Bounded delay replication in distributed databases with eventual consistency

Johannes Matheis and Michael Müssig

2003-17-12

Abstract

Distributed real-time database systems demand consistency and timeliness. One approach for this problem is eventual consistency which guarantees local consistency within predictable time. Global consistency can be reached by best effort mechanisms but for some scenarios, e.g. an alarm signal, this may not be sufficient. Bounded delay replication, which provides global consistency in bounded time, ensures that after the local commit of a transaction updates are propagated to and integrated at any remote node within bounded time.

The DRTS group at the University of Skövde is working on a project called DeeDS, which is a distributed real-time database prototype. In this prototype, eventual consistency with as soon as possible (ASAP) replication is implemented. The goal of this dissertation is to further develop replication in this prototype in coexistence to the existing eventual consistency which implies the extension of both the theory and the implementation.

The main issue with bounded time replication is to make all parts, which are involved in the replication process predictable and simultaneously support eventual consistency with as soon as possible replication.

Contents

Chapter 1

Introduction

When a large amount of data needs to be stored in a structured and reliable manner, a database management system is often used. Some distributed real-time systems are data-intensive applications that might benefit from the properties offered by database systems. For real-time systems not only the correctness of data matters, but also the timeliness of operations must be ensured. These two requirements must hold for distributed real-time databases as well.

The traditional approach of immediate consistency leads to unpredictable delays due to, e.g. network partitioning or node failures, which may cause real-time transactions to miss deadlines. One way of solving this problem is to make the transactions independent from the network and remote nodes.

The DRTS group at the department of Computer Science at the University of Skövde started the development of a **D**istributed Activ**e** R**e**al-Time **D**atabase **S**ystem, called DeeDS ([And96]). In the current prototype, eventual consistency is used to reach global consistency. This ensures that global consistency is finally reached, but without any time bounds on the replication. However, there may be a need for reaching global consistency within bounded time in some scenarios, e.g.. when an alarm signal occurs on one of the nodes and this needs to be replicated in bounded time to other nodes to avoid system damages or 'damage' to people.

Johan Lundström has done some theoretical work for this problem in his M.Sc. dissertation, *A Conflict Detection and Resolution Mechanism for Bounded-Delay Replication* ([Lun97]).

Global inconsistencies are bounded in time using bounded delay replication. Temporal inconsistencies imply conflicts between nodes, which need to be detected and resolved. Lundström developed an algorithm for conflict detection that is used in our dissertation. Daniel Eriksson continued this work in his B.Sc. dissertation, *How to implement Bounded-Delay replication in DeeDS* ([Eri98]). Eriksson developed a software design and discussed implementation issues such as i.e. suitable data structures.
The purpose of our project is to extend the theory in Lundström and Eriksson, and also to implement bounded delay replication in DeeDS, in coexistence with the existing as soon as possible replication.

1.1 Bounded Delay Replication

In DeeDS, the distributed database is fully replicated and all transactions are executed on the local node. Bounded delay replication ensures that updates made by a local *transaction* are replicated to remote nodes in bounded time. Replication consists of both *propagation* to all remote nodes and *integration* of updates, at all remote nodes.
The propagation sends the updates to all remote nodes, after a local transaction has committed. For bounded delay replication, propagation of updates needs to be done in bounded time, which requires a real-time network. Finally, the updates have to be integrated on the remote nodes, also, in bounded time. The integration executes conflict detection and conflict resolution before writing the updates to the local replica of the database.
Bounded delay replication implies several problems that are addressed in this dissertation in the following way:

Theory: What assumptions and restrictions are required to ensure timeliness for the replication process. These assumptions must consider concurrent local database transactions and other consistency classes.

Software Design & Implementation: What modifications to the DeeDS design are necessary to enable bounded delay replication when combined with the existing *as soon as possible (ASAP)* replication. What needs to be changed or added to the current

implementation.

Test Environment: What kind of test cases are required and under which conditions should they run.

1.2 Results

A theory for bounded delay replication on a distributed real-time database system is developed. Critical points concerning timeliness are identified and discussed in particular in the context of two coexisting replication classes. Thereby, we show how bounded delay replication cannot be delayed by ASAP replication due to higher priorities of bounded delay replication. Several restrictions are needed to ensure that ASAP replication does not block bounded delay replication. These restrictions are explained and illustrated by examples in section 5.2.

We present a formula for the worst case time in bounded delay replication and determine factors on which it depends.

We present a modularized software design derived from the theory developed. This design aims at being easily extendable for further modifications of the replication module. Our implementation is based on that software design and extends the current implementation of the DeeDS prototype. It implements the conflict detection algorithm developed by Lundström [Lun97] and adds the bounded delay replication class.

Finally, the implementation is reviewed concerning predictability and timeliness. Furthermore, test cases validate our work and show desired behaviour in timeliness and functionality.

1.3 Segmentation of the Work

This project is a team project by two students, Michael Müssig and Johannes Matheis. Most of the parts of this project are so closely related that it is not possible to say who has done which parts in this project. For example, extending the existing theory, implementation of bounded delay replication in DeeDS, validation of the work and also writing the dissertation about the project. So both persons are involved in each part of the project. But from

our studies so far, Michael Müssig is mainly responsible for issues concerning the database, whereas Johannes Matheis is mainly responsible for the real-time constraints.

1.4 Outline of the Report

In chapter 2, background information about real-time systems, distributed systems, database systems, distributed database systems and an introduction to the DeeDS prototype are given. Chapter 3 describes the problem and the goals of this dissertation. In chapter 4 we present our approach to solve the problem of bounded delay replication in a distributed real-time database system. Some additional assumptions are described, an extended Software Design is shown, the implementation is explained, and the test cases for the validation of the implementation are described in the last part of this chapter. Chapter 5 contains the results and the validation of our project. The last chapter, chapter 6, summarizes the work and presents future work. Our research about suitable real-time operating systems for DeeDS is shown in appendix A. And Finally, appendix B shows several real-time ethernet protocols evaluated as possible real-time network solutions.

Chapter 2

Background

This project concerns the area of distributed real-time database systems. The main topics covered in this chapter are real-time systems, distributed systems, database systems, distributed database systems, and DeeDS, which is the system the bounded replication is implemented for. This chapter explains how the topics are related to each other and how the work fits in the area of distributed real-time database systems.

2.1 Real-Time Systems

A non-real-time system assures the logical correctness of the results of an operation. In a real-time system it is also necessary to concentrate on the timeliness of an operation. For example, in an automation system which uses a sensor to measure the temperature. When a critical temperature is reached the production process must be stopped immediately, otherwise the high temperature causes damages. When the computation of the sensor data takes too long, even if the computation of the sensor data was logically correct. Therefore a real-time system must meet the following requirements:

- Read the input on time
- Calculation of the output in time
- Delivery of the output on time

According to Mullender ([Mul93]) a real-time system can be defined as follows:

Definition 2.1 *A real-time system is a system that changes its state as a function of (real) time [Mul93].*

It is possible to classify real-time systems in different ways. One classification is based on the damage caused by missing a deadline. These deadlines may be hard, firm or soft depending on the result if a deadline is missed, this is shown in Figure 2.1.

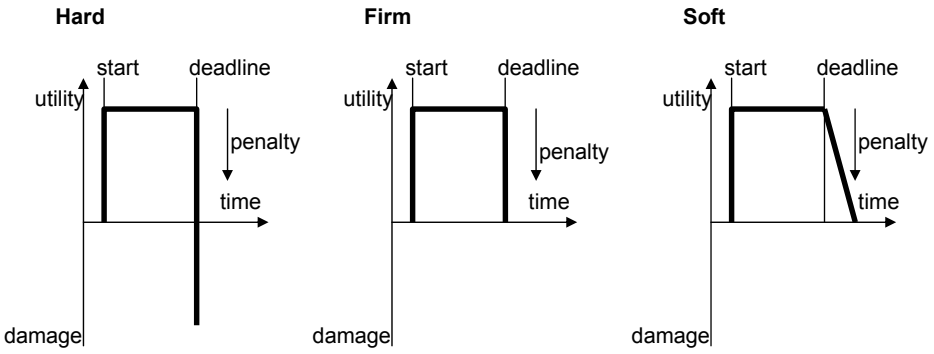

Figure 2.1: Value Functions

- **Hard deadline:** These deadlines have to be met, otherwise it may cause damages. If a hard real-time system misses its deadline it has a strong or even infinitive negative penalty on the system.
- **Firm deadline:** If this deadline is not met, the operation is aborted. The penalty of a firm real-time system for missing the deadline is (near) zero.
- **Soft deadline:** It is possible to exceed the deadline in defined borders without causing fatal faults. The value of a soft real-time system decreases over time when a deadline is missed.

Every real-time computer system has limited processing capacity. The temporal requirements for all real-time operations have to be guaranteed and a set of assumptions has to be set up. The two major assumptions needed about the system in this context are the *load hypothesis* and the *fault hypothesis* .

- **Load hypothesis:** The load hypothesis defines the peak load that is assumed to be generated by the environment. The peak load can be expressed by determining the

minimum time between - or the maximum rate of - each real-time operation [Mul93]. Peak load means that a maximum number of real-time operations enter the system at the same time. In many applications this happens only in *rare event situations* . A real-time system must be able to handle such load, even if the peak load can only occur in rare situations.

- **Fault hypothesis:** The fault hypothesis determines the possible types and frequencies of faults that is assumed in a fault-tolerant system. The system must be able to handle all these faults. During the handling of faults the system must still offer a certain degree of service. In the worst case scenario a fault-tolerant real-time system must be able to handle the maximum number of faults under peak load.

If these hypotheses are not chosen carefully, it is possible that the real-time system may fail. Real-time systems can be based on at least two different design approaches, the system can be event-triggered or time-triggered. An event-triggered systems reacts on stimuli of external events immediately. An event-triggered system is 'idle' until something happens. When an event occurs this is handled by the system immediately. A time-triggered system polls for external events at predefined points of time. Thus, a time-triggered system fits for an environment with periodic events.

2.1.1 Real-Time Operating Systems

A real-time operation system is an operation system that creates an output to a given input in predictable time. Every operation of a real-time operating system (RTOS) needs to be deterministic. For this reason, a real-time operating system has more properties than a non-real-time system.

The tasks of a real-time operation system are:

Process control is used for the allocation of the processor to different processes. **Synchronization** is needed for timing of processes. For communication between the processes the **interprocess communication** is used. Allocation of resources is done by the **resource management**. **Process protection** is needed as protection against unauthorized access. **Timeliness** and **concurrency** are necessary for the predictability of a real-time operation system.

Since a predictable operating system is needed for bounded-time replication, an analysis of existing real-time operating systems can be found in appendix A.

2.2 Distributed Systems

A definition of distributed systems is given by Burns and Wellings [BW01]:

Definition 2.2 *A distributed system is a system of multiple autonomous processing elements, cooperating for a common purpose. It can either be a tightly or loosely coupled system, depending on whether the processing elements have access to a common memory or not. [BW01]*

Distributed systems are used for decentralization to enable a faster access to information. Another reason for the use of distributed systems is redundancy to achieve fault tolerance. This is very important in real-time environments, where failures can be hazardous. For example when a real-time system in an airplane fails.
A number of problems arise during the use of distributed systems. Central issues for this dissertation are: concurrency, communication between different nodes of the distributed system and replication.

Concurrency: Every node of a distributed system wants to use shared resources or same variables. An example for a shared resource is the network. It has to be possible that all nodes are able to transmit their messages. For a real-time system it is also necessary that this can be done in predictable time. The usage of the same variables on different nodes implies the risk that this variable is written by one node and overwritten immediately by another node before it has been replicated.

Communication: By the use of communication media a transmission delay is always added for the communication between different nodes. There is also a risk of network partitioning. If the communication between the nodes breaks the entire system may fail.

Replication: When the same data is used on several nodes, the data must be replicated and simultaneously conflicting updates must be resolved to get a consistent system state. Consistency and Replication are explained in more detail in section 2.4.

2.3 Database Systems

Access and management of data is a very important issue in a large variety of areas. A database system makes it possible for many users to store and read data concurrently. For the access of data in a database, the concept of transactions is used to ensure multi-user service without being affected by operations of other users.

2.3.1 Transactions

A transaction consists of several operations that access the contents of a database. When transactions are executed concurrently without any implicit concurrency control technique, some undesired behaviour may occur. This derives from the fact that another user is accessing the database at the same time. These problems are described in [EN94] and briefly mentioned in this thesis.

The Lost Update Problem: This occurs when two transactions access the same database item in a way that makes the value incorrect. Suppose that transactions T1 reads item A. After that, the transaction is interleaved by another transaction T2 that reads and writes A. Finally, when T1 is running again it also writes A. The update of T2 is lost in this case since T1 overwrites the value based on the information of the obsolete first read of A.

The Dirty Read Problem: This occurs when one transaction writes a database item and then the transaction fails or aborts for some reason. If a second transaction has read the updated value before it is changed back to the old value due to the abort of the first transaction, the read value is called dirty data because this value is not allowed to exist in a valid database state.

The Incorrect Summary Problem: This problem occurs if one transaction is calculating an aggregate summary function while another transaction is changing some of these records. This may lead to an incorrect result since some values may be read before they are updated and some after.

To avoid these problems which may lead to an inconsistent database state, the execution of transactions is restricted to uphold the ACID properties.

2.3.2 ACID

The result of any transactions must not depend on the fact if they are running sequentially or concurrently. The ACID properties are used to ensure that all transactions lead to a consistent database state. [EN94]

Atomicity: A transaction is regarded as the smallest unit which means that either all operations of a transaction are executed or none.

Consistency: A transaction starting from a consistent state finishes with a consistent state. Otherwise it is aborted and all changes are reset.

Isolation: This property requires that concurrent transactions does not affect each other. A transaction should not make its updates visible to other transactions until it is committed.

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

2.3.3 Serializability

When transactions are executing concurrently, the operations from one transaction may interleave with operations from another transaction. The execution order of operations from the concurrent transaction form the schedule.

Definition 2.3 *A schedule (or history) S of n transactions T_1, T_2,..., T_n is an ordering of the operations of the transactions subject to the constraint that, for each transaction T_i that participates in S, the operations of T_i in S must appear in the same order in which they occur in T_i.[EN94]*

Transactions can either execute in serial order, which means that before a new transaction begins the prior has already committed, or execute concurrently which is more efficient for multi-user systems. Since there may be problems executing concurrent transactions without restrictions, the term serializability is used to avoid these problems and ensure the ACID properties for transactions.

Definition 2.4 *A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.[EN94]*

In order to guarantee these desired attributes, some kind of concurrency mechanism as i.e. the 2-phase locking protocol must be used.

2-phase-locking protocol

A scheduler working according to the *2-phase-locking protocol* ensures serializability. A transaction follows this protocol if every object accessed by the transaction is locked before and all locking operations precede the first unlock operation. Thus, the protocol consists of two phases: a growing phase during which the transaction acquires locks and a shrinking phase during which the set locks are released [EN94]. It is important to mind that during the growing phase no lock can be released and during the shrinking phase no new lock can be acquired.

The protocol just described is also called basic 2-phase-locking protocol since it has 2 drawbacks which are avoided by adding other restrictions. The basic protocol does not prevent cascading rollback of transactions. This is an absolutely undesirable property in real-time databases since it may lead to unpredictability. The *strict 2-phase-locking protocol* extends the basic one by minimizing the shrinking phase to a certain point of time. All locks are released together at the end of the transaction and no lock can be released before. This additional restriction avoids the possibility of a cascading rollback.

Both, the basic and the strict 2-phase-locking protocol are not deadlock-free. The third version is called conservative 2-phase-locking protocol and prevents from deadlocks by changing the growing phase. A transaction following this protocol has to acquire all needed locks before it begins execution by predeclaring the accessed objects. If just one object cannot be locked, it does not lock any object and waits until all items are available. Due to this fact, the growing phase is minimized to a certain point of time.

By combining the strict and the conservative 2-phase-protocol, deadlocks as well as cascading rollbacks are avoided.

2.4 Distributed Database Systems

Distributed database systems are allocated at several nodes and offer an increased availability and fault tolerance. Since information can be accessed locally the quality of service can be improved in terms of efficiency. A single database system also means a single point of failure whereas a distributed database system may handle a fault of one node.
Additional effort has to be taken to ensure the ACID-properties, especially consistency, in such a system. Concurrency does not depend on local transitions only but also on every transaction at every node. An often used criterion is one-copy serializability which ensures atomic execution of a transaction. The result of concurrent execution of transactions must be the same as one serial schedule running on one node.
There are two main approaches differing in terms of consistency and predictability: immediate and eventual consistency.

2.4.1 Immediate Consistency

Immediate consistency ensures that all nodes are fully mutually consistent at any time. This means that changes are made visible either at all nodes or not at all by using a pessimistic replication protocol. This high degree of consistency is payed by lowered predictability because the local execution time of a transaction depends on other nodes.
Immediate consistency protocols are divided into different categories [HHB96]:

Read One Write All (ROWA): With ROWA, a read operation is done locally on a single copy whereas a write is done to all replicas. The concurrency control has to ensure that its execution is equivalent to a serial execution. So, an update is made visible at either all nodes or none at all.

ROWA-Available: The original ROWA protocol is extended by allowing nodes to fail. Hence, updates are made to all available nodes in an atomic operation. A failed node has to recover before it joins again due to missed updates which may lead to mutual inconsistency.

