challenging task is thereby to keep the consistency of this list also in case of conflicting operations and aborts.
In addition, other features are provided by Cl_Lock_Table_Entry. We will discuss the most important ones:

- ***request_Read_Lock*** This function is called every time a read lock is requested. It first verifies if the corresponding operation really is a read operation. When the Op_List_Wait is empty and the Op_List_Granted only contains read operations, then the operation is appended to the waiting list and its executable flag is set. In addition the data item's address the current lock is protecting is appended to the Exec_Op_List, since it is ready to be executed. Note that the executable flag is set whenever the data item address is inserted into the executable operations list Exec_Op_List.
  If the waiting list is not empty, the read operation is simply appended to this list.

- ***request_Write_Lock*** Takes a pointer to a write operation *op* and a pointer to an object called Cl_Abort_List as parameters. The latter contains at the end of the write-lock request the identification of transactions whose read operations conflict with *op* and have to be aborted.

Write operations overtake read operations in the waiting list.

- ***executable_Op*** Returns true whenever an operation is executable. The flow chart in figure 8.4 shows when an operation is executable. This is the case when no operation holds a lock on the corresponding data item. If the granted list is not empty, then the operation at the front of Exec_Op_List is only executable, if it is a read operations and all operations already holding a lock are also read operations. A special case occurs when read operations of the same transaction as the operation requesting an exclusive lock hold shared locks on the data item. In such a case the write operation is also executable. When no operation requests a lock, executable_Op returns false.

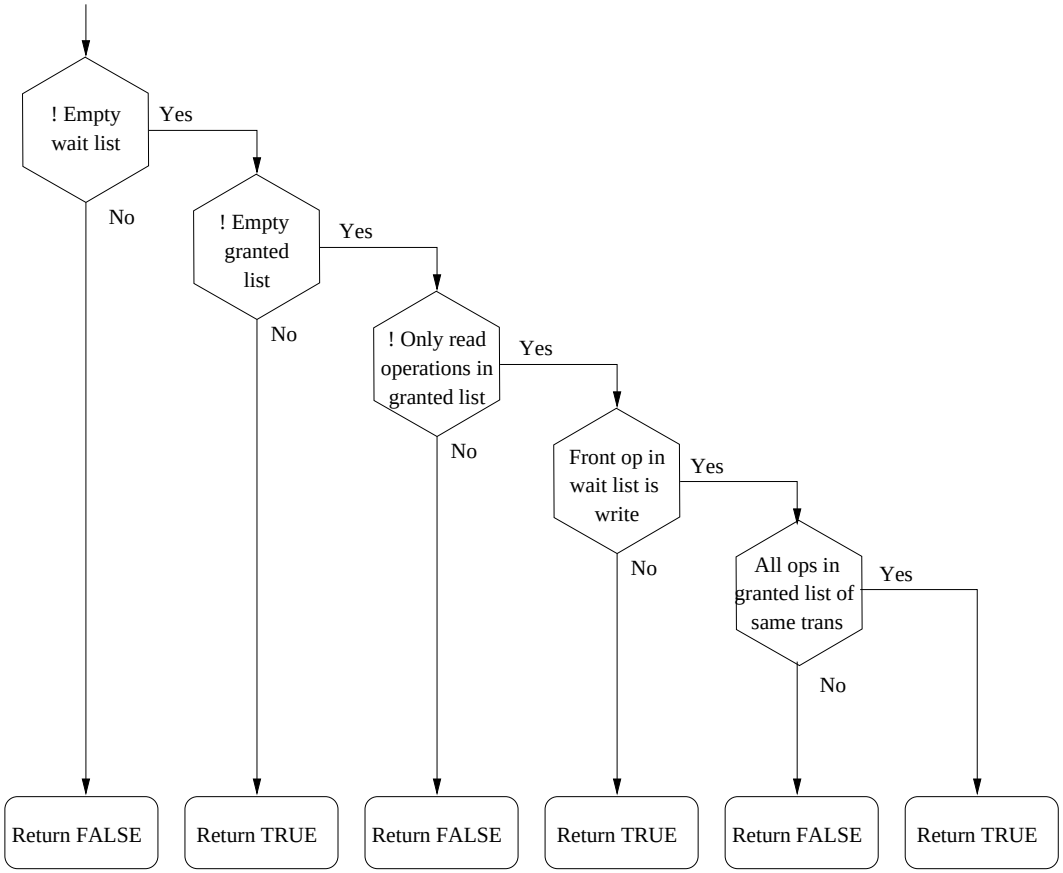

Figure 8.4: Flow chart indicating when a lock table entry contains an executable operation

- ***get_Executable_Op*** Returns a pointer to the next executable operation. It has to be the operation at the front of the Op_List_Wait. This operation is removed from the waiting list and appended to

the granted list. If the next operation in the waiting list is executable in this new situation, its data item address is appended to the Exec_Op_List and its executable flag is set.

