How to implement Bounded-Delay replication in DeeDS

(HS-IDA-EA-02-111)

Daniel Eriksson (a95daner@ida.his.se)

Institutionen för datavetenskap
Högskolan Skövde, Box 408
S-54128 Skövde, SWEDEN

Examensarbete på program för systemprogrammering under vårterminen 1998.

Handledare: Joakim Eriksson
How to Implement Bounded Delay Replication in DeeDS
Submitted by Daniel Eriksson to Högskolan Skövde as a dissertation for the degree of BSc, in the Department of Computer Science.

2002- 06 - 25
I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.
Signed: _______________________________
How to implement Bounded-Delay Replication in DeeDS

2002 - 06 - 25

Key words: Real-Time Systems, Database Systems, Replication, Eventual Consistency.

Abstract

In a distributed database system, pessimistic concurrency control is often used to ensure consistency which implies that the execution time of a transaction is not predictable. The execution time of a transaction is not dependent on the local transactions only, but on every transaction in the system.

In real-time database systems it is important that transactions are predictable. One way to make transactions predictable is to use eventual consistency where transactions commit locally before they are propagated to other nodes in the system. It is then possible to get predictable transactions due to the fact that the execution time of the transaction only depends on concurrent transactions on the local node and not on delays on other nodes and delays from a network.

In this report an investigation is made on how a replication protocol using eventual consistency can be designed for, and implemented in, DeeDS, a distributed real-time database prototype. The protocol consists of three parts: a propagation method, a conflict detection algorithm, and a conflict resolution mechanism. The conflict detection algorithm is based on version vectors. The focus is on the propagation mechanism and the conflict detection algorithm of the replication protocol.

An implementation design of the replication protocol is made. A discussion on how the version vectors may be applied in terms of granularity (container, page, object or attribute) and how the log filter should be designed and implemented to suit the particular conflict detection algorithm is carried out. A number of test cases with focus on regression testing have been defined. Three test cases have been run and the results conformed to the expected outcome defined for these test cases. This was successful.

It is concluded that the feasibility of the conflict detection algorithm is dependent on the application type that uses DeeDS.
# Table of Contents

1 **Introduction**........................................................................................................6
   1.1 Outline of the report..........................................................................................6

2 **Background** .........................................................................................................8
   2.1 Real-Time Systems ..........................................................................................8
   2.2 Distributed Systems .....................................................................................8
   2.3 Database Systems..........................................................................................9
      2.3.1 Transactions .......................................................................................9
      2.3.2 Concurrency Control .........................................................................10
      2.3.3 Transaction recovery .........................................................................10
   2.4 Distributed Database Systems ....................................................................11
   2.5 Replication in Distributed Systems ...........................................................11
      2.5.1 Immediate Consistency .......................................................................11
      2.5.2 Eventual Consistency .........................................................................12
   2.6 DeeDS .............................................................................................................13
      2.6.1 TDBM (DBM with Transactions) .........................................................15

3 **The Conflict Detection Algorithm** ..................................................................16

4 **Problem description** .......................................................................................19
   4.1 Bounded-Delay replication protocol .........................................................19
      4.1.1 Propagation protocol ..........................................................................19
      4.1.2 Conflict detection ...............................................................................20
      4.1.3 Conflict resolution ...............................................................................20
   4.2 The problem ...................................................................................................21
   4.3 Limitations of this project .............................................................................22

5 **Approach** ...........................................................................................................23
   5.1 Design Overview............................................................................................23
   5.2 Propagation of updates..................................................................................25
      5.2.1 Modifications to transaction handling in tdbm .....................................26
   5.3 Granularity .....................................................................................................26
      5.3.1 Container level .....................................................................................26
      5.3.2 Page level ............................................................................................27
      5.3.3 Object level ........................................................................................27
      5.3.4 Attribute level ....................................................................................27
   5.4 Data representation .........................................................................................27
      5.4.1 Representation of version vectors .......................................................27
   5.5 Log filter design .............................................................................................29
      5.5.1 Using a database file for log filter information .....................................29
      5.5.2 Using lists for log filter information ....................................................29
      5.5.3 Discussion result .................................................................................30
   5.6 Software test...................................................................................................30
5.6.1 The test process ................................................................. 30
5.6.2 Test cases ................................................................. 30
5.7 Summary and conclusions ........................................... 33

6 Results ......................................................................................... 35
6.1 Review of the Logger implementation .......................... 35
6.2 Test results ................................................................. 36
6.3 Discussion ................................................................. 36
6.3.1 Implementation ............................................................. 36
6.3.2 Test ................................................................. 37
6.3.3 Complexity of the algorithm ................................. 37

7 Conclusions ................................................................................. 40
7.1 Contributions .............................................................. 40
7.2 Future work .............................................................. 40

References ....................................................................................... 43
Appendix A: Enhanced version vector algorithm ............. 45
1 Introduction

Many real-time systems, such as control systems in modern fighter aircraft, need to store large amounts of sensor data. This data is stored in a database which allows structured storage and fast access to information. To achieve fault tolerance, enhance availability and get a decentralized system, data is replicated to several nodes. A replicated database is distributed over several nodes. The degree of replication can vary between a fully replicated database and a partial replicated database. One problem that replication brings is increased difficulty to uphold consistency of the replicas. There are several replication protocols that solves this kind of problem, but they often introduce non-determinism and restrain predictability due to hard consistency criteria. This is not appropriate in a real-time system. Therefore, other replication protocols have been developed that use eventual consistency.