Primary Copy: The nodes are separated into one primary copy and backups. A transaction writing an item is allowed to commit only after the primary and all backups

have updated the value whereas a read operation is executed only at the primary copy. When the primary copy fails a new one is selected. The main problem is to distinguish a failing node from slow network communication because only one node is allowed to be the primary at the same time.

Quorum Consensus (QC) or Voting: In the protocols just described, read operations are done on only one node whereas writes are done at least to a major part of the nodes. But since a write operation has to be done on all non-failing nodes, a network failure can hold the whole transaction. Quorum Consensus only needs to write to a major subset of all replicas, a so called *write quorum*. A read operation returns always the latest value since a read quorum has to overlap the write quorum. This protocol tolerates node failures as well as communication failures.

2.4.2 Eventual Consistency

In some environments like a lossy network, frequent node failures or large number of updates, immediate consistency with its inherent pessimistic replication protocol is not suitable. A distributed database using eventual consistency allows temporary inconsistencies in the global database state by executing and committing a transaction locally at one node. There are different degrees of consistency [Mat02]:

- Internal consistency: data within a node is consistent.
- External consistency: data of a database is a consistent representation of the state of the environment.
- Mutual consistency: database replicas are consistent with each other. For replicated databases with serialization as correctness criterion, replicas are always fully mutual consistent whereas with eventual consistency the databases eventually reach this state of consistency.

The execution of a transaction does not depend on any remote node since the propagation of updates takes place after the local commit which is called optimistic replication. This can be seen as a trade-off between consistency and availability/predictability where consistency

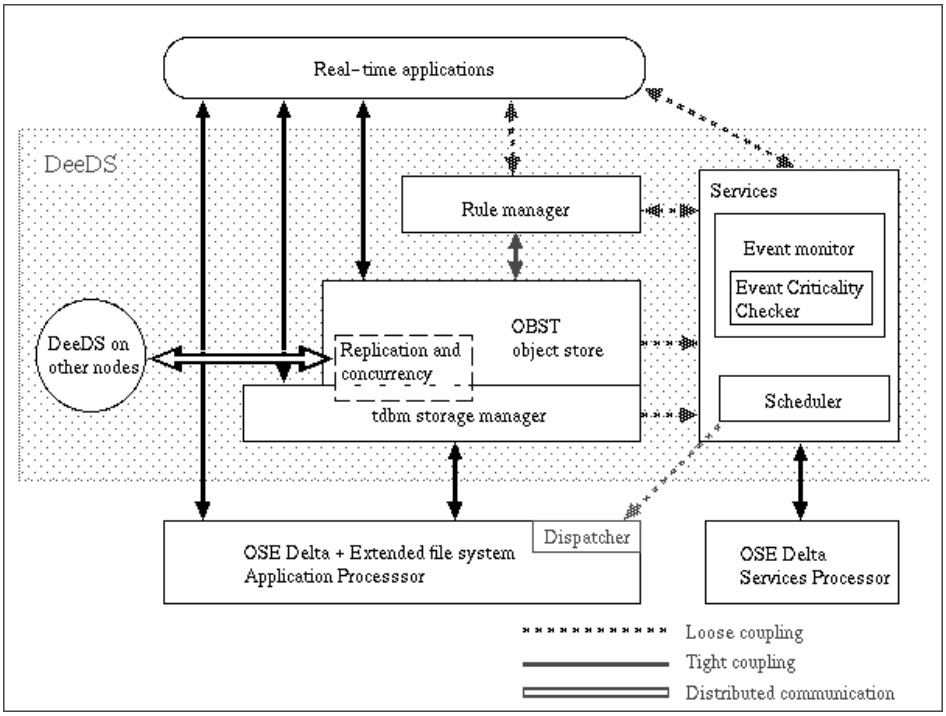

Figure 2.2: DeeDS architecture

constraints are lowered to raise availability. But eventual consistency also means that the system must converge to a globally consistent state. Conflict detection is needed to identify violation of the one-copy serializability criterion. Whenever a conflict is detected, it must be resolved in a deterministic way to reach a mutually consistent database state.

2.5 DeeDS

In complex real-time systems, a large storage capacity is often required. The Distributed activE rEal-time Database System (DeeDS) developed by the DRTS research group at the University of Skövde is one attempt to design a distributed database system to handle these large amount of data in real-time [And96]. DeeDS is an event-triggered database system, which uses dynamic scheduling of transactions [Lun97]. The architecture of DeeDS is shown in Figure 2.2.

The key features of DeeDS are [Mat02]:

- Main memory residency. To reduce unpredictable disk access, there is no persistent storage on disk.

- Optimistic and full replication is used that is necessary for real-time properties at each local node.
- Recovery and fault tolerance are supported by node replication. Failed nodes may be timely recovered form identical node replicas.
- Active functionality with rules that have time constraints.

2.5.1 DOI And TDBM

DOI the DeeDS Operating Interface is a layer that is added between the DeeDS Database and the underlying operating system. With DOI it is possible to make DeeDS operating system independent. In the current state, DOI supports POSIX compliant operating systems, like Linux or Unix, and the real-time operating system OSE Delta from ENEA. To allow this, DOI takes care of:

- Process handling: create, destroy and start processes
- Virtual node handling calls: process entry points, process instantiation and initialization
- Interprocess communication handling: send, forward and receive messages
- Process synchronization and handling: semaphores
- Memory management calls: allocate and free memory
- Calls supporting distribution: finding processes on other nodes
- Miscellaneous calls: timing information and getting information about a physical node

Tdbm (DBM with Transactions) is a transaction processing datastore with a dbm-like interface. It provides the following [Eri98]:

- Nested transactions
- Volatile and persistent databases
- Support for very large data items

Tdbm has a three layer architecture (Brachmann & Neufeld, 1992 [BN92]): item layer, page layer and transaction layer.

Item Layer: The item layer deals with key/value pairs in a page.

Page Layer: This layer is responsible for the page management and the allocation of physical pages. Tdbm allows locking of objects only on page level not on object level.

Transaction Layer: The transaction layer provides nested transactions. It is also concerned with locating the correct version of a page for a transaction, concurrency control and transaction recovery processing.

So tdbm is the core of the database processing in DeeDS.

2.5.2 Enhanced Version Vector Algorithm

DeeDS uses eventual consistency for replication. These protocols use conflict detection and conflict resolution mechanisms to resolve conflicts to reach global consistency. The conflict detection mechanism used in the DeeDS project is the enhanced version vector algorithm which is described in Lundström [Lun97]. This Algorithm is based on the version vector replication algorithm designed by Parker and Ramos (1982) [PR82]. Version vectors are defined in the following way:

Definition 2.5 *A version vector for a file A is a sequence of n pairs (S_i, V_i), where n is the number of sites at which A is stored. V_i contains the number of updates made to A at site S_i.*

For example a distributed systems consists of three nodes (node 1, node 2 and node 3) and two files, A and B. On each file a Version Vector is attached that is used to indicate if a node has updated a file. Now node 1 is updating A and node 2 is updating B. This will result in the following version vectors: A = <node 1:1, node 2:0, node 3:0> and B = <node 1:0, node 2:0, node 3:1>. In the DeeDS system this situation is mapped to objects, so each object is stored with an additional version vector, which indicates updates of an object.
A conflict is detected when neither of two version vectors, V and V', dominates the other.

Definition 2.6 *A version vector V dominates another version vector V', if $V_i >= V_i'$ for every item i in the vector.*

But the version vector algorithm detects only write-write conflicts [HHB96]. Hence, it is not suitable for a multifile environment, like a database. Parker and Ramos (1982) describe an algorithm that uses version vectors to detect multifile conflicts. This is the basis for the enhanced version vector algorithm described by Lundström.
It is not sufficient to solve just write-write conflicts, but it is also necessary to solve read-write conflicts and read-write cycles. These conflicts and how to detect them is described in detail later in this section. For detection of conflicts a Log Filter is used, which is defined in Lundström (1997, page 61) as follows:

Definition 2.7 *A Log Filter LF contains sets of database objects, where each set $S = obj_1$ ... obj_n in which an object either is a single object or an object set, depending on the granularity of version vectors.*

Each S in LF is represented by version vector sequences. These version vector sequences consist of version vectors. With version vectors it is possible to store the updates a transactions made. This information about the transaction is sent to the other nodes of the DeeDS system in update messages. An update message has the following format:

$$Update(\{readset\}, [Version\ vector\ for\ each\ object\ in\ the\ read\ set], \{writeset\})$$

Since the read set contains the write set, this information is also contained in the update message. Only a write operation on an object increases its version vector. For example in distributed database system that exists of three nodes, a transaction on the first node is executed that reads the values A, B and C and writes the value C. The update message that is sent to the other nodes looks like this:

$$Update(\{A, B, C\}, [< 0, 0, 0 >, < 0, 0, 0 >, < 1, 0, 0 >], \{C\})$$

If this update has been integrated on all nodes and a transaction that reads and writes the values B and C is executed on node 3, the update message looks like this:

$$Update(\{B, C\}, [< 0, 0, 1 >, < 1, 0, 1 >], \{B, C\})$$

After this update has been integrated on the other nodes the Log Filter contains the following version vector sequences:

$$LF = \{\{A, B, C\}, [< 0, 0, 0 >, < 0, 0, 0 >, < 1, 0, 0 >], [..., < 0, 0, 1 >, < 1, 0, 1 >]\}$$

... represents a null vector. A null vector is used to represent objects which have not been accessed by a transaction.
The algorithm described in Lundström (1997, page 62) gives a formal way of how to build up Log Filters and how to use them for conflict detection:

1. Initially, $LF = \emptyset$

2. Upon commit of a transaction with the set of objects in the read-set $S = obj_1$ ... obj_n do the following:

   (a) if S is contained in some set $S' = obj_1$ ... obj_n in the LF already, attach the version vector sequence { ... $< v_1^i >$ ... $< v_n^i >$ ... $< v_m^i >$ ... } to S' where $< v_i^j >$ is the version vector for object obj_i^j (the value j is contained in the object ID (OID) for the object), and the ... indicates that null vectors should be used.

   (b) If S is not already contained in LF , incorporate S into LF using a UNION-FIND algorithm [PR82]. The UNION(S_{obj}, S_T, S_T) operation take two sets S_{obj}, S_T,merge them to a single set S_T and then delete the two original sets. The FIND(obj) operation finds the extent of obj. The algorithm looks as follows:

   (S_T) := $\emptyset$
   for obj in S do
   begin
   S_{obj}:= FIND(obj)
   if ($S_{obj} = \emptyset$) then ($S_{obj} = obj$) –It is a new object
   UNION(S_{obj}, S_T, S_T)
   end

3. After an update sequence has been incorporated, the sequence should be checked for

conflict by executing the following:

$S :=$ FIND(obj)
if incompatible(S)
then if $S($*WRITE-SET* $) \supseteq S_{logfilter}($*WRITE-SET*$)$
    return (WW conflict)
    return (RW conflict)
else return(OK)

The purpose of the algorithm is to gather the objects, that have been accessed or updated by the same transaction, in sets. With this method, it is possible to see which objects that have been updated are in conflict. According to the algorithm, this is done by browsing the sets and finding a position where the new version vector sequence fits in. It is necessary to compare each sequence of the current Log Filter with the new sequence. Two sequences are compared by comparing the version vectors with the same OID. One version vector has to dominate the other version vector, otherwise a write-write conflict is detected. For example node 1 and node 2 update the same object A at the same time, the following update messages are created:

$$node1: \ Update(\{A\}, [<1,0>], \{A\})$$

$$node2: \ Update(\{A\}, [<0,1>], \{A\})$$

When the update message arrives on the other node, this node tries to insert the version vector sequence in its Log Filter. But the version vectors <1,0> and <0,1> do not dominate each other and a write-write conflict is detected.
After one domination of a version vector has been detected, the whole version vector sequence must dominate the other version vector sequence. If that is not the case, a read-write conflict is detected. Dominance of version vector sequences are defined as ([Lun97]):

Definition 2.8 *A sequence of version vectors V_1 ... V_n dominates another sequence W_1 ... W_n, if every version vector V_i dominates the corresponding vector W_i in the other sequence.*

A version vector sequence is inserted in the Log Filter after the sequences it dominates.
It is also possible that read-write cycles occur in a distributed database. Assume a system

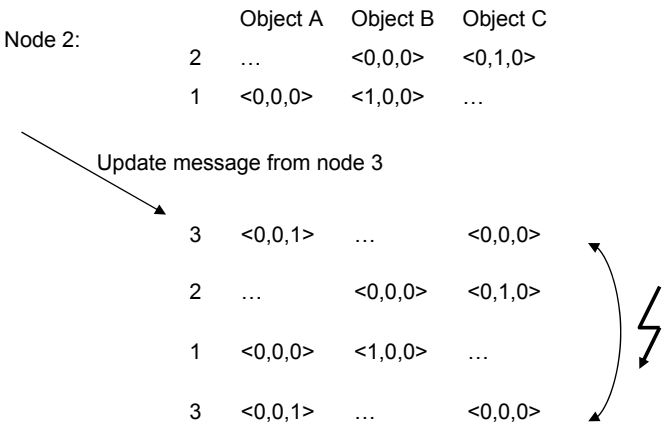

Figure 2.3: Read-write cycle

with three nodes (1,2,3). Node 1 reads object A and updates object B. Before the update message of this transaction is integrated on the other nodes, a transaction on node 2 reads object B and updates object C. And then a transaction on node 3 reads object C and update object A. The following version vector sequences are created:

$$1 \quad \{< 0,0,0 >, < 1,0,0 >, ...\}$$

$$2 \quad \{..., < 0,0,0 >, < 0,1,0 >\}$$

$$3 \quad \{< 0,0,1 >, ..., < 0,0,0 >\}$$

The read-write cycle is detected for example on node 2 (the read-write cycle is sooner or later detected on any node) in the following way (see also Figure 2.3): node 2 has received the update from node 1 and has finished its transaction, the version vector sequences in the Log Filter of node 2 are ordered. Version vector sequence 1 is ordered after the version vector sequence 2, because version vector sequence 1 dominates version vector sequence 2. This can be seen at the version vector of object B. Here version vector sequence 1 dominates and since null version vectors are not observed (these objects where not accessed by the associated transaction) version vector sequence 1 has to be inserted after version vector sequence 2 in the Log Filter. Now the update message of node 3 arrives at node 2. The algorithm tries to find a position for the version vector sequence 3. Version vector sequence 3 dominates version vector sequence 1 and has to be ordered after 1. But version vector sequence 3 is

also dominated by version vector sequence 2. This is not possible because version vector sequence 1 is already inserted after version vector sequence 2. Hence, no place for version vector sequence 3 can be found and a read-write cycle is detected. When a read-write cycle is detected, the 1-copy-serializability is no longer guaranteed until the conflict is resolved.

Chapter 3

Problem

This chapter gives a motivation why we address the problem of bounded delay replication, followed by a description of the problem. In order to narrow the issue of this dissertation, a description of the specific goals, which are separated into theory, implementation and validation, are expressed below.

3.1 Motivation

There are several replication protocols, like two-phase commit or ROWA [HHB96], that use immediate consistency to reach consistency in distributed databases. But for a large number of nodes synchronization of communication is necessary and network overhead increases. Hence, the duration of a transaction increases. The problems in this case are low availability, due to the need for synchronization, and unpredictability, caused by blocking of transactions. For distributed real-time databases however, availability and predictability are more important than immediate consistency.

If eventual consistency can be ensured and applications are tolerant of temporary inconsistencies, eventual consistency (see section 2.4.2) is a suitable correctness criterion for distributed real-time databases. Transactions are allowed to introduce global inconsistencies by committing of a local transaction, but the replication protocol ensures that replicas eventually become mutually consistent. Thus, the availability of a distributed real-time database is increased.

But in distributed real-time environments it is necessary that an update is propagated to

and integrated on all other nodes within certain time bounds. For example, an alarm signal of a machine must be replicated in bounded time in order to avoid damages. It is necessary for some transactions to propagate their changes in predictable time. With bounded delay replication added to as soon as possible replication (ASAP replication), it is possible to have the advantages of eventual consistency but also to ensure that values of transactions are propagated within bounded time.

3.2 Problem Description

As described above, bounded delay replication may be required for certain transactions. The replication process starts after the commit of a local transaction, which can specify a time bound needed to integrate the updates on all other nodes. To meet this deadline, it is necessary that every phase of replication is predictable.

So, propagation, communication between the nodes, and integration must be predictable. On the local node, after the commit of the local transaction, the message which contains the values of the local transaction must be created and passed to the network card adapter within a certain time. This part is called propagation. It is necessary for bounded delay replication that the message cannot be blocked by an update message of a transaction that is not demanding bounded delay replication. From the network adapter the update message needs to be delivered to all remote nodes in bounded time. When the update message arrives on the remote node, conflict detection and possibly conflict resolution must be performed. Conflict detection and conflict resolution are needed because it is possible that a local transaction or another remote transaction has updated the values an integration transaction intends to update. Then conflict resolution must be performed to ensure a consistent database state. This described conflict is a write-write conflict. It is also possible to have a read-write conflict, which is harder to detect and to solve. Conflict detection and conflict resolution must be done in bounded time.

In order to guarantee timeliness in bounded delay replication, it is very important that the integration process is not interrupted by an unpredictable process and that it does not depend on anything which is not predictable.