- ***delete_Lock*** First looks for the operation *op* in the granted list. If *op* is in this list, it verifies if *op* has already been processed. If its execution is still in process, nothing is done for the moment. Otherwise *op* is deleted from the granted list and the Exec_Op_List is updated. Then delete_Locks verifies whether the first operation in Op_List_Wait has become executable, if there is any.
  If the operation has not been found in the granted list, then it is contained in Op_List_Wait. We loop through the whole list in order to preserve the order of the operations. A pointer is kept on the searched operation. If the operation was at the front of the waiting list, it might have appended the lock's data item address into the executable operations list. This can be checked by inspecting the executable flag. If it is set, the corresponding entry in the Exec_Op_List is deleted. Then the situation is reconsidered with a call to get_Executable_Op.

- ***abort_Lock*** Gets an operation identification (Initiating node of transaction, transaction ID and operation ID) and the current round ID of the transaction in the argument. It does exactly the same as the function delete_Locks.

- ***undo_Updates*** Takes an identifier of the operation *op*, the corresponding transaction's ID and initiating node in the argument as well as the current round ID and a DM process identifier DM_i. Looks for *op* in the granted list, sets *op*'s undone flag and drops it into the DM_i's mailbox.

Granted locks are only appended to the granted list when they are actually executed. This means that a lock might have been granted, but no data manager process is available to process the corresponding operation. In this case the operation is still kept in the waiting list. It is appended to the granted list as soon as a data manager executes it. This allows to determine which write operations have to be undone in case of

an abort, namely the writes in the granted list.

Every time a lock request is treated, the lock table checks whether an operation gets executable.

Cl_Lock_Table

This class implements the structure of the lock table. In particular, it defines a hash table with the data item's address as key. Every entry in this hash table is either null, when no locks on the corresponding data items have been requested, or consists of a list of Cl_Lock_Table_Entry objects. In the current version of the RDBS simulation tool the hash function $h(x)$ is simply

$$\begin{array}{ll} h(x) = a \bmod M, & \forall a, 0 \leq a < D,\ a\ integer \\ where & M:\ size\ of\ lock\ table \\ & D:\ size\ of\ the\ DB \end{array}$$

Most of the functions of this class are the equivalent to the primitives in Cl_Lock_Table_Entry. They just walk through the list of operations in a transaction object of type Cl_Transaction and apply the corresponding Cl_Lock_Table_Entry primitive to every operation. A special case presents the function abort_Locks. In addition to just calling Cl_Lock_Table_Entry::abort_Lock for every operation, it distinguishes between write and read operations. All write operations that have been done involve a call to Cl_Lock_Table_Entry::undo_Updates.

The function execute_Next_Op is of particular interest. It takes the identifier of a DM process (DM_i) as input. First of all it checks whether the Exec_Op_List is empty. If this list is empty, nothing has to be done. Otherwise the front element of the Exec_Op_List is extracted and the corresponding lock is accessed. A call to the function get_Executable_Op of the lock object returns a pointer to the executable operation. This operation is then dropped into DM_i's mailbox.

8.6.3 Cl_Lock_Mgr

This class exports the primitive loop. It is responsible for the synchronization of the communication with the other components in the DBS. We have already seen how this synchronization is done conceptually in section 6.7.4.

Synchronization with the Data Managers

The lock manager controls the operation execution. An operation can only be processed, if it is executable and if a data manager process is available. The only exception to this approach occurs, when a transaction has been aborted and its write operations need to be undone. In order to process them as soon as possible, they are immediately sent to a data manager. However, no data manager might be available at this point. We therefore decided to chose arbitrary a DM process to whom we confine the write operation to be undone.

We could also have chosen another design for the execution of operations. It would have been possible to simply drop every executable operation into the mailbox of some data manager process. However aborts would have become very expensive, since the operation to abort could be in the mailbox of a DM process, waiting for some other operation(s) to finish. For example suppose that the mailbox of DM process DM_l contains the operations $[o_i, o_j, o_k]$ with o_i in process at the moment. If the transaction T_k gets aborted, there is no way of preventing DM_l from processing o_k, even though this is not necessary any more. The execution of o_k is delayed until o_i and o_j are processed. Worse, if o_k has been a write operation, it would have to be undone again. By keeping the mailbox queue length at 0 or 1, the probability that the operation is already in the mailbox queue when an abort for the corresponding transaction occurs gets much smaller. Nevertheless it is possible for the mailbox queue to get longer than 1, namely in the case of an abort of a transaction. When the effects to the DB of a write operation have to be undone, this operation is given to a arbitrary DM process, because all DM processes might be busy at the moment. Therefore a DM process could actually

contain more than one operation in its mailbox.