When using eventual consistency, transactions commit locally before they are distributed to other nodes. Updates will be propagated to the other nodes and eventually they will be consistent, i.e., contain the same data. This implies that there is inconsistency in the system for some time. In real-time environments this can be tolerated due to the fact that transactions must commit within a specified time period and this implies that transactions are more predictable.

At the department of Computer Science, Högskolan i Skövde, a Distributed Active Real-Time Database System, called the DeeDS prototype, has been developed as a platform for research in the area of distributed real-time database systems. However, in the current DeeDS prototype there is no replication mechanism implemented, only a high-level design of the replication mechanism has been developed, including a conflict detection algorithm.

This report considers the follow-up project of Johan Lundström's M.Sc. dissertation, A Conflict Detection and Resolution Mechanism for Bounded-Delay Replication (Lundström, 1997). Lundström suggested a propagation method, a conflict detection algorithm, and possible conflict resolution mechanisms suited for the demands of DeeDS using eventual consistency, without specifying low-level design and implementation details. With this as a foundation, methods for implementing Lundström’s replication protocol in DeeDS and how to verify whether that solution works satisfactorily in the real system, are investigated.

1.1 Outline of the report

The report is organised as follows. In chapter 2, a background is given to the areas relevant to the problem. That is real-time systems, distributed systems and database systems. Also, the concepts of eventual consistency in distributed databases is presented and how those concepts relate to distributed real-time systems. The DeeDS project and the architecture of the prototype is briefly described. Chapter 3 contains a description and some examples on the enhanced conflict detection algorithm (Lundström, 1997). Chapter 4 defines the problems and the limitations and constraints of the project. Chapter 5 describes the approach and describes the software design of the replication protocol. Discussions and argumentation related to the choice of, for example data representation, for design solutions are carried out. Test cases are also defined in this chapter. In Chapter 6 the results are presented.
Introduction

together with a discussion about the different design decisions made. Finally, the conclusions for this project are outlined in chapter 7, together with some discussion about contributions and future work.
Background

2 Background

This chapter covers the background material associated with the problem. It also provides an overview of distributed database systems and the differences between immediately consistent and eventual consistent replication protocols. Some issues are at a high level and are not necessary to understand the complete problem, but to get an overview of their importance and how they relate to each other.

2.1 Real-Time Systems

A Real-Time System is, according to Burns and Wellings (Burns & Wellings, 1997, pp. 2), defined as follows:

**Definition:** A Real-Time System is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment.

A real-time system must be predictable in its execution so that it performs operations within a bounded time. The fundamental attribute is **timeliness** which requires **predictability** and sufficient **efficiency** of the system. Predictability means that there is an upper bound on resource requirements. Sufficient efficiency refers to that the peak load is escheatable i.e, all tasks can be guaranteed to meet their deadlines at all times.

Real-Time systems can be either hard, firm, or soft. In a hard real-time system, meeting the deadlines is very important. Missing deadlines might result in loss of human lives or equipment may be damaged. An example of a hard real-time system is the controlling system of a nuclear power plant.

In a soft real-time system a deadline can occasionally be missed, and the errors or failures that might follow from a missed deadline can be tolerated.

A real-time system must be fault tolerant to be reliable. Research in the area of reliable and fault tolerant computing has led to the notion of **dependability**. There are especially four attributes of dependability that are of importance, as pointed out in Mullender (Mullender, 1994, chapter 16). These are reliability, safety, availability and, maintainability. They are described in Laprie (1994), and here follows a short description of each:

- **Reliability:** Low probability of failure (while operating for a certain period of time).
- **Safety:** When failure occurs, effects are not catastrophic.
- **Availability:** High probability of service operating when requested.
- **Maintainability:** Short time from a component’s failure to its reintroduction in system.

2.2 Distributed Systems

Burns and Wellings (1997, pp. 441) define a distributed system in the following way:

**Definition:** A distributed computer system is a system of multiple autonomous processing elements, cooperating in a common purpose or to achieve a common goal.
Distributed systems are classified into tightly coupled systems and loosely coupled systems. In the first case, the processing elements share a common memory or clock. In the second, each processing element has its own memory and the nodes are connected via communication links (Burns & Wellings, 1997).

One difference between tightly and loosely coupled systems is the kind of synchronization mechanism used. In a tightly coupled system the synchronization is usually based on shared variables, whereas in a loosely coupled system message passing has to be used (Burns & Wellings, 1997).

2.3 Database Systems

Database systems store large amount of data in a structured way, and make that data available to many users. Database systems are used in a large variety of areas, such as storage of information about bank accounts, and temporary storage of data from sensors e.g., in an aircraft. The data is available for reading and manipulation using transactions (see 2.3.1).

A database may be centralized in a client-server architecture where the users access one common database on a server, or the database may be decentralized and distributed on several nodes.