3.3 Objectives

3.3.1 Assumptions

We base our theory on the work done by Lundström [Lun97] and Eriksson [Eri98]. We extend their theory with additional assumptions. These assumptions are either general or concern one of the three subparts propagation, integration or communication. With the assumptions, it should be possible to determine an upper bound on the time needed for the whole replication process. Based on this, an admission controller can decide whether a request for the replication time of a transaction can be accepted or not. But not only assumptions that concern times are necessary, it is also necessary to make assumptions that guarantee predictability of all subparts. So all subparts involved in the replication process have to be covered in our theory, to guarantee the predictability of the whole process. Conflict resolution, which is part of integration, is a complex problem. There is no simple conflict resolution possible that may support all types of situations. Rather conflict resolution policies need to be adapted individually for each type of system, because the resolution of a conflict may cause cascading or sequential conflicts. However it is regarded as out of scope for this work.

3.3.2 Implementation

The first step of the implementation is to extend the existing software design of Eriksson [Eri98]. This design already shows partly how it is possible to integrate bounded delay replication during the development of DeeDS. One demand for the software design is to be as modularized as possible to allow easy changes and extensions to replication module. The extended software design is the basis of the implementation of bounded time replication. In the implementation it is again possible to identify the three parts propagation, communication and integration.

Propagation: For the propagation it is necessary to introduce the new bounded delay replication method in addition to the existing ASAP replication. It must be guaranteed that bounded delay replication messages are handled with higher priority than ASAP replication messages. So it should be possible to determine a worst-case execution time

of the propagation message.

Communication: Lundström (1997) states that:

> "To ensure timeliness in a distributed real-time system, real-time communication is a fundamental requirement to be able to guarantee an upper bound on response time of remote requests."

A real-time communication protocol is regarded as out of scope of this work, but several solutions for real-time networks can be found in Appendix B. When all the traffic in the network is known, it is possible to establish a real-time network with a standard switch. This can be done by removing the unpredictability of the back-off algorithm which is used in the Ethernet protocol when collisions occur. With the knowledge about all traffic in the network, it is possible to avoid collisions.

Integration: The integration has almost the same requirements as the propagation process. Again the existing ASAP method has to be complemented by a additional bounded delay replication part with a higher priority, in order to ensure predictability. In particular, conflict detection and conflict resolution (outside the scope of this work) have to be done in predictable time. For conflict detection the Log Filter must have a bounded size so that it can be searched in predictable time.

3.3.3 Validation and Verification

In the validation part of the work, the correctness of the theory and of the implemented source code for DeeDS has to be shown.
The theory is embedded in a complete theory about the database. But without having complete knowledge of the requirements of the database it is hard to prove the generalizability of the theory about bounded delay replication.
There are two parts that have to be verified to show the correctness of the source code: functionality and timeliness. On one hand, the functionality of the code must be tested. This is done by defining, running, and then evaluating test cases. These test cases must first show that the new code does not effect the existing code. So, regression-testing has to be done. After that it is necessary to test the new bounded delay replication part. On the

other hand the timeliness of the new code must be shown. Since real-time communication is regarded as out of scope of this work it is not possible to test the whole bounded delay replication process at once. The propagation and the integration part must therefore be tested separately. For the propagation part it must be shown that a large amount of ASAP replication traffic does not affect bounded delay replication. This is also necessary for the integration part. Additionally it is necessary to show that no parts of the program code are unpredictable. Otherwise, timeliness cannot be guaranteed.

Chapter 4

Bounded Delay Replication

In order to implement bounded delay replication for DeeDS, the existing theory needs to be extended to meet the requirements of a distributed real-time database. The extended theory, the resulting Software Design, the subsequent implementation for DeeDS and the tests of the implementation are described in this chapter.

4.1 Principles For Bounded Delay Replication

To assure predictability for real-time systems, it is necessary to define the condictions of the environment in which the system should work. Therefore, assumptions are needed to precise the scope of the problem. This makes it possible to build an appropriate model. The assumptions make it also feasible to set up the load and fault hypotheses, which are necessary for real-time systems (see section 2.1). Assumptions which are needed for bounded delay replication are explicitly motivated and separated into general assumptions and assumptions that are only needed in a special context such as propagation, integration or the communication.

4.1.1 General Assumptions

Assumption 4.1 *Every part that is involved in bounded delay replication must be predictable, in order to guarantee a bounded time for the entire replication process.*

This is the most important assumption for our work in order to guarantee bounded delay replication. It is absolutely necessary to know the worst case execution time of every single

module that is involved in the replication process. Thereby it is possible to give an upper bound for the replication process. If there is only one part that is not predictable, the whole system may fail because there is no worst case execution time anymore. But these execution times for the different parts depend on several other facts. For example, the time a CPU needs to execute a single operation, or how long it takes to transmit a message over the network.

Assumption 4.2 *Bounded Delay Replication Time = Propagation Process Time + Network Communication Delay + Integration Process Time*

This assumption shows the three main parts which are involved in the replication process. Following the "Divide and Conquer" paradigm, the abstract term *bounded delay replication* is divided into three subproblems, namely to find an upper bound on the time required for each of propagation, communication, and integration. These subproblems have to be investigated individually. The propagation process time is the time needed to create and send out an update message after the local commit of a transaction. The network delay covers the time for the transmission of the update message between two distributed nodes. The whole integration process time contains conflict detection, conflict resolution and integration of the update.

Assumption 4.3 *Strict conservative 2-phase-locking protocol (see 2.3.3) is needed for transactions.*

There are many concurrent transactions running on one node. To ensure the ACID properties (see section 2.3.2) of these transactions, strict conservative 2-phase-locking protocol (see section 2.3.3) is used to guarantee serializability. But these are not the only properties that must be guaranteed. For bounded delay replication, the time constraints also have to be followed. An ordinary 2-phase-locking protocol does not prevent deadlocks. Therefore, a resource preclaiming mechanism is used in the strict conservative 2-phase-locking protocol. This means that a transaction claims all locks, that it needs during the transaction, at its beginning instead of locking an object just before a read/write operation. By using strict 2-phase-locking protocol, a transaction is blocked at most once.

Assumption 4.4 *A minimum transaction interarrival time is required.*

To guarantee timeliness based on the capacity of the resources involved, it is necessary to determine a minimum interarrival time for transactions. This time must be at least as long as the time the admission control needs to check whether a transaction, regardless of its replication type, is allowed to enter the system or not. If such a bound is not specified, it is possible for an unlimited number of transactions to queue up before entering the system and so the execution time of each transaction can grow endlessly. So the minimum interarrival time is required to specify a load hypothesis.

4.1.2 Propagation

To get a predictable propagation process time, it is necessary to define the maximum number and the maximum size of update messages.

Assumption 4.5 *One local transaction creates at most one update message.*

Instead of using distributed transactions, DeeDS replicates data updates to replicas . A transaction running on one node will not run on other nodes to update that replica. Instead of running a transaction on each replica, the new values of data objects that were updated during a transaction, are propagated. For the purpose of conflict detection, both the context of the transaction and its current values are also stored in the update message. This is done by Version Vectors (see section 2.5.2) and the concept of two object sets: a read set and a write set. The read set contains all data objects that were read during the transaction, whereas the write set, which is a subset of the read set, contains all objects that were written during the update. Regardless of how many updates are executed during the transaction, there is only one update message, which includes the information about the original transaction that is necessary for the integration on any replica. If a transaction is only reading the database, there will be no update message because no data object has been changed. Query transactions for example do not cause any replication activity.

4.1.3 Integration

Assumption 4.6 *There can only be a maximum number of arriving transactions per time period.*

According to the load hypothesis (see section 2.1), a hard real-time system can only handle a limited rate of load due to time constraints. Since the integration is done by transactions as well, these integration transactions must also be taken into account when determining the transaction load for a node. We use the term *time period* as an specified amount of time to divide the continuous time into units. Within this unit, a fixed number of local transactions and integration transactions can be ensured to meet their deadlines. The time period starts at the beginning of the first transaction. This means that the available time per time period is consumed by local transactions, propagation of update messages, and integration transactions. Local transactions are initiated by the application running on the node whereas integration transactions result from received update messages from other nodes. Due to this fact, the number of messages for a node has to be bounded, which can be done in the following way: As there can only be a limited number of local transactions per time period due to the load hypothesis on real-time databases, there can only be a fixed number of update messages per node (see assumption 4.5). A real-time system can only handle a certain peak load due to its time constraints. This means that the number of local transactions, as one part of the load, has to be bounded as well. This leads to a maximum number of arriving transactions per time period. As stated in assumption 4.5, every transaction causes at most one update message. This update message leads to one integration transaction. Resulting from that fact, the maximum number of integration transactions per time period depends on the sum of all local transactions on any remote node. This sum is bounded because every node has a maximum number of arriving local transactions per time period according to the load hypothesis. Additionally, a maximum transaction execution time for local transactions is needed. The available time per time period decreases by every execution of a local transaction. As there should be enough remaining time for the integration transactions, there also has to be a bound on the total transaction execution time. These facts lead to the following assumption.

Assumption 4.7 *The latest entry points for local transactions have to be defined in each time period.*

Every local transaction started in time period TS_i has to commit and be propagated before the end of this time period. A local transaction can only be admitted if there is enough time

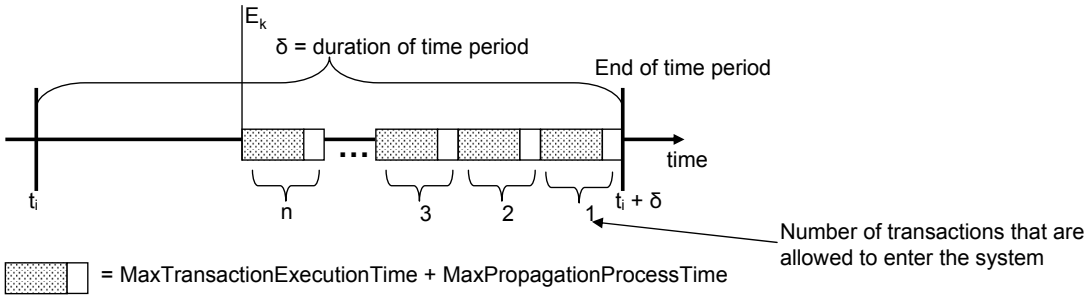

Figure 4.1: Integration Process

for its execution and propagation within the current time period. Consider the worst case where all local transactions write to the same data objects. In this case, the transactions have to be serialized. Since there is no concurrency between these conflicting transactions, the execution and propagation times have to be added up. We formalize this in the following formula (also shown in Figure 4.1):
Let us assume that n local transactions should execute in time period TS_i, which starts at t_i and lasts δ time units. The latest entry point E_k for transaction T_k is:

$$E_k = t_i + \delta - [(n-(k-1)) * MaxTransactionExecutionTime + MaxPropagationProcessTime]$$

The end of the current time period is equal to $t_i + \delta$ since it has started at t_i and lasts δ. All transactions, which have not been started yet (in our case: *n-(k-1)*), have to be executed and propagated in the remaining time. In the worst case, they all need the maximum transaction execution time and write on the same data objects. Therefore, no concurrency between those transactions is possible and since the propagation process has a lower priority than the local transactions, these times have to be summed up.

Assumption 4.8 *TimePeriod* $\geq$ *(NumberOfArrivingTransactionsPerTimePeriod* $*$ *MaximumTransactionExecutionTime) + MaximumPropagationProcessTime + MaximumIntegrationProcessTime*

As explained in assumption 4.6, there are two different kinds of transactions running on a local node: local transactions and integration transactions. There is a maximum number of (local) transactions arriving per time period that have to be executed and propagated during this time period and also a maximum number of arriving updates that have to be

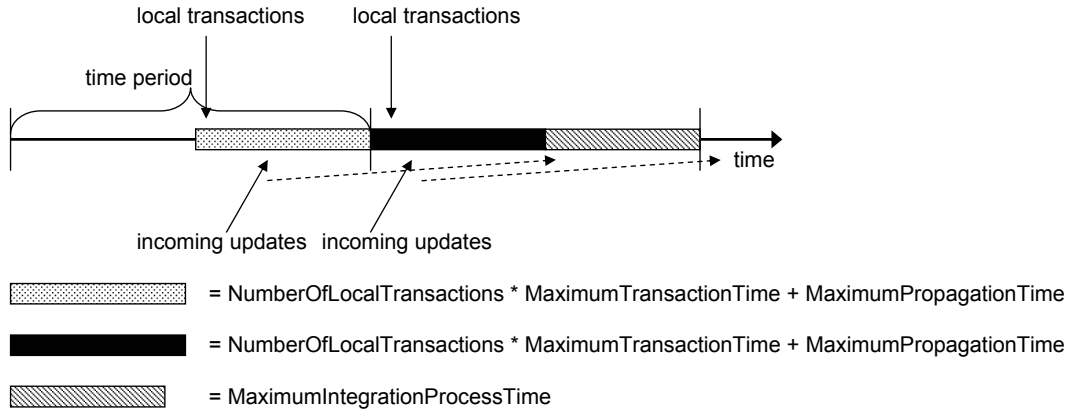

Figure 4.2: Integration Process

integrated by integration transactions. So the available time per time period has to be shared between the local transactions, their propagation and integration processes for all incoming updates. Since execution of local transactions has highest priority it is necessary to reserve time for propagation and integration. By assumption 4.7 and due to the lower priority of the integration process it is possible that update messages arriving towards the end of a time period cannot be integrated in this time period. Therefore, they have to be integrated in the next time period to ensure a predictable integration time as shown in Figure 4.2. With this result it is possible to say the following:

Assumption 4.9 *The integration of updates is done within two time periods (compare Figure 4.3).*

In order to have a bounded time for the integration process, it is necessary to have a bounded number of integration transactions per time period. A local transaction on a remote node creates at most one update message (see 4.5). As the number of local transactions per time period is limited, there is a bounded number of incoming update messages on each node. Let r be the maximum number of updates that a node propagates in time period δ. Every node of the n nodes in the network must then be able to handle $(n-1)*r$ received updates per time period. With this deterministic assertion and the following assumptions, it is possible to guarantee bounded delay integration.

Assumption 4.10 *A transaction is only allowed to access a limited number of objects.*

For the determination of a maximum transaction execution time, the number of objects that a transaction accesses must be limited. Due to this restriction, the maximum length of an

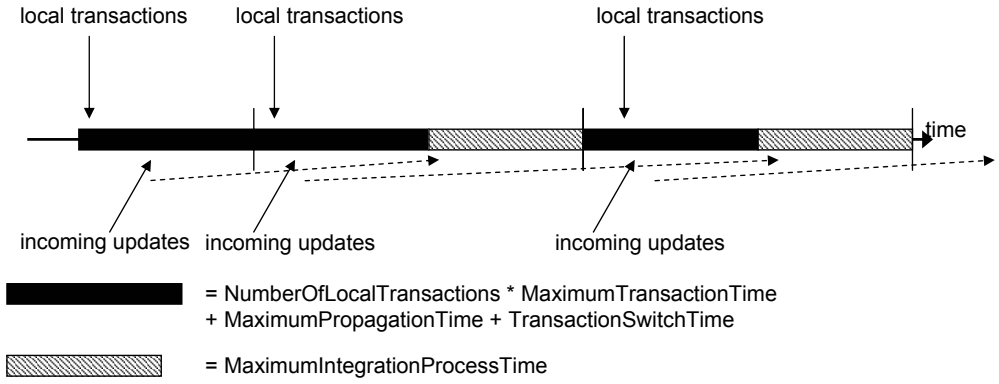

Figure 4.3: Integration Process

update message and also of an entry in the Log Filter is automatically determined. Since an update message contains information about the objects the transaction has accessed, this information is used for creating a Log Filter entry and for conflict detection. Conflict detection has to be done in bounded time, so it is not possible to have a Log Filter entry of unbounded size. Furthermore, this assumption determines as one factor the upper time for an integration transaction and thus this assumption is necessary for bounded delay replication.

Log Filter

Assumption 4.11 *The Log Filter must have a limited number of entries.*

The Log Filter is searched during conflict detection. As conflict detection has to be a deterministic process for bounded replication [Lun97], the search in the Log Filter has to be deterministic as well. For this reason, the number of entries in the Log Filter has to be bounded. This means, that the arrival of new entries and the deletion of obsolete entries must happen at the same rate. Every entry in the Log Filter which is older than *maximum length of read-write cycles * maximum replication time* is removed. Thereby, the number of entries can be limited.

4.1.4 Network

Assumption 4.12 *The update messages are transmitted in bounded time.*

The transmission time of update messages on the network has to be bounded to ensure a bounded delay replication. This property can be ensured by a real-time Ethernet protocol.

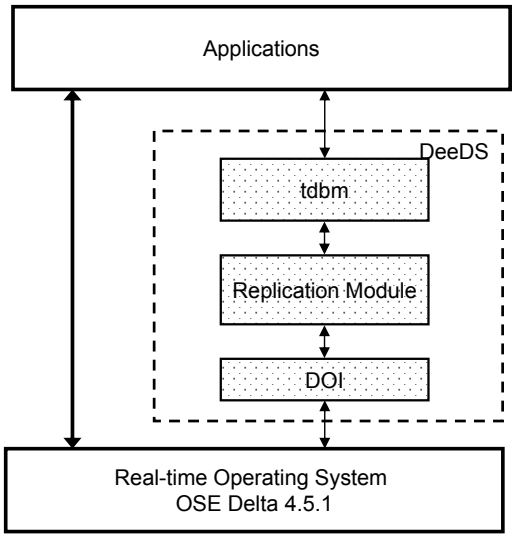

Figure 4.4: The Replication Module in the existing system

We have investigated real-time Ethernet protocols that are suitable for our system. The results of this investigation can be found in appendix B. The integration of a real-time Ethernet protocol into the system is left for future work.