Synchronization with the Interface Adaptor

All objects received from the interface adaptor are of type CI_Commit_Abort or its subtype CI_Transaction. An event is assigned to the mailbox Int_out in order to signal the message arrival to the lock manager.

Every time a write-set of a transaction T is received, the round number attached to this write-set is compared to the round number of T's entry in the transaction handler. It is possible that the write-set's round number is smaller than the round number in the transaction handler. This means that, while the write-set was broadcast, T has been aborted and restarted. Therefore some of T's read operations might already have been received by the lock manager and are processed. In this case the lock manager recognizes this situation by using the round number and ignores this write-set of T. If the write-set's round number is equal to the round number of T's entry in the transaction handler and T has not been aborted in the meantime, then the write locks are requested for T.

Synchronization with the Transaction Manager

The objects sent from the transaction manager to the lock manager are of type CI_Op and are single read operations. In this case it is also necessary to verify, for every received read operation, whether its round number is still valid. The round number of a read operation $r_i[x]$ can only become invalid, when T_i has been aborted after the transaction manager sent $r_i[x]$ to the lock manager, but before the latter has received $r_i[x]$. In such a case the read operation is ignored.

Figure 8.5 illustrates the activities of the lock manager when a read operation of transaction T is received from the TM.

First the lock manager checks whether T is already in the transaction list. If it is not, then the operation ID of the read operation has to be 0, because we expect the first operation of T, and the lock can be requested. In the other case an out-of-date read operation arrived.

If T is in the transaction list, and the operations round number is higher than the round number in T's entry in the transaction list, this entry is removed. This situation occurs when T has been aborted in the meantime. Then the read lock is requested and a new entry is made in the transaction list.

Coordinating Interactions

The lock manager uses an event-set to coordinate the different interactions. The event with index 0 in this event-set is assigned to the mailbox Int_out. The mailbox TM_out works in cooperation with the event indexed by 1, whereas the mailboxes Lock_in occupy the remaining events. This order implies a priority on the handling of the event. The events with the lowest indices have the highest priorities.

The following files implement the behavior of the lock manager: lock_Table.h, lockMgr.h, lock_Table.cpp, lockMgr.cpp

8.7 Data Manager

The behavior of a DM process is accomplished by a class called Cl_Data_Mgr. This class exports a primitive called loop, which contains an infinite loop. In this loop the data manager waits for any incoming message and processes it depending whether it is a read or a write operation. While read operations simply do nothing, write operations update the accessed data item to a random value.

The reader might also have realized that the data manager processes are not signaled by an event when a message has been dropped into the corresponding mailbox. Since DM processes only inspect one mailbox, there is no need for the introduction of an event. Remember that the termination problem has not been addressed. Certainly this additional feature would be needed in order to terminate the DM processes properly. Setting the event at the end of the simulation would then ensure that the DM process executes the break condition statement and eventually terminates.

There exists multiple data manager processes per node. Each process differs by a unique *global* identifier from all the others. DM_j specifies the data manager process with global identifier j. The same identifier is also attached to the corresponding mailboxes Lock_in and Lock_out. In addition to the global identifier every DM process has a *local* ID. In particular this local ID will be used to signal the corresponding event in the LM. We will use the notation DM_{ik} to identify the k^{th} DM process on node i. Consider for example the database system node i. The function

$$\begin{aligned} j = f(i,k) = i + k * N, \quad & \forall i, 0 \leq i < N, \\ & \forall k, 0 \leq k < L,\ i,k\ integer \\ where \quad & N:\ number\ of\ nodes\ and \\ & L:\ number\ of\ DM\ processes\ per\ node \end{aligned}$$

computes the global index j in order to access DM_{ik} from the lock manager LM_i. It will be the mailbox at the $(i + k * N)^{th}$ position in the Lock_out array.

When the DM process DM_j has executed an operation it is returned to the lock manager LM_i. For this purpose DM_j uses the mailbox Lock_in_j. In addition DM_j has to compute its local identifier k and the node ID i it belongs to from its global identifier j in order to signal the correct event in the LM_i. This is done by computing:

$$\begin{aligned} & i = g_1(j) = j \bmod N \\ & k = g_2(j) = j/N, \qquad \forall j, 0 \leq j < (N * L),\ j\ integer \\ & where \qquad\qquad N:\ number\ of\ nodes\ and \\ & \qquad\qquad\qquad\quad L:\ number\ of\ DM\ processes\ per\ node \end{aligned}$$

Since the first two events in the event-set of the lock manager are reserved for other mailboxes, DM_j signals the event $(k + 2)$ in LM_i's event_set.

The files dataMgr.h and dataMgr.cpp implement the data manager.