In the following subsection, issues related to transactions such as concurrency control and recovery are discussed. The next section (2.4) gives a brief description of problems that can arise in a distributed database system.

2.3.1 Transactions

A transaction is an execution of a program consisting of operations that access or change the contents of the database, Elmasri et al. (1994).

Several transactions can be executed sequentially or concurrently. When they are executing concurrently there is a risk that problems occur, which can cause conflicts. These problems are described in Elmasri (1994) and are summarized below.

- **The Lost Update Problem:** Occurs when a transaction $T_1$ reads a value that is updated by another transaction $T_2$ but which has not been written back to the database. This results in $T_1$ reading an incorrect value.

- **The Temporary Update Problem:** Occurs when a transaction $T_1$ reads a value updated by a transaction $T_2$ that later fails and the data item has not been changed back to its original value.

- **The Incorrect Summary problem:** Occurs when a transaction is doing a summary of the values in a number of records and some of the records are concurrently updated by other transactions. This leads to an incorrect summary because some values may be read before they are updated and others after they are updated.

To avoid these problems, some kind of concurrency control technique is needed.
2.3.2 Concurrency Control

Transactions should have certain properties. The concurrency control mechanisms are responsible for upholding these properties, which are called the *ACID* properties (Elmasri & Navathe, 1994):

- **Atomicity**: A transaction is an atomic unit and the execution should be performed entirely or not at all.
- **Consistency**: A correct execution of a transaction from a consistent state must ensure that the database end up in a consistent state.
- **Isolation**: A transaction should not make its updates visible to other transactions until it was committed (completed successfully).
- **Durability**: Once the transaction has committed the changes must never be lost because of subsequent failures.

The list of executed transactions forms the transaction *schedule*, which is a history of all transactions performed. It is defined as follows:

**Definition:** A schedule $S$ of $n$ transactions $T_1, T_2, \ldots, T_n$ 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 $T_i$ (Elmasri & Navathe, 1994).

Transactions may execute in serial order. That is: all operations of a transaction are executed in sequence, without being interrupted, before the execution of the next transaction is started. This is inefficient in a multi user system with I/O wait due to the fact that all transactions must be put on hold until the running transaction completes. Instead, concurrent execution is usually desired. However, when several transactions are executed concurrently there is a risk that they may cause some of the problems described in 2.3.1. Transactions must execute without causing these kind of problems. To avoid this transaction schedules can be made serializable:

**Definition:** A schedule $S$ of $n$ transactions is **serializable** if it is equivalent to some serial schedule of the same $n$ transactions (Elmasri & Navathe, 1994).

To achieve this, some kind of concurrency mechanism must be used to control the execution. An example of this is a two-phase locking protocol as described in Elmasri et al. (1994).

2.3.3 Transaction recovery

If a transaction is unable to commit due to a hardware problem or an error in the environment that leads to loss of data, recovery must be performed to handle this situation.

There are several ways to perform transaction recovery. Two techniques are described here: *roll-back of transactions* and *shadow paging*.

**Transaction Roll-back**

When a failure occurs (because of a system crash or a transaction system error), the database must be brought back to a consistent state. This is made by using a *system log* which
keeps track of all transaction operations that have modified any values in the database. The values are *rolled back* from the log and put back in to the database (Elmasri & Navathe, 1994).

**Shadow Paging**

When using shadow paging the database is considered to consist of several fixed-size pages which hold the data. A table (or directory) is used with the same number of entries as there are pages. Each entry contains a pointer to a specific page. This is used to find the right page. When a value on a page is to be updated a copy of the entire page is made. Thereafter all modifications are made on the copy of the page. When a transaction commits, the table is updated to that the pointer now point to the new page. If a failure occurs during the transaction, the page is discarded and no roll-back is needed. With this scheme no modifications are made to the actual database, which makes it easier to undo modifications (Elmasri & Navathe, 1994).

**2.4 Distributed Database Systems**

Distribution of data is a way to achieve decentralization and better service for the users, in terms of faster access to information, due to the fact that much information can be accessed locally. It is also one way to make a system fault tolerant. This is very important in, e.g., real-time environments where a computer failure may be hazardous (see 2.1).

The main problem with distribution and replication of data is to uphold consistency in the system. Every update made on one node must be propagated to and performed on all other nodes. Concurrency control is complicated due to the fact that serializability no longer is dependent on local transactions only, but on every transaction in the system. There are several *replication protocols* available that solve the problems with inconsistency and resolves any possible conflicts. These are described in the next section.

**2.5 Replication in Distributed Systems**

Replication of data requires a mechanism that provide conflict detection and thereby consistency among nodes. The serializability criterion must be extended so that the nodes are synchronized. Updates made in the system should not be seen until they are committed and registered and approved by all nodes in the system.

There are two classes of replication protocols available that does this. There are those that use *immediate consistency* and those that use *eventual consistency*. They are described in 2.5.1 and 2.5.2 respectively.

**2.5.1 Immediate Consistency**

The protocols that enforce immediate consistency are called *classical protocols* in Helal et. al. (1996). These are divided into different categories.