4.2 Software Design

We based our design on the design presented by Eriksson [Eri98]. Figure 4.4 shows how the replication module interacts with the existing system. In the replication module, all data that is accessed during a local transaction is logged. This log is used for the creation of update messages which are sent out to remote nodes. The replication module is also responsible for integration of updates received from other nodes. The DOI (DeeDS Operating System Interface) is used to interact with the operating system, in this case OSE Delta 4.5.1.
We have decided to split the replication module into several submodules. This is shown in Figure 4.5. Most of the modules can be found in Erikssons work ([Eri98]), but we made slight changes in the design compared to Eriksson ([Eri98]). The main modules in the replication module which are explained in more detail below are: the Logger, the VersionVector Handler (VVHandler), the Log Filter (ASAP/bounded), the Propagator (ASAP/bounded) and the Integrator (ASAP/bounded). ASAP/bounded means that the specific module depends on the replication type ASAP replication or bounded delay replication. The Logger,

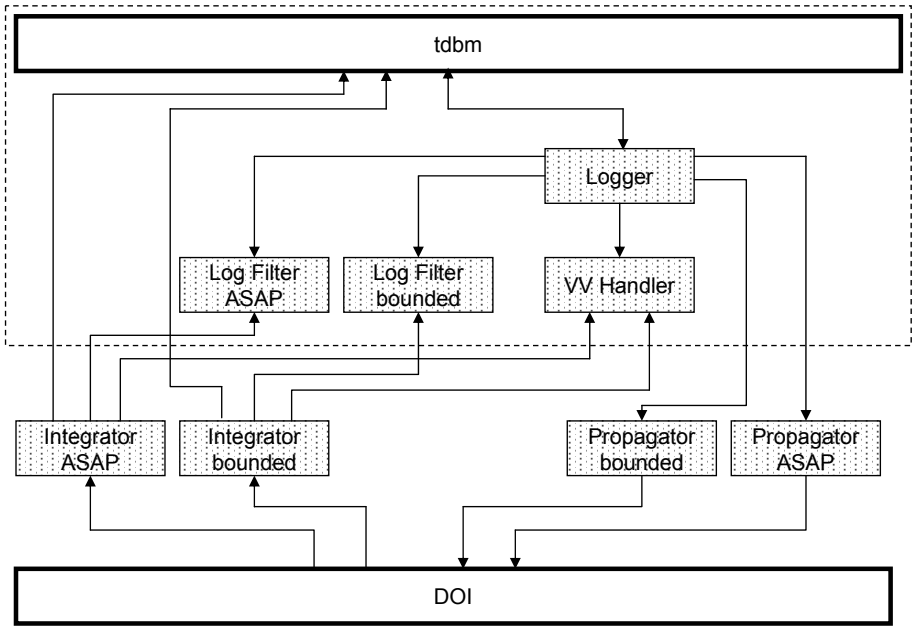

Figure 4.5: Replication Module

the VersionVector Handler and the Log Filter (ASAP/bounded) form the inner part of the system. During the execution of a local transaction the modules in the inner part can only be accessed by the processes needed for execution of the local transaction. This is especially important for the consistency of the Log Filter (ASAP/bounded). Without locking it would be possible that the integration process (ASAP/bounded) integrates a new update without finding any conflicts, but the currently running local transaction updates the same value as the integrated update. In this scenario, a conflict is not detected and the local update is lost, leading to an inconsistent database state. The inconsistency occurs among different nodes, since the update message was sent out, but the node which has sent the update does not contain this value anymore.

4.2.1 Logger

The Logger logs every local transaction running on the database. Since it records every data object that is read or written during a transaction. The resulting log contains the read and the write set of the corresponding transaction. The Logger determines the transaction type (read only or write transaction), it appends Version Vectors to data objects and it also creates

a log, which the Propagator (ASAP/bounded) uses in its update message. The information contained in the log is needed for conflict detection both on remote nodes and on the local node. Thus, this information is also inserted in the local Log Filter (ASAP/bounded).

4.2.2 Version Vector Handler

The Version Vector Handler (VVHandler) deals with all operations that are needed for processing Version Vectors. Version Vectors are grouped in Version Vector Sequences, which in turn are used within update messages and in the Log Filter (ASAP/bounded) for conflict detection. The Version Vector Handler module offers functions to insert Version Vectors into a dedicated sequence at a specific position. The result of these functions is a Version Vector Sequence, which is used in an update message. There are functions used during the creation of update messages as well as functions used in the process of conflict detection. The VVHandler can also compare two Version Vector Sequences to establish their dominance relationship.

4.2.3 Log Filter (ASAP/bounded)

The Log Filter (ASAP/bounded) represents the history of committed transactions and is used to detect conflicts. The module offers functions to insert new entries and delete obsolete ones. Changes to the Log Filter (ASAP/bounded) are done either by the Logger for updates made by local transactions or by the conflict handling module for updates resulting from transactions on remote nodes.

The Log Filter is separated into two units, one for each consistency class i.e. one unit for bounded delay replication and one for ASAP replication. The Log Filter (ASAP/bounded) is searched during the replication process, which is a real-time task in case of bounded delay replication. For this reason, the number of entries in the bounded delay Log Filter unit is limited to bound the search time. All entries older than *maximum length of read-write cycles * maximum replication time* are removed because after this amount of time, every update which could lead to a potential conflict has already been integrated on every node. Obviously, this means that the maximum length of read-write cycles must be determined. Our hypothesis is that the length of a read-write cycle depends on the number of objects in

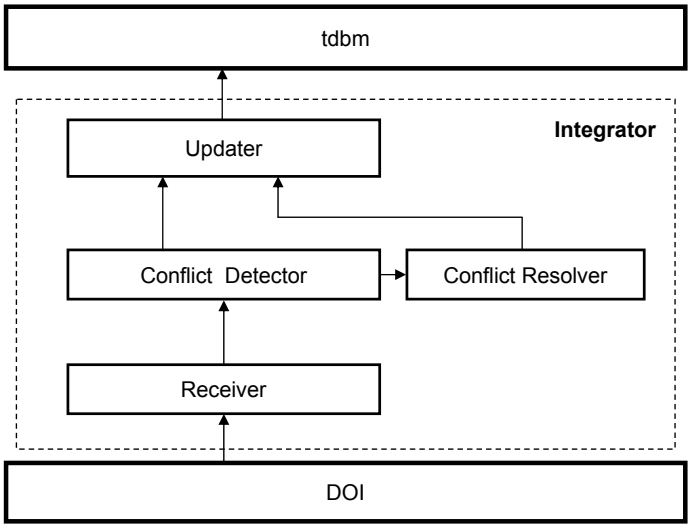

Figure 4.6: Integrator

the database, but a proof for this is left open as future work. For ASAP replication, there is no such limitation. Therefore, the entries in its unit are not deleted within a guaranteed time. A possible solution for the ASAP part could be a replication with acknowledgements.

4.2.4 Propagator (ASAP/bounded)

The Propagator (ASAP/bounded) is a unit that updates messages to remote nodes and it runs independently from other processes. The bounded Propagator is sending its update messages to the bounded Integrator and the ASAP Propagator is sending its updates to the ASAP Integrator on the remote node. When the local database is changed by a transaction, all changes result in an update message which has to be sent to remote nodes. This update message is delivered to the Propagator (ASAP/bounded), depending on the replication type. The bounded Propagator has a higher priority than the ASAP Propagator, so that no ASAP update message can block a bounded replication update message. These messages are sent to Integration processes (ASAP or bounded) on all remote nodes, depending on the replication type, via `doi_send()`, which is a function call in DOI.

4.2.5 Integrator (ASAP/bounded)

The Integrator (ASAP/bounded) (see Figure 4.6) processes incoming update messages and integrates them into the local database while resolving any conflicts. This is done in several steps where the functionality of various submodules is used. These modules are the Receiver, the Conflict Detector, the Conflict Resolver and the Updater. The Conflict Detector and the Conflict Resolver are described below. The Integrator module has the same functionality for both replication types, the difference is that the Conflict Detector accesses the corresponding Log Filter, so that the Conflict Detector returns in bounded time for bounded delay replication. Like the Propagator modules the bounded Integrator has a higher priority than the ASAP Integrator.

The integration process consists of two main tasks. On the one hand the update of the local database, on the other hand the update of the Log Filter (ASAP/bounded). These two parts of the system, the local database and the Log Filter (ASAP/bounded), have to be consistent to avoid system failures. The Integrator receives an update message from DOI, this is handled by the Receiver module. This message contains information about written and read data objects. These objects are going to be locked by the tdbm module. The update message is checked for conflicts by the Conflict Detector which locks the Log Filter (ASAP/bounded). In case of a conflict, the Conflict Resolver has to manage it. The result is written into the Log Filter (ASAP/bounded). As the final step, the updates are written into the local database during an integration transaction created by the Updater module and the locks in the database are released.

Conflict Detector

The Conflict Detector checks the update message for potential conflicts with the database replica. It locks the corresponding unit of the Log Filter (ASAP/bounded) so that its state is not changed during conflict detection. The Conflict Detector uses the Version Vector Handler to compare every Version Vector sequence in the Log Filter (ASAP/bounded) with the one given by the update message. It checks for write-write as well as for read-write conflicts. After the conflict detection, the lock on the Log Filter (ASAP/bounded) is released and the Integrator (ASAP/bounded) either calls the Conflict Resolver or inserts the given Version

Vector sequence in the Log Filter (ASAP/bounded) at the correct position depending on whether a conflict was detected or not.

Conflict Resolver

The Conflict Resolver is called by the Conflict Detector in case of a conflict and is expected to return with the conflict resolved deterministically in bounded time. This module is regarded as future work.

4.2.6 Modifications On Existing Software Design

This section focuses on the changes that were made to ensure bounded delay replication in coexistence with ASAP replication.

Modifications On Propagator & Integrator

Both the Propagator and Integrator handle update messages. There are two different types of update messages corresponding to the two replication classes: bounded delay replication and ASAP. Bounded delay replication update messages have a higher priority than ASAP update messages due to their time constraints. A message of higher priority is handled first even if it was received later than a message of lower priority. To achieve this functionality, there is a process for each replication type with the same functionality but with a higher priority for the bounded Propagator and Integrator so that bounded delay replication cannot be blocked by ASAP replication.
This design decision makes it possible to extend the system with further consistency classes. The introduction of another consistency class can be done by an additional Propagator and Integrator with an appropriate priority and a corresponding Log Filter.

Modifications On Log Filter

These modifications apply for both Log Filter modules: ASAP Log Filter and bounded delay Log Filter. The Log Filter is a central component for conflict detection. For this reason, the Log Filter and the local database necessarily have to be consistent. If the update of the database and the Log Filter is not done as an atomic operation, potential conflicts can be

missed which might lead to a system failure. We illustrate this with an example: assume that a local transaction holds a number of locks on the database while an update message is received that updates the same data objects. As the data objects are locked by the local transaction, the integration of the update message has to wait for the locks being released. If the update of the Log Filter is not done before or as the local transaction commits, the integration of the update message can be processed immediately after the commit since all locks are released. At this moment, the Log Filter has not been updated yet, which means that no conflict is detected and the values in the local database are updated again resulting in a lost update. Consistency between the local database and the Log Filter is achieved by having the Log Filter update be part of the transaction. The update of the Log Filter is done immediately after the changes in the database, but before the transaction commits.
On this point, our software design differs from the one in [Eri98]. In that design, the update of the Log Filter is done after the commit of the local transaction. The update message is send both to the remote nodes and the local Integrator. As described above, this could lead to a failure, since in the time gap between the commit and the update of the Log Filter an update message of a remote transaction which is currently waiting in a queue can be integrated.
In our design, the Logger writes directly into the Log Filter before the transaction commits. As the local transaction uses the strict conservative two-phase-locking protocol, there is no need for conflict detection because any operation which might lead to a conflict is avoided by locks.

Modifications On Integration Of Updates

In databases with eventual consistency, the handling of conflicts between simultaneous updates from more than one node is a very central and important task. As the conflict resolution is out of scope of our work, the whole processing of conflicts is split into modules which can be individually exchanged. In the design of [Eri98] the conflict detection, conflict resolution and the integration of updates are done within one module. We have decided to define several submodules in the Integrator, to simplify extension of the system with an independent module for each task. With this approach, it is possible to use a simple Conflict Resolver at the beginning which does not limit further work.

We also use one instance of the Integrator for each replication type to make it easy to set priorities and bounded replication can never be blocked by ASAP replication.

4.3 Implementation

The new implementation is based on the existing DeeDS which is written in C and C++. The former implementation supported only as soon as possible (ASAP) as replication type without any conflict detection and use of the Log Filter. The missing parts concerning ASAP replication and modifications to allow use of two replication types, ASAP and bounded delay, were added.

The interprocess communication in DeeDS is done by messages. Each process has a message queue used as a buffer for messages sent by other processes. Each message is sent with a signal indicating the type of the message. A process can either receive a message sent with a specified signal or just any kind of message. A standard receive *doi_receive()* checks if there is already an appropriate message in the queue and if not, waits for the next suitable one. The *doi_receiveWTmo()* function in contrast waits only a specified time and returns NULL in case of no appropriate message.

In the following subsections, the implementation of all changed or extended modules is described.

4.3.1 Logger

The Logger logs every database transaction executed by an application and is called by the tdbm database module, as seen in Figure 4.5. A transaction initiated by an application is called a *normal transaction*. There are two other transaction types: *logging transactions* and *update transactions*. A logging transaction is related to a normal transaction and logs all operations performed by the normal transaction. In other words, the logging transaction is executed by the Logger and finally used to create a log message for the propagation of updates. Whenever a transaction begins, reads/writes a value or commits, an appropriate function of the Logger is called as follows:

begin: When a transaction is started by the application, the *drd_createLog()* function is auto-

matically called. This function initializes the *logInfo* variable of the normal transaction and starts the logging transaction. The *logInfo* is a structure containing information about the transaction type, the replication type (either bounded delay or ASAP) and about the logging transaction.

fetch/store: Whenever a read or write operation is done by the transaction, *drd_storeLogEntry* is called. It creates a log entry consisting of four parts:

- Header: The header contains information about the operation type (read/write), the key, the value and the pathname of the associated database file.
- Key: The key is used as an unique identifier to locate the object on the database.
- Version Vector: This field contains the Version Vector of the logged object.
- Value: This is the actual value. In case of a read operation, this field is not interesting and therefore not set.

The *logInfo* of the transaction is also adjusted by setting the actual log size which is the sum of all log entries associated to this transaction, and increasing the number of updated objects by one.

commit: As a transaction commits, it calls two functions of the Logger, one before the actual commit of the database transaction and one after. *drd_fetchLog()* is called before the commit and creates a final log by merging all log entries created during the transaction. This final log has a dynamic size depending on the number of log entries and their sizes. For this reason, it contains the number of log entries at the beginning followed by the actual log entries. The changes are also written at the end of the local Log Filter. The Log Filter is used for conflict detection during the integration of remote updates and represents in some way the history of the database. Therefore, the Log Filter and the database always have to be mutually consistent. The Log Filter can also be accessed by the Integrator which makes semaphores for mutual exclusion necessary. Conflict detection is not necessary for insertion of local updates since the updates are made on the database of the local node following the serializability criteria. After that, the Log Filter and the database are locally consistent and the commit can be done.

After the commit, the *drd_finishLogging()* function commits the logging transaction and sends the final log to the appropriate Propagator (bounded/ASAP) process. The log is sent with a different message type depending on the replication type of the transaction, so that the Propagator can distinguish them.

4.3.2 Propagator

The Propagator module sends the transaction log in an update message to all replicas to distribute the updates after a local commit. It consists of two processes: a Propagator for ASAP and a Propagator for bounded delay replication. Since the propagation of bounded delay update messages must not be delayed by ASAP update messages, the Propagator for bounded delay replication has higher priority.
During initialization each Propagator process receives a list of processes on remote nodes to which it should send update messages. The Propagator for ASAP replication receives a list of Integrator processes for ASAP replication/integration and vice versa the Propagator for bounded delay replication receives a list of Integrator processes for bounded delay replication/integration.
Both Propagator processes are running in an infinite loop checking for received update messages. This is done by calling the function *doi_receive()*. The Propagator first checks if the received update message corresponds to the correct replication type. If so, the update message is sent to the suitable processes on the replicas.

4.3.3 Integrator

The Integrator receives update messages from other nodes and integrates them into the database to reach a globally mutual consistent database state. It uses conflict detection and conflict resolution for this purpose. The Integrator consists of two processes: an *Integrator_bounded* and an *Integrator_asap* process.
The two Integrator processes receive messages from the appropriate propagation process. The time needed for the integration process includes time for conflict detection and possibly conflict resolution. The *Integrator_asap* has lower priority and is therefore always preempted by the *Integrator_bounded* when there is a bounded delay update to be integrated. Hence,

the integration of the bounded delay replication class is not delayed by the existence of the ASAP replication class. Here after, Integrator is used as a name for both the Integrator_asap and the Integrator_bounded processes.