8.8 Input Parameters

Every important simulation parameter can be specified in a file called *params*. The RDBS simulation tool reads these values at the beginning

of its execution and starts a simulation run according to this parameters. In particular, the following parameters can be specified:

- the number of nodes in the system
- the global simulation time
- the different simulation delays
- the number of DM processes per node
- the size of the DB, ...

In addition, as already mentioned in section 6.5, the transactions can be specified in a second file. Every line determines the set of operations of a transaction. The first column thereby indicates the number of operations, followed by all the operation entries. Each operation entry consists of two numbers: the type of the operation and the data item accessed. In contrary to the simulation parameters, the transaction input file is read continuously during execution of the simulator.
It is important to see that all transaction managers actually access the same file. This approach allows to create only one such file, instead of N, where N is the number of nodes. Since CSIM does not support real parallelism, the processes never access this file simultaneously.

Having read all the input parameters, the simulator starts all the processes necessary to model the components of the RDBS. Then transactions start to get processed.

8.9 Initializing the System

When the system is started, it first reads the values of the input parameters. Then it allocates the storage for all components in the system and calls the CSIM function sim. This function attaches a process to every component and starts it on every node. The processes initialize the corresponding objects and call the function loop. An additional process is created for reporting the simulation results. It is called reporter. It prints

its measurements in regular intervals into a file, which allows to observe the behavior of the system during the whole simulation time.

8.10 Limitations

This section outlines the limitations of the current version of the RDBS simulation tool.

8.10.1 Internal Limitations

1. The maximal number of r-broadcasts that can be sent on one node is limited by the parameter BCAST_NBR_INTERVAL specified in file simulation_Globals.h.
2. The number of mailboxes, events, messages, processes and facilities is limited to some default values set by CSIM. The current version of the RDBS simulation tool has relaxed these limitations where it was necessary. If these limits are violated by some simulation, due to a big number of system components, the simulator stops with a CSIM error message. The user might want to increase these parameters in such a case.

8.10.2 External Virtual Memory Limitation

The main problem when running a simulation is the amount of memory that is required. It might happen that the memory requirements of the simulation exceed the virtual memory allocated to this application by the operating system. We could imagine different scenarios where this actually occurs:

- when the communication module can not handle all the incoming messages any more. These messages then get queued up and cannot be deleted.
- the longer the simulation runs, the more entries are created in the different measurement tables. These tables also require memory.

- the number of components in the system also limits the memory available for the actual simulation.