- **Read One Write All (ROWA):** A read operation on item d is performed on one replica (chosen from all replicas in the system). The write operations are made on all replicas in the system. The local concurrency control mechanism must guarantee mutual exclusion.
Background

- **Read One Write All Available (ROWA-A):** This is the same as the ROWA except for the write operations. Here write operations only have to update values on all available replicas. This makes the protocol more tolerant to failures. Sites which were not available when the update was made must copy the updates from another site before they become visible to all other nodes.

- **Primary Copy ROWA:** A specific copy of the data item is called the primary copy and the others are called backup copies. The write operations are performed on the primary and on all the available backups. The transaction performing the update is only allowed to commit when the primary copy and all the backup copies have registered the update. The read operations are executed on the primary copy. The problem that can arise with this protocol is when delay in communication makes the backup copies think that the primary has failed. When the primary fails a backup copy is chosen (in some way) to be the new primary copy. If delay in communication is the reason for a new primary to be selected, this can lead to a situation where the system contains two primary copies.

- **Quorum Consensus:** In the above described ROWA protocols the write operations are not allowed to be completed if not all replicas are unavailable (due to network failure for example). Therefore, Quorum Consensus protocols are used. With such protocol the write operations are executed only on a subset of the available sites. This subset is called the write quorum. The read operations are executed on a read quorum which is guaranteed to intersect with the write quorum. In this way a read operation on a value is guaranteed to access the latest version of that value. QC protocols can tolerate both site and communication failures (Lundström, 1997).

### 2.5.2 Eventual Consistency

The protocols described in 2.5.1 ensure immediate consistency. There are other protocols that guarantee only eventual consistency instead. Updates made are committed locally before they are distributed over the network to all other replicas. The protocols are called optimistic because they not limit availability under network partitions in order to maintain consistency (Helal et al., 1996).

Protocols that use eventual consistency does not use conflict avoidance to prevent conflicts. They use conflict detection and a resolution mechanism to resolve conflicts.

The conflict detection mechanism used in this work is the version vector replication algorithm. It is described by Parker and Ramos (1982). Version vectors are defined in the following way:

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

An example: Consider a system with three nodes (A, B, and C) and two distributed files \( f \) and \( g \). Each file has a version vector attached to it which is used to indicate if a node makes an update to it. Consider a situation where node A has made an update on file \( f \) and node C has made an update on file \( g \). The version vectors will end up looking like this: \( f = \langle A:1, B:0, C:0 \rangle \) and \( g = \langle A:0, B:0, C:1 \rangle \).
This indicates that there are three nodes in the system and that node A has made an update to the file f and node C has updated file g.

A conflict is detected if neither of two version vectors, V and V', dominate the other. A version vector V dominates another version vector V' if V_i >= V'_i for every item i in the vector.

The version vector algorithm is not suited for a multifile environment (such as a database environment) due to the fact that it only detects write-write conflicts (Helal et. al., 1996). Parker and Ramos (1982) describes an algorithm that is based on version vectors that do detect multifile conflicts. This is the foundation for the enhanced version vector algorithm described by Lundström (1997), which is the foundation for Lundström’s and this work.

In Parker and Ramos’ (1982) description of a multifile system with version vectors, a log filter is used to store all version vectors used in a transaction. With this mechanism, both write-write conflicts and read-write conflicts can be detected. Examples of both types of conflicts will be given in the next chapter, where the algorithm is described. The log filter for a multifile environment is described below, where f is a file on which the transaction operates (Lundström, 1997, pp. 44):

**Definition:** A log filter is a group of sets S, S = \{f_1, ..., f_m\}, drawn from a set of files where for each set S and T, if S \neq T then S \cap T = \emptyset

### 2.6 DeeDS

The DeeDS prototype is a research project carried out by the Distributed Real-Time Systems Research Group at Högskolan i Skövde. The goals of the project are to develop and evaluate an architecture for distributed active real-time database systems with a minimum required level of functionality and considerable flexibility (Andler et. al., 1995; Andler et. al., 1998). The database is memory resident to eliminate unpredictable delays from disk access. It is also fully replicated, i.e., each node has a copy of the entire database, which helps to eliminate delays in access due to network communications.
An overview of the DeeDS architecture is shown in figure 1. The architecture is divided into application-related modules and critical system services (Andler et. al., 1998). These two modules reside on different processors for predictability and flexibility reasons. The application-related modules are the database system and the rule manager, while the critical system services module consist of the event monitor and the scheduler.

The rules in DeeDS are of the form Event Condition Action (ECA) (Chakravarthy, 1989). If an event occurs that matches a rule, and its condition evaluates to true, the action is executed.

The database component of DeeDS is a public domain object store, OBST (Object Management System of STONE) (Casais, 1992), whose storage manager has been replaced with tdbm, see section 2.6.1. The main reason for replacing the storage manager is that tdbm supplies nested transactions, which is strongly desirable for active database functionality and exception handling (Andler et. al., 1998).

In DeeDS, delayed replication is performed as soon as possible (ASAP) or within bounded time. Transactions commit locally, before the updates are replicated to all other nodes (eventual consistency). The concurrency control in the local nodes is pessimistic. Replication is done in three steps:

1. Updates are propagated to all other nodes.
2. New updates are checked against updates performed by other nodes for conflicts.
3. If conflicts are detected they are resolved as specified by the application programmer.
The event monitor module (EMM) handles the events that are generated in the sensors, which are connected to the system, or from other nodes in the system.