First the Integrator checks the replication type of the update message and if the message has the correct type, it locks the objects on the database that are going to be updated. After that, the Log Filter is also locked for mutual exclusion. The Integrator parses the update message and creates Version Vector Sequences (VVSeq, see Listing 4.1) derived from the read/write operations done on the replica. A VVSeq logically consists of several but at least one Version Vector. The VVSeq is implemented as a structure that consists of a pointer to the first Version Vector of this sequence, a pointer to the next VVSeq (used for the Log Filter) and a timestamp needed for the pruning of the Log Filter.

Listing 4.1: Version Vector Sequence

```
typedef struct drd_VVSequence_struct {
  drd_VerVec *head; /* the first Version Vector of this sequence */
  struct drd_VVSequence_struct *next; /* link to the following VVSeq */
  doi_Time time; /* timestamp, used for pruning of Log Filter */
} drd_VVSequence;
```

After the creation of the new VVSeq, the Log Filter is checked for obsolete entries in case of a bounded delay integration. The subsequent conflict detection and, in case of a conflict, conflict resolution determine the position at which the new VVSeq should be inserted and the values that are used to update the database. As a result of this, the VVSeq is first inserted into the Log Filter and then the updates are written to the database. An update transaction now executes which is not logged as a normal transaction. Since the Log Filter is locked during the whole process, the update of the Log Filter and the database is an atomic operation. The state of the Log Filter or the values that are going to be inserted can therefore not be changed by a concurrent non-integration transaction. At the moment, this situation can still lead to a deadlock when the concurrent transaction accesses values residing on the same page that are updated in a different order.For example, assume that the Integrator locks page A and wants to lock page B which is already locked by a local transaction and this local transaction in turn wants to lock page A. This deadlock situation can be avoided by the strict conservative 2-phase-locking protocol. Unfortunately, this is not implemented in tdbm yet and regarded as future work. For our test cases we assume that there is no

concurrent local transaction accessing the same objects and pages, respectively. Finally, the lock on the Log Filter is released and the integration process is finished.

4.3.4 Log Filter

The Log Filter represents a logical serialization of the update history of the local database. It is implemented as a structure which contains the number of Version Vector Sequences (VVSeqs) currently residing in it and a pointer to the first as well as to the last sequence (see Listing 4.2). The VVSeqs also have a pointer to their next VVSeq. In other words, the Log Filter is a linked list of VVSeqs used for conflict detection.

Listing 4.2: Log Filter

```
typedef struct drd_Log Filter_struct {
  int noSequences; /* the current number of VVSeqs in the Log Filter */
  struct drd_VVSequence_struct *first;
  struct drd_VVSequence_struct *last;
} drd_Log Filter;
```

The Log Filter is accessed by either the Logger in case of a local update or by the Integrator / Conflict Detection in case of a remote update. Since both execute read and write operations on this structure, their accesses must be mutual exclusive. There are functions to insert a new VVSeq: *drd_insertSequence()* (see Listing 4.3), *drd_addSequenceFirst()*, *drd_addSequenceLast()*

The first function inserts the VVSeq *newSeq* after a specified VVSeq *oldSeq* which already resides in the Log Filter.

Listing 4.3: Insert a sequence to the Log Filter

```
void drd_insertSequence(drd_Log Filter *lf,
                        drd_VVSequence *newSeq,
                        drd_VVSequence *oldSeq)
{
  newSeq->next = oldSeq->next; /* newSeq points to the next VVSeq of oldSeq */
  oldSeq->next = newSeq; /*oldSeq points to newSeq */
  lf->noSequences++;     /* increase number of VVSeq residing in Log Filter */
}
```

The other functions are similar and insert the VVSeq either at the beginning or at the end of the Log Filter. It is important to insert the new VVSeq at the correct position, because the Log Filter is totally ordered to represent a logical serialization of the history of updates. This means that a VVSeq at position k dominates all k-1 VVSeqs before it and is dominated

by all VVSeqs that follow it.

There are two Log Filters, one for ASAP and one for bounded delay updates. This is necessary because in order to ensure timeliness of bounded delay replication, the transactions have to be restricted in some way, i.e. a transaction with bounded delay replication is not allowed to access an object which is written by a transaction with ASAP replication. ASAP replication gives no time guarantees at all, therefore a transaction with bounded delay replication cannot rely on this data. For this reason, every object in the database has to have exactly one replication type. This is explained in more detailed in the Results chapter 5.

The Log Filter for bounded delay replication has to be of bounded size to ensure that conflict detection/resolution finishes in bounded time. Therefore, the pruning method is run every time a new VVSeq is inserted (local/remote update). During the pruning, the Log Filter is checked for obsolete entries. Every VVSeq has a timestamp and any VVSeq older than a certain time can be removed. After a time a bounded delay update is integrated on every remote node and, therefore, stable. This means that no simple conflict can occur anymore related to this VVSeq. This conclusion is also part of our results chapter and explained in 5.2.

4.3.5 Conflict Detection

Conflict Detection is a central module in the implementation. The replication protocol uses optimistic replication, which may lead to conflicting updates. Since the protocol must ensure eventual consistency, i.e., the replicas must eventually converge to a mutually consistent state, conflicts have to be detected and solved.

Conflict detection uses the new VVSeq created by the Integrator and the Log Filter to check if there are conflicts between a remote update that is going to be integrated and updates that have already been made on the database. The module uses the function *drd_compVVSeq()* (see Listing 4.4) which compares two VVSeqs using the method *drd_compVV()*. The method *drd_compVV()* compares two Version Vectors and checks whether either Version Vector dominates or if there is a conflict. This is done for every Version Vector in the new VVSeq having a corresponding Version Vector (a Version Vector with the same OID) in the old VVSeq. If the two Version Vectors indicate a conflict, the method returns with DRD_WW_CONFLICT

signalling a write-write conflict between the two corresponding updates. Otherwise the number of dominations of a Version Vector either of the new VVSeq or of the old VVSeq is stored in the integer variables *old*, *equal* and *ok*, respectively. *Old* is increased if a Version Vector of the old VVSeq dominates whereas *ok* is increased if the Version Vector of the new VVSeq dominates. *Equal* is increased if the two Version Vectors are equal.

Listing 4.4: Compare Version Vector sequences

```
drd_Rc drd_compVVSequence(drd_VVSequence *newSeq,
                          drd_VVSequence *oldSeq)
{
  drd_VerVec *tmpNew, *tmpOld;
  drd_Rc result;
  int old = 0;
  int ok = 0;
  int equal = 0;

  tempNew = newSeq->head; /* first VV of the new VVSeq */
  tempOld = oldSeq->head; /* first VV of the old VVSeq */

  /* while both VVSeq are not at their end */
  while(tmpNew!=NULL && tmpOld!=NULL) {
    if(tmpNew->oID == tmpOld->oID){
      /*corresponding VVs */
      result=drd_compVV(tmpNew,tmpOld);
      switch(result){
      case DRD_WW_CONFLICT:
        return result;
        break;
      case DRD_OK:
        ok++;
        break;
      case DRD_OLD:
        old++;
        break;
      case DRD_EQUAL:
        equal++;
        break;
      ...
      }
      tmpNew = tmpNew->next;
      tmpOld = tmpOld->next;
    }
    else if....
```

When all Version Vectors of the new VVSeq have been evaluated and there was no write/write conflict, the variables *ok*, *equal* and *old* are examined. If *ok* and *old* are bigger than zero, this is a read-write conflict since at least one vector of the new VVSeq dominates the old one and vice versa. In case of no conflict, information about which VVSeq dominates is returned.

This return value is used for the following conflict detection (see Listing 4.5). For each type of conflict (write/write, read/write, read/write-cycle), the Log Filter is searched. In the first phase write/write conflicts are detected. In the second phase, the Log Filter is searched for read/write conflicts and read/write-cycles. The following code detects any kind of conflict and is explained in the sequel:

Listing 4.5: Conflict Detection

```
*oldSeq = NULL; // insertion point for new VVSeq
int rwConflict = 0;
int rwCycle = 0;
...
tempNext = lf->first // tempNext is first VV of the Log Filter
do{
  rc = drd_compVVSequence(seq, tempNext);
  if((rc==DRD_OK)&&(rwConflict==0)&&(rwCycle==0)){
    /* seq dominates tempNext, oldSeq is NULL till seq is dominated first */
    if(*oldSeq==NULL){
      /* save last vvSeq as a possible insertion point for the new sequence */
      tempPredecessor = tempNext;
    }else{
      /* READ/WRITE-Cycle Error */
      rc=DRD_RW_CYCLE;
      rwCycle=1;
    }
  }else{
    if(*oldSeq==NULL && rc==DRD_OLD && rwCycle==0 && rwConflict==0){
      /* seq is dominated by tempNext, oldSeq is insertion point for VVSeq
       * in case of no conflict */
      *oldSeq=tempPredecessor;
    }
    if(rc==DRD_WW_CONFLICT){
      *oldSeq=tempNext;
    }
    if(rc==DRD_RW_CONFLICT && rwConflict==0){
      *oldSeq=tempNext;
      rwConflict=1;
    }
  }
  /* go to next vvSeq of log filter */
  tempNext=tempNext->next;
}while(tempNext!=NULL);
...
```

The conflict detection consists of a loop which compares any VVSeq in the Log Filter with the new VVSeq of the corresponding remote update. The return code of this function is used for further decisions if there is a conflict.

- If the return code is DRD_OK and no conflict has been detected yet, the new sequence

dominates the old one. This old VVSeq *tempNext* is stored in the variable *tempPredecessor* which represents the predecessor of *tempNext*. But only if the VVSeq *oldSeq* is still NULL. This means that the return code for every passed loop before was DRD_OK and the new VVSeq has not been dominated or conflicting yet. The Log Filter is totally ordered in a way that a VVSeq i that dominates VVSeq j is always sorted in after j. If *oldSeq* is not NULL anymore, the new VVSeq of the remote update has already been dominated by a VVSeq i in the Log Filter and dominates now a VVSeq j which in turn dominates i according to the ordering in the Log Filter. Therefore, no total order can be established and a read/write-cycle is returned.

- If the return code is not DRD_OK or a conflict has already been detected, the function works as follows:
  - If the return code is DRD_OLD and no conflict has been detected, the insertion point *oldSeq* is set to the prior VVSeq. This means that the new VVSeq of the remote update was dominated and should therefore be sorted in after the predecessor of the current VVSeq of the Log Filter. If *oldSeq* has already been set, nothing needs to be done.
  - If the return code is DRD_WW_CONFLICT, a write/write conflict was detected and *oldSeq* is set to the conflicting VVSeq to mark this position.
  - If the return code is DRD_RW_CONFLICT and no write/write conflict has been detected, the read/write conflict flag is set and oldSeq is set to the conflicting VVSeq to mark this position.

This loop ensures that all write/write-conflicts are detected followed by the detection of read/write conflicts and read/write-cycles.

4.3.6 Tdbm

The interface of tdbm (see section 2.5.1) has been changed at to fit an architecture with two replication types. The old interface still exists and is used for transactions with ASAP replication whereas the added interface is used for transactions with bounded delay replications. The internals of tdbm has not been changed.

Open

Each replication type owns a separate database segment which is implemented as a database file. Every object in this segment inherits the replication type. There are two tdbm functions to open a database file: *DbmBegin()* for objects with ASAP replication and *DbmBeginBounded()* for objects with bounded delay replication. Each database file can be opened multiple times. When a database file is opened for the first time, it is automatically created. When it is opened again, it is checked that a bounded delay segment is not opened by the function *DbmBegin()* and vice versa that an ASAP segment is not opened by the function *DbmBeginBounded()*, because a segment can only have one replication type, which cannot be changed.

Begin

There are two transaction types: ASAP and bounded delay. The replication type has to be declared at the beginning of the transaction because a transaction with bounded delay replication could be refused in an overload situation.
DbmBegin() starts a transaction with ASAP replication. *DbmBeginBounded()* tries to start a transaction with bounded delay replication but first checks if this transaction can be admitted or not. The Admission Control counts the transactions with bounded delay replication that have already been admitted within the current time period. If this number is less than the maximum allowed number per time period, the transaction is admitted.

Fetch/Store

A transaction with ASAP replication is allowed to read objects from the bounded delay database segment and to read/write objects from the ASAP segment. A transaction with bounded delay replication is only allowed to read/write objects of the bounded delay database segment and must not access any object in the ASAP segment. The reasons for this restriction are explained in the result chapter 5.2.

Store: The creation or update of an object is made with *DbmStore()*. This function checks the replication type of the current transaction and the replication type of the database

segment going to be accessed. If those are not the same type, the function aborts and returns an error code (*DRD_WRONG_REPLICATION_TYPE*).

Fetch: The current value of a database object is read with *DbmFetch()*. This function also checks the replication type of the current transaction and the replication type of the database segment. If the replication type of the transaction is bounded delay replication and the database segment owns the ASAP replication type, the function aborts and returns an error code. If the replication type of the transaction is ASAP replication and the database segment owns the bounded delay replication type, the read operation is still allowed but not logged by the *Logger*. Of course, read operations where both replication types match are allowed.

Commit

One of the biggest changes in the logical structure was made to the *DbmCommit()* function. In the former implementation, operations concerning the propagation such as finishing the log and updating the Log Filter were made after the actual commit of the transaction. This leads to a undesirable temporary inconsistency between the database and the Log Filter representing the history of the database (see section 4.2.6).
Completing the log and updating the Log Filter is now done before the commit. The update is inserted at the end of the Log Filter and needs no conflict detection since local serializability is ensured by the database. Therefore, this process is done in bounded time regardless of the replication type. After the commit of the transaction, the final log is sent to the Propagator starting the replication process.

4.4 Test Scenario

It is important to verify a developed system, especially for a hard real-time system. It is also important to test the system with several test cases in a test model. In this section the test environment and the test cases are described. The test cases are necessary for the investigation of the behaviour of the software system.

4.4.1 Test Environment

The test environment depends on the type of test. The functionality test can either be made on DeeDS running Linux, Unix or OSE Delta. For the timing test it is necessary to have a predictable clock and a predictable scheduler for the processes. Running Linux with the highest priority, on softkernel of OSE Delta, or on OSE Delta hard target gives the possibility for time measurements. The softkernel OSE Delta, which is then running on Sun Solaris, is in a sense predictable. The internal Softkernel clock ticks only when the Softkernel gets process time from the underlying operating system.
When using Linux, Unix or Softkernel it is possible to have several nodes on one machine, since the communication between the nodes is done in the same manner as interprocess communication.
OSE Delta is a real-time operating system created by ENEA for fault tolerant and safety embedded systems. Thus, special hardware is needed for tests with hard kernel OSE Delta. The test setup and the tests we wanted to run on OSE Delta are described in in the section Tests on OSE Delta.

4.4.2 Test Cases

At first it is important to show that the new code does not affect the existing program. So, all existing test cases should be still able to run and end with the same result as before. This means regression testing has to be done. After that it is necessary to test the functionality of the new code. Predictability and timing of bounded replication is also a important issue for testing. So, we define functionality tests as well as timing tests.
The following test cases are based on the test cases that were defined in [Eri98]. Also the segmentation into the three statements: *purpose, stimuli* and *expected outcome* are defined in [Eri98].

4.4.3 Functionality Tests

For all tests it must be checked that each replication class, ASAP replication and bounded replication, does not cause any error, when tests are performed for this class. When both

classes are tested together it must also be possible to see that bounded replication is favourly handled.

Local changes on one node

Test 1 - Store Entry

Purpose: To test that no values get corrupted by the Replication module.

Stimuli:

- A local node inserts a new entry in its database.
- The same local node reads the new entry.

Expected outcome: The output is equal to the input.

Test 2 - Update Entry

Purpose: To test that the Replication module does not cause the updated entries to become corrupted.

Stimuli:

- A local node updates an existing entry in its database.
- The same local node reads the new entry.

Expected outcome: The output is equal to the update entry.

Test 3 - Store Entry with Version Vector

Purpose: To test that the version vector which is created with a new entry is handled correctly.

Stimuli:

- A local node stores a new entry in an existing database.
- The same local node reads the newly stored entry along with its version vector.

Expected outcome: The output is equal to the input and a correct version vector should be shown.

Test 4 - Update Entry and Version Vector

Purpose: When an update is done on a local node the version vector of the object shall be increased.

Stimuli:

- A local node updates an existing entry in the database.
- The same local node reads the updated entry along with its version vector.

Expected outcome: The output is equal to the update entry. The version vector is correctly increased.

Test 5 - Check Propagate Message at sending node

Purpose: To test that a propagation message is build in a correct way related to the performed transaction.

Stimuli:

- A node stores a new Entry in the database.

Expected outcome: The propagation message should fit to the expected propagation message.

Test 6 - Check Propagate Message at receiving node

Purpose: The propagation message on the receiving node must match with the one on the sending node. Furthermore the propagation message shall be interpreted in the right way at the receiving node.

Stimuli:

- A node stores a new entry in the database.
- Another node receives this new entry as a propagation message.

Expected outcome: The received propagation message should be equal to the propagation message which was sent.

Test 7 - Replication (create new Entry) without conflicts

Purpose: After receiving a propagation message with a new entry, the entry shall be created in the remote database.

Stimuli:

- Node A creates a new entry on the database.
- Node B receives the propagation message.
- Node B processes the received message and inserts the value in its database.

Expected outcome: The entries in both nodes should be the same.

Test 8 - Replication (update existing Entry) without conflicts

Purpose: Update the entry in a remote database by using the propagation and replication mechanism.