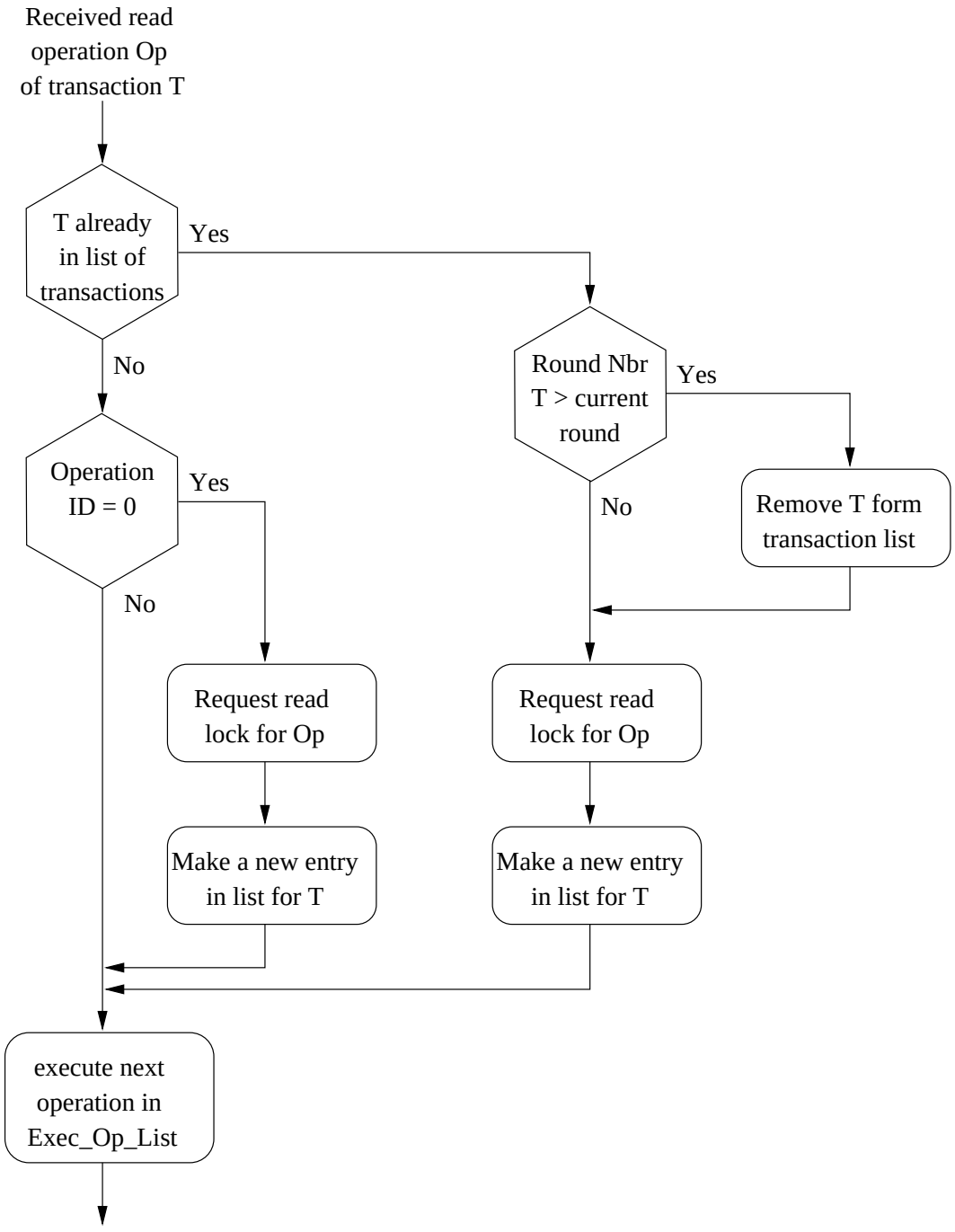

Figure 8.5: Flow chart indicating the activities of the LM upon reception of a read operation from the TM

Chapter 9

Conclusion and Further Work

9.1 Conclusion

We presented in this paper a simulation tool for a replicated database system that relies on an atomic broadcast primitive. It uses the serialization protocol proposed in [AAE96] to preserve database consistency and integrity. In this protocol the concurrency control is distributed between the lock manager and the communication protocol.
A modular, object-oriented architecture has been selected for the simulation tool. The system design is extensive in that it includes important components of a replicated database system like transaction processing, concurrency control and communication. Moreover it considers the hardware components that can be potential bottlenecks like CPU, disks and the network. It is easy to exchange single components of the system by others, that implement different protocols. This and it's highly parametrisied nature makes it an ideal tool for testing various database configurations.

The first set of testing has shown that the atomic broadcast proposed by Chandra/Toueg [ChT93] proves to be a serious bottleneck in the RDBS. This incurs basically from the cost of the reliable broadcast. Every broadcast produces N^2 messages in the network. Since we require reliable communication channels, the network is not able to handle the number of messages produced already in a system with a limited number of nodes.

The atomic broadcast algorithm has an indeterministic behavior, because we do not have any influence on the moment the consensus is started. This leads to a degeneration of system performance for some particular system configurations. We have seen that rotating the initial coordinator of the consensus could be a solution in such cases.

From this considerations we incur that the atomic broadcast algorithm has to be modified in order to use it in a real database system. This can be done either by using a reliable broadcast algorithm, which does not produce such a high number of messages, or by selecting another algorithm for the whole atomic broadcast.

The replicated database system simulation tool should prove to be a valuable tool for further work in this field of interest.

9.2 Further Work

As indicated in chapter 7, the testing of the current version of the RDBS simulation tool is not yet entirely performed. For a better understanding of the behavior of the RDBS further tests are necessary.
This chapter provides an outlook on the future work which can be done in this domain. Whereas in a first part we only consider the atomic broadcast, we will look on the entire RDBS simulation tool in the second part.

9.2.1 Atomic Broadcast Simulation

The tests performed up to this point are limited to a certain interval of message sending delays. Additional information and indications about the behavior of the atomic broadcast might be obtained by considering larger testing domains. In addition, special cases need to be paid more attention. This is especially the case for system configurations, which lead to a degenerate behavior in terms of message response time. In this context it is important to clearly understand the influence of the consensus onto the overall performance. Also it would be interesting to test

the protocol for different hardware configurations regarding CPU power and network capacity.
We also believe that modifications to the existing atomic broadcast protocol would allow better results in terms of response time. For example, a good improvement would be to decrease the number of messages needed for the reliable broadcast.

In future it might also become necessary to explicitly model the computation time used by the reliable broadcast, the consensus and the atomic broadcast. In our simulation this overhead is combined with the general overhead of sending and receiving a message.

9.2.2 Replicated DBS Simulation Tool

More measurements can be done using the RDBS simulation tool in order to estimate the cost of the serialization protocol and find the bottlenecks for different system and workload configurations. In particular, system configurations should be studied where some nodes fail. How does the RDBS perform in such failure-prone environment?
In addition it might be interesting to measure the performance of the RDBS with other communication modules or other serialization protocols.

A final step would be to implement a real system with the best performing serialization protocol.