DeeDS uses dynamic scheduling, i.e. scheduling decisions are made at run-time, as opposed to static scheduling where all predictions are undertaken before execution, i.e. cyclic executive (Burns & Wellings, 1997).

2.6.1 TDBM (DBM with Transactions)

As stated above, the storage manager in OBST has been replaced with tdbm, which is an extension to the database library dbm that has been used in the UNIX environment.

Tdbm is a transaction processing datastore with a dbm-like interface and it provides the following:

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

Tdbm is a layered architecture that consists of three layers (Brachman & Neufeld, 1992): the item layer, the page layer and the transaction layer.

The Item Layer deals with the key/value pairs in a page. Two different pages are used: physical pages which contain a directory of items stored in the page together with zero or more items and indirect pages which contain data too large to fit in an ordinary page. An indirect page can consist of several physical pages.

The Page Layer is concerned with page management and allocation of physical pages. Also mappings from keys to logical pages and logical pages to physical pages are carried out in this layer.

The Transaction Layer provides nested transactions over logical pages. Locating the correct version of a page for a transaction, concurrency control and transaction recovery processing is also carried out in this layer.

The recovery mechanism is based on a transaction file (intention list) which is created at a commit of a top-level transaction. The transaction file contains several shadow pages that represent modified pages in the physical database. After a commit, a mode is set (using chmod()) on the transaction file to indicate that the transaction was completed successfully. After a successful commit of a transaction the transaction file can be removed (Brachman & Neufeld, 1992).

Recovery is made by removing incomplete transaction files (those without the special mode set) and apply the contents of complete transaction files.
The Conflict Detection Algorithm

A central part of this project is the conflict detection algorithm (see section 2.5.2) described by Lundström (1997). In this section, a brief description of the purpose and function of the algorithm is given. This is important to gain an understanding for the rest of this document and the purpose of the project.

Lundström’s algorithm is based on an algorithm used in distributed file systems (Parker & Ramos, 1982), but Lundström has refined it to better suit the needs of a distributed database system and calls it the “Enhanced version vector algorithm” (Lundström, 1997). A formal specification of the algorithm can be found in Appendix A.

The function of the algorithm is to detect conflicts between concurrent updates on different nodes. When an update is received on one node the algorithm is executed, which includes modifying a log filter (see 2.5.2). The log filter entries which contains sets of objects that have been read and/or updated by transactions. To each such set there are sequences of version vectors indicating which objects have been involved in the same transactions. Each version vector is associated with one object.

The principle of the algorithm is to gather the objects, that have been accessed (read) or updated (write) together in the same transaction, in sets. This is necessary to see if common objects that have been updated are in conflict. According to the algorithm, this is done by going through the read set received in the update and fetch all the version vector sequences, associated with that particular object, from the log filter. A version vector sequence is a vector or list of objects and their version vectors which have been included in the same transaction. This information (both the objects and the version vector sequences) is merged, by a UNION operation (Aho, Hopcroft & Ullman, 1974), with the objects and sequences that are common in the new update. Here follows a small example of this:

\[ \text{i) } \quad \text{LF} = \{ \{a\}[<1,0,0>], \{b\}[<0,1,0>] \]  
\[ \text{ii) } \quad \text{Update}\{a,b,c\}, [<1,0,0>, <0,1,0>, <0,1,0>], \{c\} \]  
\[ \text{LF} = \{ \{a,b,c\}, \]  
\[ \quad [<1,0,0>, \ldots, \ldots, \ldots, \ldots] \]  
\[ \quad [\ldots, <0,1,0>, \ldots, \ldots] \]  
\[ \quad [<1,0,0>, <0,1,0>, <0,1,0>], \]  
\[ \} \]  

**Figure 2:** Example of an existing log filter and how it is modified after an update

It is assumed that this system consists of three nodes (which is indicated by three positions in each vector). In the first example (see figure 2), objects \(a\) and \(b\) have been updated independently of each other. This can be seen in the log filter by that there are two sets of
The Conflict Detection Algorithm

objects with their version vector sequences attached to them. This implies that there are two entries in the log filter.

In the second example, an update is made that reads objects \( a, b \) and \( c \) and write object \( c \). This is illustrated in the example by the \( \text{Update()} \) clause which is sent to all other nodes in the system. The \( \text{Update()} \) clause has the following format:

\[
\text{Update}(\{\text{read set}\}, [\text{Version vector for each object in the read set}], \{\text{write set}\})
\]

The transaction, on which this new update is based, has included objects which are already in the log filter. The objects \( a \) and \( b \) are in the read set of the new update. They have to be merged with the read set \( \{a, b, c\} \), so it is certain that every possible case of update is covered.

This also includes extending \( a \) and \( b \)'s version vector sequences with null vectors (these are indicated with \(<. . . . .>)\) and merging \( a \) and \( b \)'s version vector sequences with the sequence \([<1,0,0>, <0,1,0>, <0,1,0>]\) which is provided by the update. The null vectors indicate that the corresponding object has not been involved in that particular transaction. They are added to the version vector sequence to make it possible to compare the version vector sequences, version vector by version vector, with each other to determine whether the last updated sequence dominates every other sequence attached to that particular set of objects.