Stimuli:

- Node A updates an existing entry in its database.
- Node B receives the updated with a propagation message.
- The object in the remote database on node B is updated with the information found in the propagation message.

Expected outcome: The entries in both nodes shall be the same.

2 and more nodes

All the following test shall be run with 2 and 3 nodes performing updates that consistency can be observed.

Test 9 - Replication more than one node

Purpose: All the nodes must be able to handle local transactions and update transactions at the same time.

Stimuli:

- Node A and B create or update entries in their database.
- The nodes receive the update messages.

Expected outcome: The Log Filter and the database entries on all the nodes shall be the same.

Test 10 a) - c) - Log Filter

Purpose: With this test the functionality of the Log Filter is tested.

Stimuli:

- Create update messages and local updates in the Log Filter that:
- a) the next update message is inserted in the first position.
- b) the next update message is inserted at a given position n.
- c) the next update message is inserted at the end of the Log Filter.

Expected outcome: The entries should be inserted in the right position in the Log Filter.

Test 11 - Bounded delay Log Filter

Purpose: After a given time the bounded delay Log Filter must be pruned.

Stimuli:

- Set the value for the pruning time of the Log Filter.
- Add several entries in the Log Filter.
- After the pruning time run a local transaction.

Expected outcome: The too old entries should be deleted.

Test 12 a) - c) - Conflict Detection

Purpose: Check whether all conflicts are detected.

Stimuli:

- Run transactions on the nodes that they cause:
- a) write-write conflicts
- b) read-write conflicts (read-write cycle between two nodes)
- c) read-write cycles (between three nodes)

Expected outcome: The conflicts shall be detected.

Test 13 - Admission Control

Purpose: To test whether the admission control rejects new transactions with bounded replication, when the maximum number of transactions with bounded replication is reached.

Stimuli:

- Create the maximum number of transactions per time period.
- Create a new transaction with bounded replication in the same time period.

Expected outcome: The new transaction should be rejected.

Test 14 a) and b) - Database access

Purpose: A transaction with ASAP replication is only allowed to read objects which are in the bounded replication class. Transactions with bounded replication are not allowed to

access objects which are in the ASAP consistency class.

Stimuli:

- a) Run a transaction with ASAP replication which wants to write an object in the bounded replication consistency class.
- b) Run a transaction with bounded replication which wants to access an object in the ASAP replication consistency class.

Expected outcome: A tdbm error message should be shown.

Test 15 - Performance test

Purpose: To test whether bounded replication messages are always handled with favour.

Stimuli:

- Run transactions with ASAP replication on high load.
- Run several transaction with bounded replication.

Expected outcome: The transaction with bounded replication shall be integrated before a ASAP transaction, which commits at the same time.

Test 16 - Long run test

Purpose: Simulate a "real" application.

Stimuli:

- Run transactions with mixed replication types, on high load with event showers.

Expected outcome: The system should be able to handle this load.

4.4.4 Timing Tests

Bounded delay replication aims to ensure timeliness for update messages. So all defined test cases above should have an upper time bound, as long as they concern bounded delay replication. It is also necessary to define additional test cases that show the timeliness of bounded delay replication. Since real-time communication and a global clock are not part of DeeDS at the moment, it is not possible to measure the whole replication process. For this reason, propagation and integration can only be measured separately. Since every part of bounded delay replication is predictable it should make no difference adding the times for propagation and integration instead of measuring both at a time. But this has to be tested as well and is left open until the whole DeeDS prototype is build. For the timing test, the following test cases are defined:

1. **Propagation process:** In this test case, the time which is needed for a bounded delay replication message from the commit of the transaction to the time point when the message is propagated is measured. As a first step, the time is measured when only transactions with bounded delay replication are running. The result time of this test is than compared to the result time when transactions with ASAP replication are running. As result of this test, it should be possible to see that the result time for bounded delay replication is the same in both cases.

2. **Integration process:** The time from the receipt of an update message from another node until the values are integrated in the database is measured. The time for the integration process can vary much depending on the conflict resolution. So an upper bound for the integration without conflicts and the worst case when conflicting with all given objects can be found during this test case.

3. **Long running test:** In the last test case, many transactions with both replication types are created over a long time. Due to the pruning of the bounded replication Log Filter it should be possible to see that the bounded replication messages are always integrated in bounded time. Whereas the integration time of ASAP updates grows, because the Log Filter the ASAP integration has to search grows with every received ASAP update message.

4.4.5 Tests On OSE Delta: Setup

To get really reliable results for our timing tests, a real-time operating system and a real-time network are necessary. OSE Delta 4.5.1 was the chosen real-time operating system for our tests. For the tests with OSE Delta 4.5.1 we got three Motorola MBA860 boards. These boards are equipped with a MAC860 Powerfully processor, 32 Bytes RAM, a PCMCIA interface and a 10Base Ethernet interface.
When the operating system, DeeDS, and an application, in our case the test cases, are running on a MBA860 board, it is possible do determine the exact time of every operation and then compare it with the expected outcome.
Since we know all the network traffic, our real-time network has to handle, it is not necessary to have a "real" real-time network. If we want to use a standard solution like Ethernet, we have to avoid collisions, because after a collisions has occurred the unpredictable back-off algorithm is used for determining the time when a node is allowed to send again. The avoidance of collisions is possible by the use of a standard switch and the restriction that each node is not allowed to send more packets than the switch and the full duplex connection are able to handle. If the nodes where allowed to send more than the switch is able to process collisions would not occur on the network, but the switch would discard the packets it is not able to process.
With this premises and the knowledge about the traffic it is possible to determine the number of Update messages our network is able to handle. Hence, we know how many transactions can be scheduled. The network can also be used to synchronize the clocks of each board, so a global time can be reached.
The setup for our tests on OSE Delta 4.5.1 is shown in figure 4.7.

4.4.6 Tests On OSE Delta: Description

It would be really interesting to run all the described timing tests also on the MBA860 boards and compare these results with the results of the Softkernel and Linux timing tests. But the different performances of the MBA860 boards and for example a 1 GHz Linux Server have to be kept in mind.
With the possibility of a having a global clock, the duration of the whole bounded replication

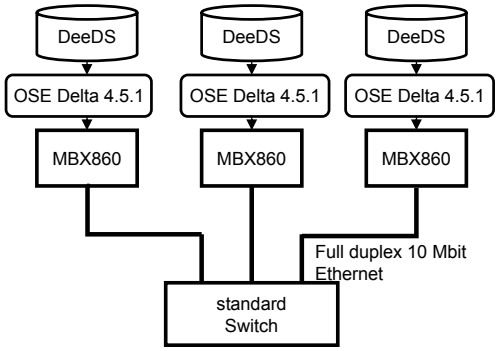

Figure 4.7: Hardware Setup

process can be measured. Thus, the upper bound of the bounded replication process can be found in a worst case test. Thereby, it can be seen whether the bounded replication process is really bounded.

Out of this it is possible to define the following additional test cases for OSE Delta 4.5.1:

1. **Timing tests:** Rerun all the described timing tests and compare the results. This can be done first with two nodes and after that with three nodes and then compare the resulting times for two and three nodes. One result might be that the single times for propagation and integration are increasing because of the higher number of arriving updates. This can then cause a higher load on each node. Finally with global time the whole bounded delay replication process can be measured, this can also be done with two and three nodes.

2. **Worst case test:** Having hardware and network performance values, it is possible to determine the bottleneck of the system and according to this bottleneck the worst case test can be designed. This bottleneck has to be identified for the use of two and for three nodes as well because the total number of updates can increase with additional nodes, when the network is not the bottleneck. By running this test, the worst case bounded delay replication time can be determined for two and for three nodes. This test should also be executed with only ASAP replication to see the difference to the bounded delay replication. For ASAP replication, the number of transactions is only restricted by the hardware and not by the network protocol, the propagation, or the integration. It should also be possible to run this test with more than three nodes.

3. **Real application test:** It might be interesting to talk with industry or other research

projects where DeeDS can be used with its different replication classes. With these new requirements, it can be seen whether DeeDS replication is designed and implemented sufficient for the use together with real applications.

In this test case it should be possible to integrate these new requirements, then simulate the application on the boards and see in which manner DeeDS is used by the application.

Chapter 5

Results

In the previous chapter we have described the method used to address the problem of bounded delay replication in a distributed real-time database. In this chapter we review this approach, by combining the theory and the implementation, stating the restrictions found during this project, analyzing the implementation and showing the results of our test cases.

5.1 Theory and Implementation

Out of our theory and implementation, it is possible to determine the time bound for bounded delay replication. But first, it is necessary to show that every component that is involved in bounded delay replication is predictable. Thus, we review the different assumptions and restrictions that ensure bounds for the processes. The following components have bounds for predictability reasons (an explanation for the restrictions, which are not already explained in the assumptions, is given in the next section):

- Transactions can only be blocked once during their execution (assumption 4.3).
- A maximum interarrival time for transactions is given (assumption 4.4).
- The number of bounded delay replication update messages is not higher than the number of allowed transactions per time period. A local transaction creates at most one update message (assumption 4.5).
- The number of arriving transactions per time period is limited (assumption 4.6).

- Latest entry points for transactions in time periods are defined (assumption 4.7).
- The Log Filter used for bounded delay replication, has a limited number of entries (assumption 4.11).
- Transactions with bounded delay replication cannot access the database segment of transactions with ASAP replication.
- Transaction with ASAP replication can only read from the bounded delay replication database segment and perform any operation in the ASAP database segment.
- Bounded delay replication has higher priority than ASAP replication.
- The bounded Propagator process has higher priority than the bounded Integrator process.

By the use of these limitations, assumptions and implementation it is possible to determine the time bound. In assumption 4.2, we defined the Bounded Delay Replication Time as Propagation Process Time + Network Communication Delay + Integration Process Time. The sum of these times represent the execution time which is needed for a single update message with bounded delay replication.
For the propagation of a single bounded delay update message, the process fetches the update message out of its message queue and sends it to the corresponding bounded Integrator. This means that the message has to be passed to the network card adapter. The time for a thread switch from the ASAP Propagator to the bounded delay Propagator must additionaly be added to the Propagation Process Time. The switching time and the time needed for the propagation process is the Propagation Process Time.
The bounded Integrator receives a bounded delay update message from a remote node and passes this message to the Conflict Detector. The Conflict Detector uses the bounded sized Log Filter for the conflict detection. The worst case is, if the conflict is detected in the last row or if no conflict is detected, because all version vector sequences have to be compared. When no conflict is detected, no conflict resolution is required. Otherwise, conflict resolution is called. Since conflict resolution is regarded as out of scope of this work we can assume that conflict resolution is done in bounded time. Finally when the conflict detection (CD) and

the conflict resolution (CR) is done, the updated values have to be written to the database. In the worst-case every object in the database has to be updated. We can formalize this as follows:

$$Integration\ Process\ Time\ =$$
$$ReceivingTime\ +\ MaxCDTime\ +\ MaxCRTime\ +\ MaxUpdateTime$$

By using assumption 4.8 and the limitations due to priorities, the worst case time for integration of an update and for propagation of local transactions that arrive during a time period can be determined. For example, assume that n is the number of nodes of the distributed database and r is the number of transactions arriving during a time period. In the worst case, every local transaction needs bounded delay replication. Thus, there are r bounded delay replication update messages per node in a time period. Local transaction have the highest priority in the system. Hence, the propagation can only take place after the execution of local transactions. The message queue of the Propagator contains r updates and all updates need the same PropagationTime. In the worst case, the MaxPropagationTime is:

$$MaxPropagationTime\ =\ r\ *\ PropagationTime$$

With the use of assumption 4.9, which is shown in figure 4.3 it is possible to determine the MaxIntegrationTime as:

$$MaxIntegrationTime\ =\ 2\ *\ TimePeriod$$

This MaxIntegrationTime contains both the time needed for integration of updates and the time during that the integration process has to wait due to local transactions or propagation. This fact is explained in assumption 4.9.
A maximum number of updates per time period arriving on a node can only be assured with a real-time network. The real-time network offers a maximum delay to the update message,

so we can determine the worst case bounded delay replication time as:

$$MaxReplicationTime = MaxPropagationTime + NetworkDelay + 2 * TimePeriod$$

Finally we can state that the time period which has to be defined by a later Admission Control depends on the number of objects in the database (to determine the worst case execution time of the local and integration transactions), the number of nodes (to determine the maximum number of updates), the length of the read-write cycle (to determine the worst case conflict detection time) and whatever values a conflict resolution might need.

5.2 Restrictions

The coexistence of two replication classes with different constraints concerning timeliness imposes some restrictions on the behavior of the transactions as well as on the implementation. As mentioned earlier in this dissertation, a transaction with a specified replication type is not allowed to read or write every object. The database is separated into two segments, one for each replication class. Every object resides in one segment and implicitly inherits the replication type of this segment. A transaction with bounded delay replication is only allowed to read/write an object of the corresponding bounded delay segment whereas a transaction with ASAP replication is allowed to read/write objects of the ASAP segment and additionally to read objects of the bounded delay segment. It is not allowed that a transaction with ASAP and a transaction with bounded delay replication write the same object illustrated by the following example (see Figure 5.1):
Assume a transaction T_1 with bounded delay replication on node N_1 and a transaction T_2 with ASAP replication on node N_2 both writing object A at the same time. The bounded delay update has time constraints and arrives at N_2 within a guaranteed time which leads to the detection of a write/write-conflict. The propagation of the ASAP update of T_2 has no time constraints and may be delayed for a long time i.e. by the propagation of other bounded delay updates. The bounded delay Log Filter of N_1 is pruned after a certain time and therefore the write operation of T_1 is not visible for the conflict detection anymore. Finally, when the ASAP update arrives at N_1 the conflict is missed which leads to a permanent

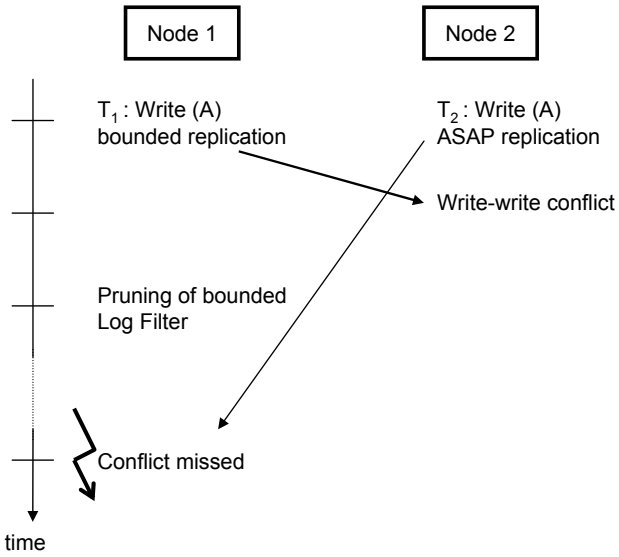

Figure 5.1: Restricted write access

inconsistent database.

A transaction with bounded delay replication is not allowed to read objects which reside in the ASAP segment and may be updated by a transaction with ASAP replication. Assume three objects A, B and C. A and B reside in the bounded delay database segment whereas C resides in the ASAP segment. Transactions T_1 and T_3 are transactions with bounded delay replication and T_2 is a transaction with ASAP replication. All three transactions are running concurrently on different nodes. T_1 reads A and writes B, T_2 reads B and writes C and T_3 reads C and writes A (see Figure 5.2). This is a read-write cycle which is not detected until a node receives all update messages of this cycle. Since the sending of the update message of T_2 can be delayed after the time when the Log Filters of Node 1 and Node 2 are pruned, this read-write cycle may never be detected on these two nodes.

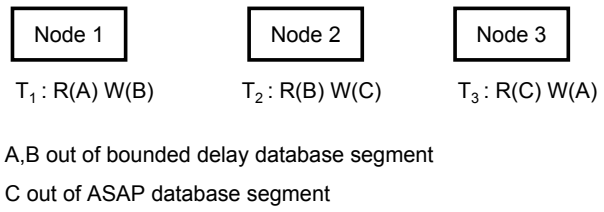

Figure 5.2: Restricted read access

Every replication type has its own Log Filter. The bounded delay replication needs to be

finished within bounded time. A part of the replication process is conflict detection that compares the new VVSeq with the VVSeqs residing in the Log Filter. This process has to be done in bounded time. Therefore, the number of entries in the bounded delay Log Filter has to be limited. Bounded delay replication ensures the integration of updates within bounded time. This means that an update cannot conflict with new updates after a fixed time. Therefore, its corresponding VVSeq can be removed from the Log Filter, which makes it possible to guarantee an upper bound on the number of entries. ASAP replication has no time constraints concerning the integration of updates. Hence, the VVSeq of the ASAP update cannot be pruned after a fixed time which makes it necessary to have a dedicated Log Filter for every replication class.

But there is another restriction concerning the pruning of the bounded delay Log Filter. Even if an integrated update can be regarded as stable after a certain time, there can be a conflict later on that is not detected before. This conflict is a read-write cycle which is either limited by the number of objects in the bounded delay segment (which may be very large) or manually limited by imposing a restriction to the maximum allowed length of read-write cycles. Therefore, an entry in the Log Filter cannot be removed till the time when no update closing this possible read-write cycle could arrive at this node anymore.