Acknowledgements

I would like to thank Dr. G. Alonso and Dr. R. Gerraoui, who were responsible for this diploma project. Special thanks to B. Kemme, who provided support in all questions concerning database replication.

Bibliography

[AAE96] D. Agrawal, G. Alonso, A. El Abbadi: *Broadcasting in Replicated Databases*
University of California, Santa Barbara, and ETH Zuerich, Switzerland, 1996

[ADM92] Y. Amir, D. Dolev, P.M. Melliar-Smith, L.E. Moser: *Robust and Efficient Replication Using Group Communication*
Hebrew University of Jerusalem, Israel, 1992
University of California, Santa Barbara, USA, 1992

[Alo96] G. Alonso: *Partial Database Replication and Group Communication Primitives*
Extended abstract, Institute for Information Systems, ETHZ, Switzerland, 1996

[BHG87] P. A. Bernstein, V. Hadzilacos, N. Goodman: *Concurrency Control And Recovery In Database Systems*
Addison-Wesley, Canada, 1987
0-201-10715-5

[Bur94] D. K. Burleson: *Managing distributed databases: building bridges between database islands*
Wiley, New York 1994
0-471-08623-1

[CGM88] W. Cellary, E. Gelenbe, T. Morzy: *Concurrency Control in Distributed Database Systems*
North-Holland, Amsterdam, Holland, 1988
0-444-70409-4

[ChT93] T. D. Chandra, S. Toueg: *Unreliable Failure Detectors for Reliable Distributed Systems*
Technical Report 93-1274, Department of Computer Science, Cornell University, August 1993. A preliminary version appeared in the Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing, pages 325-340, ACM press, August 1991

[ElN94] R. Elmasri, S. B. Navathe: *Fundamentals of Database Systems*
Second Edition, The Benjamin/Cummings Publishing Company, Inc, Redwood City, CA 94065, 1994
0-8053-1748-1

[GuS95] R. Guerraoui, A. Schiper: *Transaction model vs Virtual Synchrony model: bridging the gab*
Swiss Federal Institute of Technology (EPF), Lausanne, Switzerland, 1995
To appear in Proc. International Workshop 'Theory and Practice in Distributed Systems', Schloss Dagstuhl, Springer Verlag, LNCS 938, 1995

[GLP75] J.N. Gray, R.A. Lorie, G.R. Putzulo, I.L. Traiger: *Granularity of Locks and Degrees of Consistency in a Shared Database*
Research Report RJ1654, IBM, September, 1975

[GrR93] J. Gray, A. Reuter: *Transaction Processing: concepts and techniques*
Morgan Kaufmann, 1993
1-55860-190-2

[FLP85] M. Fischer, N. Lynch, M. Paterson: *Impossibility of Distributed Consensus with one Faulty Process*
Journal of the ACM, 32(2):374-382, April 1985

[Lam94] Winfried Lamersdorf: *Datenbanken in verteilten Systemen: Konzepte, Loesungen, Standards*
Vieweg, Braunschweig/Wiesbaden 1994
3-528-05467-0

[San92] A. Sandoz: *Casual Approaches to Concurrency Control In Distributed And Replicated Database Systems*
These No 1036 (1992), Swiss Federal Institute of Technology (EPF), Lausanne, 1992

[Str95] V. Strumpen: *The Network Machine*
Phd Thesis, Swiss Federal Institute of Technology (ETH), Zuerich, 1995

Appendix A

Performance Measurements Tables

A.1 TCP

Msg delay [ms]	Response t min [ms]	Response t max [ms]	Response t mean [ms]	Net utilization	CPU utilization	CPU_1 utilization	Nb msgs deliv max	Nb msgs deliv mean	Nb msgs sent	Nb msgs rcvd
100	3.7	173.8	28.6	0.2862	0.2186	0.2492	7	1.217	2501.4	1519.8
500	4.0	483.8	40.5	0.0876	0.0666	0.0762	4	1.0462	502	501.2
1000	4.2	905.5	74.3	0.0444	0.0334	0.0382	3.6	1.0478	252.2	251.8
1500	4.0	1488.9	124.7	0.0298	0.0222	0.0258	4.2	1.0604	168.8	168.8
2000	4.7	1685.1	105.2	0.0226	0.0168	0.0196	2.8	1.034	127.4	127.2
5000	4.8	4093.0	256.1	0.0096	0.007	0.0082	1.6	1.013	52.2	52.2