**Definition:** A sequence of version vectors \( V_1 \ldots V_n \) dominates another sequence \( W_1 \ldots W_n \) iff every version vector \( V_i \) in the sequence dominates the corresponding vector \( W_i \) in the other sequence.

In 2.5.2 it is defined how a version vector can dominate another version vector.

If the update sequence does not dominate and is not dominated by every other sequence a conflict is detected. If a conflict is detected the conflict resolution mechanism takes over the execution and resolves the conflict.

To illustrate what happens when a conflict occurs, two more examples follow.

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Update({a}, [&lt;1,0,0&gt;], {a})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 2</td>
<td>Update({a}, [&lt;0,1,0&gt;], {a})</td>
</tr>
</tbody>
</table>

**Figure 3:** Example of write-write conflict

The first example (in figure 3) illustrates a write-write conflict, i.e. both nodes have updated the same object. Node 1 reads object \( a \) and updates \( a \), and Node 2 does the same thing. None of the version vectors dominates the other, which indicates a conflict. Moreover, the objects of the conflicting version vectors is in the write set on both nodes, and this indicate that it is a write-write conflict.
In the second example (shown in figure 4), Node 1 (first row) reads objects $a$ and $b$ and updates object $a$. Node 2 also reads $a$ and $b$, but updates object $b$ instead. The new values are based on both object $a$ and $b$.

When the update made at Node 2 arrives at Node 1, a conflict is detected. At this point it can be established that Node 2’s vector for object $b$ dominates Node 1’s corresponding version vector. This is so because Node 2’s object $b$ has a greater version number. The problem is that Node 2’s version vector for object $a$ is dominated by Node 1’s version vector for object $a$ (object $a$ on node 1 has greater version number). It is now clear that the version vector sequence received from Node 2 is neither dominant nor dominated by Node 1’s version vector sequence (according to the definition of dominance above). The conclusion drawn from this is that the updated object in itself is not in conflict, but some other object is. Therefore this signals a read-write conflict. There will be a conflict on both Node 1 and Node 2. The node’s signals that their updated object is in conflict with some other object on some other node. In Node 1’s case this is object $a$ and in Node 2’s case it is object $b$.

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Update(${a,b}$, $[&lt;1,0,0&gt;,&lt;0,1,0&gt;]$, ${a}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 2</td>
<td>Update(${a,b}$, $[&lt;0,0,0&gt;,&lt;0,2,0&gt;]$, ${b}$)</td>
</tr>
</tbody>
</table>

**Figure 4:** Example of a read-write conflict
4 Problem description

Most of the existing replication protocols use serializability and mutual exclusion to uphold consistency among nodes in the network. In other words conflict avoidance is used to prevent conflicts from occurring. The requirement of strong serializability brings extensive message passing between nodes, so that all nodes are synchronized and aware of changes to data, which makes the transaction execution time unpredictable. This is inappropriate for use in a distributed real-time database system because in a real-time environment transactions must commit within bounded time to be sure that they meet their deadlines and thereby uphold predictability.

DeeDS is designed to allow delayed replication. This means that transactions commit locally before the updates are propagated to all other nodes in the system. This, in contrast to e.g., two-phase commit, implies that there may be inconsistencies in the system for short periods of time. When an update is propagated to other nodes the conflict detection algorithm compares the new update with previous updates to decide if there are conflicts between them. If conflicts are detected, the conflict resolution mechanism resolves them.

Lundström (1997) investigated how a bounded-delay replication protocol can be defined by considering different propagation methods, conflict detection algorithms and possible conflict resolution mechanisms suited for the demands of DeeDS. Lundström made a high-level design and developed algorithms, but no implementation has been made and thereby no testing or validation of Lundström’s solution has been carried out.

In 4.1, Lundström’s replication protocol is briefly described (see (Lundström, 1997) chapter 5 for details) and in 4.2 the problem is described and how this work will extend the work of Lundström.

4.1 Bounded-Delay replication protocol

The protocol is divided into three sub parts to simplify its construction (Lundström, 1997). The replication protocol consists of:

- A propagation method.
- A conflict detection algorithm.
- A conflict resolution mechanism.

In the following subsections these sub parts are briefly described.

4.1.1 Propagation protocol

In Lundström (1997) properties of a bounded propagation protocol are described. Lundström 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.”
Two protocols that support reliable broadcast are CSMA-DCR, which stands for Carrier Sense Multiple Access with Deterministic Collision Resolution and DOD-CSMA-CD, which stands for Deadline Oriented Deterministic CSMA-CD.

The CMSA-DCR protocol works like the CSMA-CD protocol (often referred to as Ethernet) except when a collision occurs. CSMA-CD uses the truncated binary exponential backoff for retransmission scheduling (Halsall, 1996). The CSMA-DCR, however, uses a binary tree search to deterministically resolve collision situations. The binary tree is based on the number of sources in the network (Forss, 1999).

The DOD-CSMA-CD protocol is similar to the CSMA-CD protocol (Forss, 1999). However, DOD-CSMA-CD uses a global synchronized clock to resolve collision situations deterministically (Forss, 1999).