5.3 Implementation Review

There were two goals for the implementation. The first goal was to finalize the former DeeDS implementation in order to use a Log Filter and to do conflict detection. The second one was the implementation of a bounded delay replication class in coexistence with the already implemented ASAP replication. One issue of this implementation is to show that bounded delay replication is not delayed by any ASAP replication. This is done in two ways: reasoning about the implementation and testing. The following section explains why a bounded delay replication process cannot be delayed by any ASAP replication.

The replication consists of propagation of update messages and integration of updates on remote nodes. There is one Propagator for each replication type. The Propagator process for bounded delay replication has higher priority than the Propagator process for ASAP replication. The Propagator processes do not share any resources so they cannot block each

other. Therefore, the higher prioritized bounded delay Propagator always preempts a running ASAP Propagator and cannot be preempted by this one again. The propagation process is predictable since it only checks the message type and sends the update to the corresponding integration process. Since the number of bounded delay update messages per time period is bounded and the propagation process is predictable, the propagation time of each bounded delay update message is bounded as well.

The integration consists of one integration process for each replication type. The Integrator for bounded delay update messages has higher priority than the Integrator for ASAP update messages. The priority order of the processes involved in the replication is therefore in descending order: Propagator for bounded delay replication, Integrator for bounded delay replication, Propagator for ASAP replication and Integrator for ASAP replication. The integration of bounded delay updates accesses disjoint resources from the ones accessed by the integration of ASAP updates. They use different Log Filters and different database segments which means that the higher prioritized Integrator for bounded delay updates can not be blocked by the Integrator of ASAP updates but can preempt the ASAP Integrator at any time. The integration of bounded delay updates can therefore not be delayed neither by the integration nor by the propagation of ASAP update messages.

Bounded delay replication ensures integration of updates in bounded time. The Integrator first locks all objects in the database which are going to be updated. It can only be blocked by local transactions accessing the same database page (tdbm locks on page level) but this is a predictable delay according to our theory that allows only a limited number of local transactions per time period. Then it locks the bounded delay Log Filter followed by a pruning of it and finally by the conflict detection. Since the Log Filter is bounded in the number of its entries and each version vector sequence has a maximum number of version vectors, all this operations are done in bounded time. In case of a conflict, a bounded conflict resolution has to be done which is regarded as future work. At the end of the integration process, the updates are written into the database which is also finished in bounded time since every object is already locked and the number of objects to be updated is limited by the assumption that a transaction is only allowed to access a limited number of objects.(see Assumption 4.10)

5.4 Test Results

To run our test cases, described in section 4.4, we wrote several test programs that act like an application that uses DeeDS and especially the replication module. The following two sections show our results for the defined functionality and timing tests.

5.4.1 Functionality Tests

Test number	Transactions with asap replication	Transactions with bounded delay replication	Transactions with both replication types
Test 1	OK	OK	—
Test 2	OK	OK	—
Test 3	OK	OK	—
Test 4	OK	OK	—
Test 5	OK	OK	—
Test 6	OK	OK	—
Test 7	OK	OK	—
Test 8	OK	OK	—
Test 9	OK	OK	OK
Test 10 a)	OK	OK	—
Test 10 b)	OK	OK	—
Test 10 c)	OK	OK	—
Test 11	—	OK	—
Test 12 a)	OK	OK	—
Test 12 b)	OK	OK	—
Test 12 c)	OK	OK	—
Test 13	—	OK	—
Test 14 a)	OK	—	—
Test 14 b)	—	OK	—
Test 15	—	—	OK
Test 16	—	—	OK

Table 5.1: Results of the functionality tests

Table 5.1 shows the results of our functionality tests, which are described in section 4.4.3. Since most of the parts are used by ASAP replication and bounded delay replication, it was necessary to test the functionality of both replication types. When the execution of only one transaction was sufficient for a test, we ran the test for each replication type. Otherwise it was sufficient to test the replication module with both replication types in one test run,

because this shows implicitly that the test would also run with only one replication class. When a test was not necessary to run with a specified replication type, this is marked with "—" in the table. When the test ended with expected outcome, this is shown with OK in the table otherwise the test result is indicated with NOK that stands for not okay. That all tests ended with the expected outcome gives the indication that the functionality of the implementation is correct.

5.4.2 Timing Tests

We ran our timing test on a PC equipped with a Pentium III-500 CPU running Debian-Linux as OS with kernel version 2.2.20. Since we only measured the process time of the propagation and integration process isolated from each other, it was possible to have two nodes for the test on one machine. Only one node was doing local transactions so that the processes (application, bounded propagation and bounded integration) executed serialized due to the priorities.

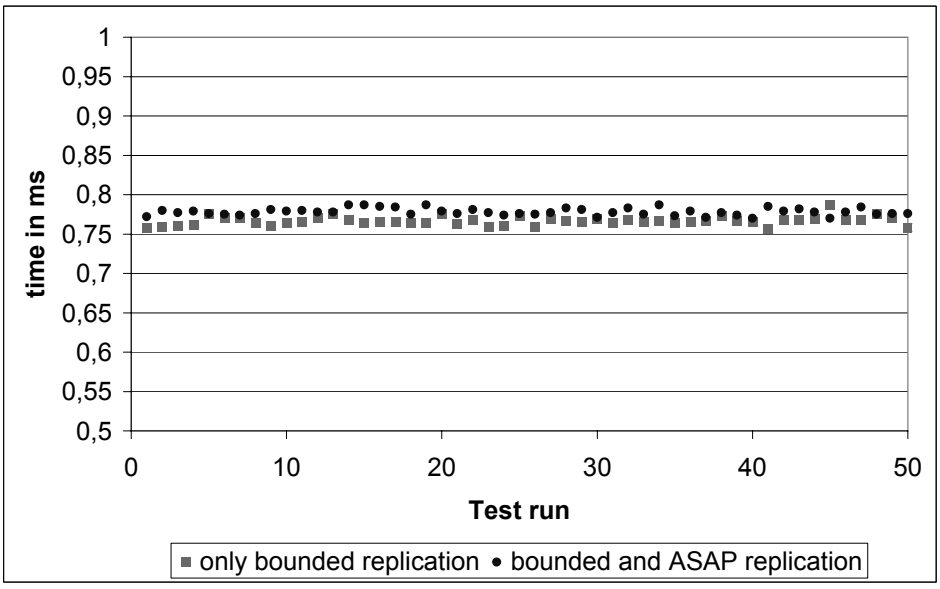

Figure 5.3: Timing test: Propagation process

For the first timing test, we run a transaction with bounded delay replication and measured the time for the propagation. We measured the time that was needed from the commit of the local transaction the the sending of the update message. This time indicates the propagation time. After executing the test only with transactions with bounded delay replication, we

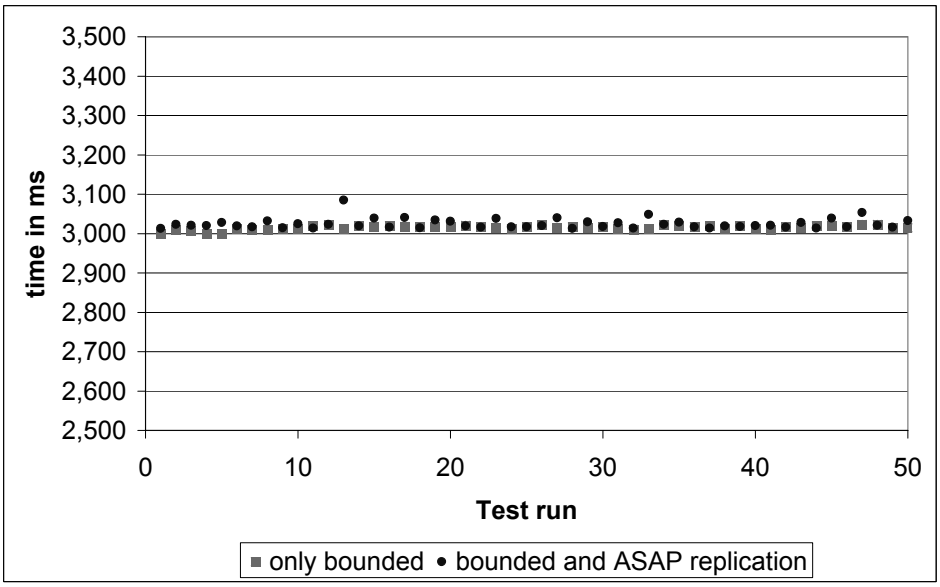

Figure 5.4: Timing test: Integration Process

measured the time again. But this time, transactions with ASAP replication were running, too. The results of these test are shown in figure 5.3. It can be seen that the difference between running ASAP and bounded delay replication and only bounded delay replication does not exceed 0,02 ms. In most cases the duration of ASAP and bounded delay replication is a bit longer than only bounded delay replication. An explanation for this is: when ASAP propagation is running and a bounded delay message has to be propagated, the propagation must do a context switch from Propagator ASAP to Propagator bounded. This is also bounded since this can happen only once.

In the second timing test we measured the duration of the bounded delay integration process with and without the ASAP integration process running. Here we began the measurement after the receipt of a message and finished the time measurements after the update has been integrated. The results of this test can be found in figure 5.4. Here it can be seen that most integration processes needed nearly the same time, but some of the processes when running ASAP integration concurrently needed longer for their execution. This can be explained in the following way: the page locking of tdbm has an higher priority than all integration processes. The higher priority of tdbm is reasonable because a local transaction should not be blocked by an integration transaction for unbounded long time. During the locking of a page, the ASAP integration process cannot be preempted by the bounded delay integration

process and so the bounded delay integration process has to wait until the ASAP integration process can be preempted.

In our long running test 10000 transactions without causing any conflicts have been executed. It was necessary to have such a number of transactions in order to see the influence of the Log Filter size on the integration time. The pruning time of the Log Filter, used for bounded delay replication, was set to a short time, in this case to 3 seconds, because no conflicts have to be detected. In the first run 10000 transactions with ASAP replication were executed and the duration of the integration process was measured. Then we ran 10000 transactions with bounded delay replication with the same measurements. The results of those runs are shown in figure 5.5. In this figure it can be seen that for the first 2000 transactions the

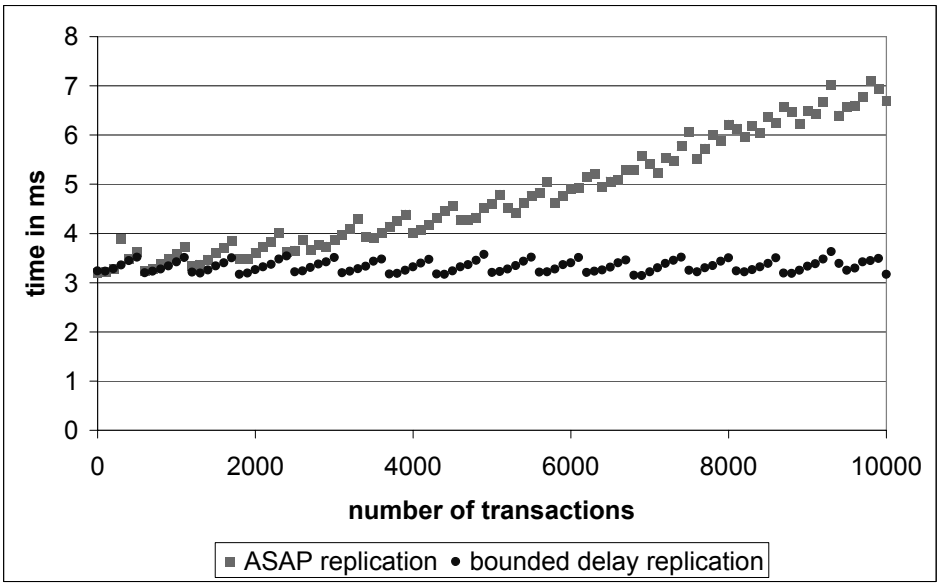

Figure 5.5: Result: Long run test

duration of both integration processes were nearly the same but then the duration of the ASAP integration process starts to grow as expected, whereas the duration of the bounded delay integration process stays within a bound. Also for the bounded delay integration it can be seen that the duration is also growing until the Log Filter is pruned and then the duration starts growing again. The reason for this is the Log Filter. For ASAP replication more information has to be stored for a longer time. At the moment there is theoretically no point in time for pruning the ASAP Log Filter. The bounded delay Log Filter can be pruned after a certain time, so the bounded integration process has only a bounded number

of entries in the Log Filter in opposite to the ASAP integration process.

5.4.3 Tests on OSE Delta

Unfortunately, there was no possibility for testing on OSE 4.5 and the planned tests could not be done. These tests are left open for future work.

Chapter 6

Conclusion

This chapter contains the conclusions from our work about bounded replication. At first the work is summarized, afterwards we list what we regard as our main contributions. Finally we show what is seen as future work to conclude bounded delay replication.

6.1 Summary

We define conditions, which determine an upper time bound for the replication process. Supplementary, we need restrictions that ensure the predicability of the replication process, formulated as a number of assumptions.
The assumptions define a time period that helps us to determine the upper time bound, determine a certain number of transactions that can arrive in this time period and also state that the Log Filter which is used for conflict detection has an upper bound of entries.
We have developed a modularized software design and thus this software design aims to be extensible for adding additional replication types. This software design is the basis for the implementation of bounded delay replication in DeeDS.
We have finalized the database part of the the implementation of ASAP replication and reuse some of its existing functionality for implementing bounded delay replication. The main point concerning the implementation is that every component has to be predictable in order to ensure bounded delay replication. One of the most difficult components is conflict detection. Which needs to be done in bounded time as well. This is assured by the fact that the conflict detection searches the bounded sized Log Filter only once.

We defined test cases for different environments, for testing our implementation. When working with a real-time system it is necessary to test the functionality and the timeliness of the system. Hence, we distinguish between functionality tests and timing tests that all can run in the different environments.
In the functionality tests we show, in 16 test cases, the correct functionality of bounded delay replication in DeeDS.
The timing test ended with the expected results and, therefore, it is possible to say that bounded delay replication is not blocked by ASAP replication. The timing tests still have to be re-run in a predictable test environment, such as with dedicated hardware and on a real-time operating system environment, so that the results of our tests can be confirmed.
Also, a real-time network implementation is required for actually measuring the timeliness of a complete system. When we started the project we did an evaluation of real-time operating systems and real-time networks. These are important since one of our assumptions says: "every part that is involved in bounded delay replication must be predictable, in order to guarantee a bounded time for the entire replication process" (assumption 4.1). The evaluation was only done for the DeeDS project to find suitable real-time platforms. The results of the evaluations can be found in appendix A and B.

6.2 Contributions

We have made the following contributions:

Determination of a Maximum Bounded Replication Time: The following formula gives the result of the maximum bounded replication time that we have found: MaxReplicationTime = MaxPropagationTime + NetworkDelay + 2 * TimePeriod

Development of a modularized software design: This software design shows how it is possible to integrate bounded delay replication into a system with an existing ASAP replication. The software design is then the basis of the implementation DeeDS.

Implementation for DeeDS: With the implementation of bounded delay replication we show, that it is possible to extend the current implementation containing ASAP replication with another replication type, namely bounded delay replication.

Setup of a proper test environment: This is done by the definition of several test cases to test the functionality and the timeliness on different platforms and architectures and evaluation of potential real-time operating systems and real-time networks.

Testing the correctness of the implemented system: The test environment builds the basis for the system test. In these tests we show that the implementation works correctly in functionality and also the behaviour concerning timeliness is shown in the test results.

6.3 Future Work

For completing the implementation of bounded delay replication an Admission Controller, based on our theory, is necessary. So far only the beginning of an Admission Controller is implemented that only determines the time period in which the system is and whether the system can allow the incoming local transactions in this time period or not. The future Admission Controllre needs to know the number of nodes in the system, the maximum number of updates, the maximum length of a read-write cycle and the maximum transaction execution time. With this knowledge it is possible to determine the time period. Thereby, the Admission Controllre can decide whether a transaction is allowed to enter the system and the replication can be done in the specified (bounded) time.

Another goal for future work is conflict resolution. When the conflict resolution is implemented, the upper time bound for the integration process can be determined. At the moment, only the time for conflict detection is added to the integration process. To ensure bounded delay replication it is also necessary that the conflict resolution is done in bounded time. For the conflict resolution it is important to determine the length of a read-write cycle. A proof for our hypothesis that the length of a read-write cycle depends on the number of objects in the database has to be found.

For testing the implementation and validating the theory behind the implementation, the system should be modified in the sense that it can run on OSE Delta 4.5.1 hardkernel. When this is done, a real-time network should be set up for testing the system with several distributed nodes.

After the setup of a distributed system, it should be possible to re-run all the test cases

described and to validate the results of this dissertation.

Chapter 7

Acknowledgements

First of all, we would like to thank our supervisor Gunnar Mathiason for his support, his help and for lending us some of his time to discuss problems and ideas.
We would like to thank Sten F. Andler for his support, his open mind during our discussions and making our stay in Skövde possible.
We would also like to thank the whole Distributed Real-Time research group at the University of Skövde, especially Henrik Grim for his guidance concerning coding matters, Sanny Gustavsson for fruitful discussions and proofreading and Marcus Brohede for helping us with the servers and always being in a good mood.
Furthermore, we thank Måns Telander and ENEA for the OSE support.
We thank our Erasmus co-ordinators Elizabeth Özdemir and Jutta Mülle (University of Karlsruhe) for arranging a pleasant stay in Skövde.
Last but not least, we thank our girlfriends and our parents for their support during our stay in Sweden far away from home.