Table A.1: Performance Measurements for 5 Nodes on TCP

Msg delay [ms]	Response t min [ms]	Response t max [ms]	Response t mean [ms]	Net utilization	CPU utilization	CPU_1 utilization	Nb msgs deliv max	Nb msgs deliv mean	Nb msgs sent	Nb msgs rcvd
100	8.5	262.3	73.2	0.9236	0.446	0.496	15	3.1574	3999.2	3982.8
500	6.5	580.1	40.0	0.3314	0.1582	0.188	8.8	1.087	804.8	804.8
1000	6.6	760.0	43.8	0.169	0.0812	0.0962	4.6	1.0528	402.8	402.6
1500	7.0	1047.8	58.2	0.1142	0.0544	0.0642	4.6	1.0488	271	271
2000	6.6	1155.9	59.4	0.088	0.042	0.0508	3.2	1.0292	204.6	204.6
5000	7.3	2502.4	157.8	0.0348	0.0164	0.0198	2.4	1.0214	84.6	84

Table A.2: Performance Measurements for 8 Nodes on TCP

Msg delay [ms]	Response t min [ms]	Response t max [ms]	Response t mean [ms]	Net utilization	CPU utilization	CPU_1 utilisation	Nb msgs deliv max	Nb msgs deliv mean	Nb msgs sent	Nb msgs rcvd
500	9.2	535.9	65.3	0.5594	0.2146	0.258	9.4	1.2714	1004	1002.8
1000	9.0	856.0	60.2	0.3038	0.1172	0.1414	6.2	1.1288	505.4	505
1500	9.2	1354.1	83.9	0.2044	0.0786	0.0952	8.2	1.1068	339	338.8
2000	9.5	1456.7	78.6	0.1564	0.06	0.0728	6	1.071	254.2	254.2
5000	9.1	2620.6	235.9	0.0634	0.0244	0.0296	4.8	1.0522	104.6	104.6

Table A.3: Performance Measurements for 10 Nodes on TCP

A.2 UDP

Msg delay [ms]	Response t min [ms]	Response t max [ms]	Response t mean [ms]	Net utilization	CPU utilization	CPU_1 utilisation	Nb msgs deliv max	Nb msgs deliv mean	Nb msgs sent	Nb msgs rcvd
100	2.6	188.4	28.4	0.0982	0.1488	0.1764	7.2	1.283	2501.4	1568
500	2.7	829.3	84.0	0.0298	0.0434	0.053	5.8	1.162	502.6	502.6
1000	2.8	1191.3	161.7	0.0148	0.0214	0.0264	5.4	1.1692	251.2	250.4
1500	3.1	1891.2	267.6	0.01	0.0146	0.0174	4.4	1.172	168.2	168.2
2000	3.1	1923.4	257.9	0.0076	0.011	0.0136	4	1.1274	127.4	127.2
5000	3.3	4962.3	1001.9	0.003	0.0044	0.0054	3.6	1.2332	53	52.4

Table A.4: Performance Measurements for 5 Nodes on UDP

Msg delay [ms]	Response t min [ms]	Response t max [ms]	Response t mean [ms]	Net utilization	CPU utilization	CPU_1 utilisation	Nb msgs deliv max	Nb msgs deliv mean	Nb msgs sent	Nb msgs rcvd
100	3.9	201.5	32.6	0.257	0.3084	0.3756	12.6	1.5854	3998.2	2691
500	3.6	644.5	57.9	0.079	0.0914	0.1198	8.8	1.2066	803.4	802.8
1000	3.8	944.6	91.3	0.0396	0.0462	0.0612	5	1.1596	403.4	402.4
1500	4.0	1573.5	140.6	0.027	0.031	0.041	6.6	1.1836	271.4	271.4
2000	4.1	1920.4	187.7	0.02	0.0234	0.0306	5.4	1.1724	203.8	203.2
5000	4.2	4788.2	620.6	0.0086	0.0098	0.0126	4.4	1.2262	84	83

Table A.5: Performance Measurements for 8 Nodes on UDP

Msg delay [ms]	Response t min [ms]	Response t max [ms]	Response t mean [ms]	Net utilization	CPU utilization	CPU_1 utilisation	Nb msgs deliv max	Nb msgs deliv mean	Nb msgs sent	Nb msgs rcvd
100	4.8	209.9	42.1	0.424	0.4546	0.5592	16.4	2.1152	5000.2	4298.4
500	4.5	570.6	57.1	0.1202	0.1276	0.1738	8.6	1.2732	1004.2	1003.4
1000	4.5	892.0	96.3	0.0616	0.0654	0.089	5.8	1.235	503.6	502.4
1500	5.0	1566.0	135.9	0.0412	0.0436	0.0598	8.4	1.23	338.2	336.6
2000	4.9	1796.0	172.4	0.0306	0.0326	0.0446	5.4	1.2362	254.8	253.2
5000	5.2	4571.0	450.3	0.0124	0.0134	0.0184	5.2	1.2922	104.8	104.4