Some assumptions about the environment are made. A faulty node will not make any updates and will therefore not affect the state of the database. The network is a local area network where partition failures are rare and therefore it is assumed that partition failures will not occur.

4.1.2 Conflict detection

The conflict detection algorithm is based on the version vector algorithm as described by Parker and Ramoz (1982). Lundström has adjusted it to fit in a distributed database system environment rather than a file system. The algorithm works with updates on objects in sets (which is how OBST stores data). One problem with this is at what granularity the version vectors should be applied. Four different granularity levels are identified: container level (a version vector for each set of objects), page level (one version vector for each database page), object level (one version vector for each object), and attribute level (one version vector for each attribute of an object). The algorithm is presented in pseudo code and with some brief description in Appendix A.

4.1.3 Conflict resolution

The conflict resolution mechanism in the replication protocol is based on forward recovery. That is, in case of a conflict occurring the database must be brought to a consistent state without making a roll-back. This is due to the fact that making a roll-back on the updates includes contacting the node which made the first update and asking it to perform the update again, which introduces non-determinism. In a real-time environment, this is not a feasible solution. Therefore, some mechanisms to make forward recovery are suggested (Lundström, 1997):

- **Ignore**: Do nothing. Used when a read-write conflict is detected.
- **Average**: The conflict is resolved by choosing the average value of both updates.
- **Time-stamp**: Each transaction is given a time-stamp which is used to determine which one of the two updates that is the latest and use that one.
- **Priority**: The nodes have priority values. In case of a conflict the update made by the node with the highest priority is chosen.
Problem description

4.2 The problem

The purpose of this project is to make an analysis and design of how the replication protocol proposed by Lundström can be implemented in DeeDS and make an implementation in the real system. This will make replication available in the implementation of the DeeDS prototype and makes it possible to determine whether Lundström’s suggestion for a replication protocol works in the real system.

The problem consists of both a theoretical part and a practical part.

The theoretical part involves three issues:

• Investigate at which granularity level the version vectors should be applied.
• Investigate how the internal data structures should be represented.
• Investigate how to implement the replication protocol.

The practical part involves two issues:

• Implementation of the replication protocol.
• Define and perform test cases for replication protocol.

The main problem is to design the replication protocol and investigate how it can be integrated with TDBM.

With the enhanced version vector algorithm the question arises of how to incorporate a log filter in tdbm. This is needed for the conflict detection and conflict resolution mechanisms. This in turn brings up the problem with data representation. The things that need to be investigated and carried out are summarized below:

• **Implementation of the protocol:** Investigate where in tdbm and how the implementation of the replication protocol can be done. This consists of three parts due to the fact that the protocol consists of three parts; a propagation method, a conflict detection algorithm, and a conflict resolution mechanism. The different parts bring different requirements to the implementation.

• **Data representation:** This involves investigating how the version vectors shall be represented and associated with a physical object. This leads to another important question and discussion: at which granularity should the version vectors be applied?

• **Log filter:** This is needed by the conflict detection algorithm to be able to determine which objects (at which nodes) that are in conflict. It must be investigated how the log filter can be implemented, i.e., high level design and how data should be represented and how entries should be handled.

• **Testing and verification:** When the implementation is done there has to be some testing to verify whether the protocol is predictable and efficient. Regarding predictability, a decision on which criteria to use must be made. Test cases must be defined.

The practical part of the problem aims at defining a software design that can be implemented to prove the feasibility of Lundström’s work. The theoretical part consist mostly of discussions about the data structures and granularity of version vectors, correctness criteria, etc. This is important for the implementation.
4.3 Limitations of this project

The project will focus on the design of the replication protocol proposed by Lundström. An analysis of the conflict detection algorithm to state its behaviour over time is made, but no further test to measure the actual outcome in practice is made.

Moreover, concerning the propagation of updates, it is assumed that communication links are available and that each node will receive the updates only once. The initialization of the network and communication links is assumed to be carried out by the underlying operating system, in this case OSE Delta.

The conflict resolution is a quite complex problem and to investigate this thoroughly a separate project has to be carried out so this can be specified satisfactorily. The conflict resolution is left as future work.
In this chapter, a review is given of the design decisions made for the implementation of the propagation and reception mechanisms and the conflict detection algorithm described in Lundström’s work. Different ways are investigated for how to solve certain problems related to the implementation. This includes high level design of the replication protocol and data representation.

In 5.1 an overview is given of the design of the replication protocol and how it is incorporated in the existing system. The propagation and reception model is described in 5.2. In the remaining sections the implementation related material is discussed in detail, such as how data structures shall be represented.

5.1 Design Overview

The replication protocol is integrated in tdbm as an extension module. This is illustrated in figure 5.

![Diagram of Replicator as a module attached to TDBM](image)

The applications run as individual processes and use tdbm to store and fetch objects from one or more database files (tdbm files). Tdbm uses the Replicator to log and replicate the transactions to the remote nodes. The Replicator uses the operating system, in this case OSE Delta, to propagate the updates via the network. It also receives updates from the network, and merges these with the log filter and performs conflict detection (according to the algorithm, see chapter 3). If no conflict was detected the replicator updates the local database with these updates.