Appendix A

Investigation of Hard Real-time Operating Systems

A.1 RTAI

RTAI stands for Real Time Application Interface. Therefore RTAI is not a standalone real-time OS but based on a Linux kernel. It is a patch to the Linux kernel which introduces a hardware abstraction layer. The functionality of the hardware abstraction layer is described later in this section. RTAI includes the following features:

- POSIX 1003.1c compatibility (Pthreads, including mutexes and condition variables)
- POSIX 1003.1b compatibility (POSIX message queues (Pqueues) only)
- Traditional RTOS IPCs including: Semaphores, mailboxes, FIFOs, shared memory, and RPCs
- Dynamic Memory Allocation - non-blocking in the Real-Time domain.
- /proc interface, which provides information on the real-time tasks, modules, services and processes extending the standard Linux /proc file-system support.
- LXRT, which allows the use of the RTAI system calls from within standard user space
- UniProcessor, Multi-UniProcessor and Symmetric Multi-Processor (SMP) support

- FPU support
- One-shot and periodic schedulers

RTAI modules take control of hardware interrupts, and schedule Linux as a very low priority task. Linux only executes when there are no real time tasks to run, and the real time kernel is inactive. Linux processes can be preempted by hard realtime processes at any time. Linux processes are prevented from disabling/enabling interrupts directly. So, the Linux kernel can never block interrupts. The interrupt dispatcher uses the concept of HAL (hardware abstraction layer), which traps the interrupts. The Real-Time Hardware Abstraction Layer (RTHAL) only passes interrupts to the Linux kernel when there are no active real-time tasks. Interrupts for real-time processes are directed immediately to the corresponding task. The changes needed to the standard Linux kernel are minimal, a few lines in eleven source files plus configuration additions to three files in the build structure, (Makefile, configuration files etc). The patch makes the following changes:

- Collects pointers to interrupt-related functions and structures within Linux, and copies them to a single structure.
- Creates substitute functions for cli/sti (clear/set Interrupt flag) and processor locking.
- Substitutes the original sti/cli and locking functions in the Linux kernel with the new RT-HAL functions.

This lower intrusion on the standard Linux kernel improves the code maintainability, and makes it easier to keep the real time modifications up-to-date with the latest release of the Linux kernel. The real time extensions can also easily be removed by replacing the interrupt function pointers with the original Linux routines. This is useful for performance comparison between Linux with and without real-time extensions. RTAI is broken up in several modules which may be loaded/unloaded as required. The most important ones are:

- Rtai: The principle module which handles the RTHAL.
- Rtai-sched: Implements the real-time scheduler and interprocess-communication (IPC) mechanisms such as mailboxes and semaphores.

- Rtai-fifos: Implements FIFO-based IPC.
- Rtai-shm: Implements shared memory.
- Rtai-lxrt: Exports the RTAI interface to processes running in user-space.

A.2 The Posix compliant API of RTAI

The following calls are supported by the Posix compliant API of RTAI:

Basic Taskfunktions

- pthread_create
- pthread_exit
- pthread_self
- pthread_setschedparam
- pthread_getschedparam

- pthread_attr_init
- pthread_attr_destroy
- pthread_attr_setdetachstate
- pthread_attr_getdetachstate
- pthread_attr_setschedparam
- pthread_attr_getschedparam
- pthread_attr_setschedpolicy
- pthread_attr_getschedpolicy
- pthread_attr_setinheritsched

- pthread_attr_getinheritsched
- pthread_attr_setscope
- pthread_attr_getscope

Synchronization

- pthread_mutex_init
- pthread_mutex_destroy
- pthread_mutex_lock
- pthread_mutex_unlock
- pthread_mutex_trylock

- pthread_mutexattr_init
- pthread_mutexattr_destroy
- pthread_mutexattr_setkind_np
- pthread_mutexattr_getkind_np

- pthread_cond_init
- pthread_cond_destroy
- pthread_cond_wait
- pthread_cond_signal
- pthread_cond_broadcast
- pthread_cond_timedwait

- pthread_condattr_init
- pthread_condattr_destroy

Communication IPC

- mq_open
- mq_receive
- mq_send
- mq_close
- mq_getattr
- mq_setattr
- mq_notify
- mq_unlink

A.3 Linux RK

Linux RK is developed by Dr. Ray Raykumar at the Carnegie Mellon University. Linux RK adds the so called resource kernel to the existing Linux kernel so that real-time constraints can be guaranteed. Only a few changes are made to the existing Linux Kernel:

Makefile	Modify to compile and link RK files
arch/i386/Makefile	Do not check vmlinux for zlilo (takes too long to install)
arch/i386/kernel/entry.S	Add hooks when leaving from the kernel; add system call entries (201 - 205)
arch/i386/kernel/irq.h	Add hooks when entering the kernel
drivers/char/serial.c	Make the speed of serial console 38400bps from 9600bps
include/linux/sched.h	Add the pointer to a resource set in task structure; modify INIT TASK to initialize that pointer
include/asm-i386/unistd.h	Add system call entry numbers (201 - 205)
init/main.c	Add rk init()
kernel/sched.c	Add schedule hook

The following information are from a paper published Shuichi Oikawa and Ragunathan Rajkumar with the title: Linux/RK: A Portable Resource Kernel in Linux ([SR98]).
A resource kernel is one which provides timely, guaranteed and enforced access to physical resources for applications. With a resource kernel, an application can request the reservation of a certain amount of a resource, and the kernel can guarantee that the requested amount is available to the application. Such a guarantee of resource allocation gives an application the knowledge of the amount of its currently available resources. A Quality of Service manager or the application itself can optimize the system behaviour by using the available resources. Linux was the first choice for which the kernel is developed because of the free availability.
For the reservation of resources, such as CPU time, physical memory pages, a network bandwidth, or a disk bandwidth., a reserve is established. A reserve represents a share of a single computing resource. Different reserves can be combined to a resource set. A reserve is implemented as a kernel entity; thus it cannot be counterfeited. The kernel keeps track of the use of a reserve and will enforce its utilization, when necessary. Appropriate scheduling and enforcement of a reserve by the resource kernel guarantees that the reserved amount is always allocated for it. When a reserve uses up its allocated time units C within an interval T, it is said to be *depleted* . Otherwise it is *undepleted* . At the the end of the current interval T, the reserve will obtain a new quota and it is said to be *replenished* . There exist tree forms of reserves:

- *Hard reserves:* will not be scheduled on depletion until they are replenished.
- *Firm reserves:* scheduled for execution on depletion only if no other undepleted reserve or unreserved resource uses can be scheduled.
- *Soft reserves:* can be scheduled for execution on depletion along with other unreserved resource use and depleted reservations.

The changes to the existing Linux kernel are done by *callback hocks.* Those hooks catch relevant scheduling points in the Linux kernel, and sends these events to the resource kernel. The resource kernel uses the well-defined functions in the Linux kernel to control kernel entities, such as processes and device drivers. There exist also some mechanism which guarantee resource utilization based on reservation. These are:

- Admission control
- Scheduling Policy
- Enforcement
- Accounting

The resource kernel also provides an API (Application Program Interface) for use by user-level applications. User level applications use this API to create reserves and resource sets, and then attach their processes to these resource sets.
The combination of a timestamp counter and a high-resolution timer provide a accurate time management. A timestamp counter build into most modern CPUs, provides the standard time for the use by resource kernels. The representation of time in the resource kernel for use in accounting and scheduling is based on the values from this timestamp counter.
A high resolution timer is supported to make the enforcement of reserves more precise. It is implemented by using the one-shot mode of the ISA clock timer on chip in PC compatible systems. The resource kernel sets the latch for the next interrupt every time after a timer interrupt occurs.
More information is provided by different papers which are available on the homepage of the Institute at Carnegie Mellon University.

http://www-2.cs.cmu.edu/∼rajkumar/linux-rk.html
On this web side the resource kernel can also be downloaded.

A.4 OSE

OSE Delta supports hard real-time constraints. Thereby it is a fully pre-emptable real-time operating system. The following processors are supported by OSE Delta: all PowerPCs, ARM 7-9, strongARM, Xscale and MIPS CPUs. The footprint of the kernel starts at 100Kb. And it is also possible to support multiprocessors with the OSE link handler. With the link handler it is possible to build distributed systems with transparent IPC (inter process communication) between different CPU's. The OSE architecture is based on message passing this is described later in detail. C, C++, Ada and Java are the supported programming languages of OSE Delta. OSE Delta has no features for real-time communications, but this can be bought in from other companies. OSE only supports a TCP/IP stack for networking, other protocols must be bought also.
OSE is a fully pre-emptive RTOS which means that a higher priority task will always interrupt a lower priority task, even during the execution of a system call. OSE has three different process types (Interrupt processes, triggered by an hardware interrupt or by software, Timer interrupt processes, triggered by an internal timer and Prioritized processes, written as infinitive loop and can either be running , waiting or blocked). Each process type has 32 priority levels, an interrupt process has always higher priority then an prioritized process. Processes on the same priority level are scheduled due to a FIFO queue. In OSE it is also possible to assign new priority to a process in runtime.
OSE has a message-based architecture and does not rely on semaphores and shared memory for communication. The message passing Inter-Process Communication is fully supervised by the RTOS. In this architecture the two main entities are processes and messages:

Process: An OSE process is an independent object that contains the application code. Every process has its own message queue. There a two process categories: Static and Dynamic, and four types of processes: Interrupt -, Timer Interrupt -, Prioritized - and Background Processes. All four-process types are fully supervised by the kernel.

Message: An OSE message is an independent non-shared entity that contains not only the message ID and the data, but also the sender, receiver and owner of the message. This allows strict error checking and enables system-level debugging and tracing on all message passing activity within the system. Messages are always sent directly from a process to another process.

The data in the message itself can be any user-defined length, limited to eight different (user configurable) fixed buffer sizes to prevent memory fragmentation. In an OSE system it makes no difference if a message is sent to a process that is running in the same CPU or somewhere in a remote node of the system. The communication between the processes is always fully supervised by the kernel and fully transparent to the application code.
The advantages of this programming model are:

- Easy to learn an understand
- Fully transparent IPC
- Object oriented model
- Compatible with top-down development
- No set up and supervision of queues and mailboxes required
- Allows very powerful system level debugging
- Very strict error checking and supervision
- Easy to maintain code
- Easy to re-use code
- Very high performance
- Fully controlled by the RTOS kernel

Blocks: OSE-Delta processes can be grouped together to form logical Blocks of processes. This allows the system to be split up in several independent functional blocks that can be implemented by separate development groups. A block of processes can have its own

memory pool, and treated as one big 'super'-process by the other processes by sending and receiving messages to and from this block. Also, entire blocks of processes can be started, killed or replaced. The OSE Program Handler makes it is easy to load and unload blocks at run-time. Error handling can also be performed at the block-level. Within a block, message redirection is used to make IPC transparent at the block level.

Appendix B

Real-time Ethernet

B.1 CSMA-DCR

CSMA-DCR is a deterministic variant of the Ethernet standard. Thereby conventional CSMA-CD chips and CSMA-DCR chips can co-exist and exchange messages over the same shared medium. The reason for this is that the variation between CSMA-CD and CSMA-DCR simply consists in replacing the binary exponential back-off algorithm with a deterministic binary tree search algorithm. While no collision occurs CSMA-DCR behaves exactly like CSMA-CD. Upon collision occurrence, CSMA-DCR iteratively separates the set of messages sources. Only an upper bound on the number of sources need be known at configuration time.

The collision is handled in the following way:

Let Q be the number of consecutive indices allocated to the message sources. Let q be the smallest power of two greater than or equal to Q. So if Q is 5, q is 8. Whereby each index identifies a leaf in the binary tree, when a collision occur the binary tree is separated. In [LR93] the tree is separated in the subset [0,q/2[. This is repeated until the winning set either is empty or contains exactly one active source.

B.2 DOD/CSMA-CD

DOD/CSMA-CD is similar to CSMA-DCR. The major difference stems from using indices that are computed on-line, rather than resting solely on indices that are pre-allocated to

sources. With this algorithm, some problems which could not be solved with CSMA-DCR might be solvable for DOD/CSMA-CD.

B.3 Switched real-time Ethernet

Switched real-time Ethernet uses enhancements to full-duplex switched Ethernet to achieve the ability of giving throughput and delay gurantees. Thereby the switch and the end-nodes control the real-time traffic with Earliest Deadline First (EDF) scheduling on the frame level. There is no modification to the Ethernet standard necessary. The switch is responsible for admission control where feasibility analysis is made for each link between source and destination. The switch broadcasts Ethernet frames regularly to clock synchronize the end nodes. In Switched real-time Ethernet non-real-time-traffic is supported as well.

On a switch-based LAN the possibility to offer real-time services is also improved. The main-point in switch-based LAN's is that the collision possibility in Ethernet networks can be eliminated and methods to support real-time services can be implemented in the switches without changing the underlaying widespread protocol standard. To suppport real-time only a thin real-time layer is needed between the Ethernet protocols and the TCP/IP suite in the end-stations.

For the architecture only a single switch is assumed. Every node in the network is connected to other nodes via the switch and nodes can communicate with each other over logical real-time channels (RT channels), each being a virtual connection between two nodes in the system.

To set up an RT channel the application interacts directly with the RT layer. The RT layer then sends a question to the RT channel management software in the switch. Outgoing real-time traffic from end-node uses UDP and is put in a deadline-sorted queue in the RT-layer. Outgoing non-real-time traffic from the end-node typically uses TCP and put it in a FCFS-sorted queue in the RT layer. In the same way there are different output-queues for each port on the switch too. Before real-time traffic may be deliverd an RT-channel must be established. This is done by request and acknowlodgement communication where the source node, the destination node and the switch agree on the channel establishment. When the RT channel has been established the network guarantees to deliver each generated message

with a bounded delay, $T_{maxdelay} = T_{deadline} + T_{latency}$.

B.4 RETHER

RETHER is a software-based timed-token protocol and stands for Realtime ETHERnet. It supports real-time bandwith guarantees without any modifications to existing ethernet hardware by adopting a contention-free deadline-driven token-bus protocol to provide predictable performance. The RETHER protocol has the following features:

- It allows applications to reserve bandwith and guarantees the reservation throughout the lifetime of the application
- It is implemented completely in software over off-the-shelf hardware.
- It adopts a hybrid scheme whereby the network operates using a timed token-bus protocol when there are real-time sessions and using the original ethernet protocols at all other times for better performance of non-real-time traffic.

There are two modes: the regular CSMA/CD mode and the real-time RETHER mode. When a node wants to send real-time messages, it broadcasts a Switch-to-Rether message on the Ethernet, which must be acknowledged by all other nodes. There are two reasons for the necessity of acknowledgements. Firstly, they signify the willingness of the nodes to switch to the real-time mode. Secondly, they indicate that there are not any pending packets in the backoff phase of the CSMA/CD protocol. After the transition to the RETHER mode, a token based sending policy is used for real-time traffic. After the last node finishes its real-time request, the token is destroyed and a Switch-to-CSMA message is broadcasted.

Bibliography

[And96] Andler, S., Hansson, J., Eriksson J., Mellin J., Berndtsson M. and Eftering B.: DeeDS towards a distributed active and real-time database system, Special Issue on Real Time Data Base Systems, ACM SIGMOD Record, 25(1)

[BN92] Brachmann B. & Neufeld G.: TDBM: A DBM Library with Atomic Transactions, Department of Computer Science, University of British Columbia

[BW01] Burns, A & Wellings, A.: Real-Time Systems and Programming Languages (3 ed.), Harlow, England, Addison Wesley

[EN94] Elmasri, R., Navathe, S.: Fundamentals of Database Systems (2nd ed.), Redwood City, USA, Benjamin/Cummings Publishing Company, Inc.

[Eri98] Daniel Eriksson: How to implement Bounded-Delay replication in DeeDS, B.Sc. dissertation, Department of Computer Science, Högskolan i Skövde

[HHB96] A. Helal, A. Heddaya, B. Bhargava: Replication Techniques in Distributed Systems, Massachusetts, USA, Kluwer Academic Publishers

[HJHK01] Hoai Hoang, Magnus Jonsson, Ulrik Hagströ, Anders Kallerdahl: Switched real-time ethernet with earliest deadline first scheduling - protocols and traffic handling, School of Information Science, Computer and Electrical Engineering, Högskolan i Halmstad

[LR93] Gérard Le Lann, Nicolas Riviere: Real-Time Communications over Broadcast Networks: the CSMA-DCR and the DOD-CSMA-CD Protocols

[Lun97] Johan Lundström: A conflict detection and resolution mechanism for bounded-delay replication, M.Sc. dissertation, Department of Computer Science, Högskolan i Skövde

[Mat02] Mathiason Gunnar: Segmentation in a Distributed Real-Time Main-Memory Database, M.Sc. dissertation. Department of Computer Science, Högskolan i Skövde

[Mul93] Mullender S.: Distributed Systems (2nd), Addison-Weasly, 1993

[PR82] Parker D.S. & Ramos R.A.: A distributed file system architecture supporting high availability, Proc. of sixth Berkely Workshop on Distributed Data Management and Computer Networks, pages 161 - 183

[SR98] Suichi Oikawa & Ragunathan Rajkumar: Linux/RK: A Portable Resource Kernel in Linux, Carnegie Mellon University

[VC95] Chitra Venkatramani, Tzi-cker Chiueh: Design, Implementation and Evaluation of a Software-based Real-Time Ethernet Protocol