Table A.6: Performance Measurements for 10 Nodes on UDP

Msg delay [ms]	Response t min [ms]	Response t max [ms]	Response t mean [ms]	Net utilization	CPU utilization	CPU_1 utilisation	Nb msgs deliv max	Nb msgs deliv mean	Nb msgs sent	Nb msgs rcvd
100	9.9	274.1	71.9	0.5962	0.6884	0.8034	34.4	5.7518	7497.6	7485
500	7.0	517.2	66.9	0.2388	0.2258	0.3166	12.2	1.5938	1505.6	1500.4
1000	6.9	927.6	96.1	0.1276	0.1198	0.1704	10.4	1.4364	755.2	753.6
1500	7.1	1450.2	163.7	0.0848	0.0798	0.1138	12	1.4294	507.2	506.2
2000	7.3	1887.9	196.8	0.0638	0.0604	0.087	11.2	1.3972	381	379.2

Table A.7: Performance Measurements for 15 Nodes on UDP

Appendix B

User Manuel

Since further work might be done with the RDBS simulation tool, this chapter briefly discusses its application. The discussion is divided into two parts: section B.1 presents the simulation environment for the atomic broadcast, whereas in the second part (section B.2) we explain how the entire RDBS simulation tool can be invoked.

B.1 Starting the Atomic Broadcast Simulator

The atomic broadcast simulation tool can be started with the following command:

./test params_com

The simulation parameters are specified in the input file params_com. Each line of this file consists of the name of the parameter and its value. Whenever the corresponding parameter specifies a delay, it has to be given in milliseconds. Table B.1 displays a possible parameter input file.

The parameter sim_Seed has not been explained yet. CSIM usually generates the same 'random' numbers for every simulation run. It basically keeps a sequence of random numbers and always accesses it from the beginning. Every time a random number is needed, the next number in this sequence is returned.
However, there exists a CSIM primitive which allows to specify the initial index at which the simulation should access the sequence of random numbers. The parameter sim_Seed specifies this index.

sim_Number_Of_Nodes	10
sim_Number_Msgs_Per_Node	10000000
sim_Seed	1
sim_Simulation_Time	50000.0
sim_Timeout_Value	1000.0
sim_Network_Delivery_Delay	0.1
sim_Send_Preparation_Time	0.2
sim_Receive_Preparation_Time	0.2
sim_Msg_Delay	500.0
sim_Delay_Var	0.1

Table B.1: Parameter input file for the atomic broadcast simulator

The following files implement the atomic broadcast simulation tool:

simulation_Globals.h
communication.h, communication.cpp
consensus.h, consensus.cpp
atomicBcast.h, atomicBcast.cpp
test.cpp

B.2 Starting the RDBS Simulation Tool

The RDBS simulation tool's main program is contained in the file db_test.cpp. It can be invoked with the command:

./db_test params infile

The file params contains the simulation parameters. Each line consists of the name of the simulation parameter followed by its value. Table B.2 gives an example of a parameter input file for the RDBS simulation tool.

The second input file contains the transactions. Every line defines one transaction. It starts with the number of operations in the transaction, before then specifying the corresponding operations. Every operation is given with its type (0 for a read operation and 1 for a write) and the data item it accesses. The following table presents such a transaction input file:

sim_Number_Of_Nodes	10
sim_Number_Msgs_Per_Node	10000000
sim_Seed	1
sim_Simulation_Time	50000.0
sim_Timeout_Value	1000.0
sim_Network_Delivery_Delay	0.1
sim_Send_Preparation_Time	0.2
sim_Receive_Preparation_Time	0.2
sim_Max_Size_Database	2000
sim_Lock_Table_Size	200
sim_Number_DM_Processes	5
sim_Number_Ops_Per_Transaction	6
sim_Nb_Trans_Proc	2
sim_Data_Access	40
sim_DM_Operation_Time	0.7

Table B.2: Parameter input file for the RDBS simulator

4	0	617	1	1895	0	1404	0	989				
6	1	629	0	1834	1	802	1	1570	1	1739	1	1348
6	1	831	0	1958	0	425	1	1474	0	1560	1	1913
5	0	118	0	1830	1	237	1	504	0	473		
4	0	1728	1	415	0	589	1	437				

Table B.3: Transaction input file for the RDBS simulator

The following files implement the RDBS simulation tool:

simulation_Globals.h
communication.h, communication.cpp
consensus.h, consensus.cpp
atomicBcast.h, atomicBcast.cpp
transactMgr.h, transactMgr.cpp
interfAdapt.h, interfAdapt.cpp
lock_Table.h, lock_Table.cpp
lockMgr.h, lockMgr.cpp
dataMgr.h, dataMgr.cpp
db_test.cpp

Appendix C

Code

This chapter contains the code of the RDBS simulation tool. In addition, the code used for the performance measurements of the atomic broadcast is added at the end (file test.cpp).