The Replicator will be broken down into smaller modules.

- Logger
- Propagator
- Replicator
Approach

- Version Vector Handler (VVHandler)
- Log filter

Figure 6 shows the modules and their relations to each other and the environment (tdbm and OSE Delta).

![Figure 6: Overview of the model](image)

The Logger is called during a transaction to create a log of the current transaction. The log is then propagated to the other nodes via the Propagator. The log is also sent to the Replicator so that the transaction can be inserted in the log filter on the node.

The propagation mechanism (the propagator) and the reception mechanism (the replicator) must be able to work concurrently so that replication-related work, such as logging and replication transactions, doesn’t interfere with local application related transactions. To achieve this they run as two independent processes.

The Propagator has only one responsibility: to broadcast the updates made locally to all other nodes. Updates are sent to the propagator, which then performs the broadcast. The propagator run as an independent process and takes over the propagation work from the process currently running a transaction. This frees tdbm as soon as possible after a transaction has committed. There is no broadcast primitive available in OSE Delta, so the broadcast must be made using point to point communication. OSE Delta uses a link handler to communicate between nodes. This mechanism can be used transparently with the send and receive primitives.

The Replicator has three responsibilities:
1. Receive updates from remote nodes.
2. Perform conflict detection and update the database if no conflict was detected.
3. If a conflict was detected: notify the conflict resolution mechanism.

When a new update is received, the Replicator locks the objects in the database that are going to be affected by the new update. It then verifies the log filter with the new update.
no conflict was detected the update is committed. The objects are locked to prevent them from being changed by a local transaction while the log filter is being updated. If the objects are not locked the new update could conflict with some local transaction. If a conflict is detected in the Log Filter the locks on the objects are released.

The Version Vector Handler is used by the Log filter to build version vector sequences. It is a way to gather the creation and deletion of version vectors and version vector sequences in one place and thereby make these operations in a structured and controlled manner.

The Log filter is used to store version vector sequences and supply the Replicator with the result from conflict detection.

5.2 Propagation of updates

The propagation of updates is a problem in the sense of what information must be propagated. The propagation method must, of course, propagate any locally committed update to all other nodes in the system. It is specified that no transactions are replicated, only the updates (Lundström, 1997). In other words, the sending node must make it possible for all other nodes to locate the updated object in their own databases and modify that object so that it contains the same value as everywhere else in the system. The questions are how this can be done and what information should be propagated?

Lundström (1997) has given a formal description of what the update information shall contain and it is represented in the following way:

\[ \text{Update} \{a,b,c\}, \{<1,0,0>, <0,1,0>, <1,0,0>\}, \{b\}, 12 \]

The first set is the read set. this is followed by the version vectors for the objects in the read set. The third parameter is the write set (a subset of the read set) and the last parameter show the new value of the object after the update. This leads to that there are three things about the update message that must be determined.

First of all, in the above description the names of the objects are used, but in the real system this must be the object identifiers to make it possible to find the correct objects in the database. The object identifier is used as the key for the object in the database. Second, it must be determined how the information about the actual update should be propagated. There are two variants of this discussed here. i) Only the parts of the objects that have been changed are propagated or ii) the entire object is propagated and replaces the old one on the receiving node.

With the first propagation alternative some extra information must be added, the offset to the position in the object where the update is to be put and the length of the block that contains the update. This must then be used to put the piece of data in the right place. This approach can be complicated. It is difficult to determine which part of the object that has been updated.

The second propagation alternative is simpler because the complete object is propagated and then put in the database using tdbm’s transaction and hashing mechanism.

In this project, the second propagation alternative is the approach chosen, since it is simpler and easier to realize.
The third thing to consider is the version vectors. The version vectors must be supplied in the update message so that they can be compared to the other version vectors on the other nodes. The version vectors must be fetched from their corresponding object and placed in the update message. This is a quite simple operation and is not considered as a problem. Is is done by reading the objects and extracting their version vectors.

As mentioned in 5.1 above, the log created during a transaction is propagated. The log should conform to the format described above.

5.2.1 Modifications to transaction handling in tdbm

The execution of a transaction in tdbm follows the schema described by Elmasri and Navathe (1994, figure 17.4 at page 535). However, the execution of a transaction in tdbm must be extended with calls to the Replicator so that the log, which is used for propagation, is created. Also, the transaction used to perform an update to the database based on updates received from remote nodes use the same interface as application transactions. However, these update transactions should not be logged and propagated. Therefore, it must be possible to distinguish the different transactions.

5.3 Granularity

Before the data structures can be described, it has to be determined at what granularity the version vectors shall be applied.

Four levels of granularity have been identified:

- **Container level**: One version vector for each container (set of objects).
- **Page level**: One version vector for each disk page.
- **Object level**: One version vector for each object stored in the database.
- **Attribute level**: One version vector for each attribute of an object.

The following subsections contain a discussion of the advantages and disadvantages with each approach.

5.3.1 Container level

A container groups objects that make up a logical unit (Casais, 1992). At container level (which can be said to be the highest level) there will be very few version vectors which saves storage space, i.e. minimize the overhead imposed by the version vectors. There is one disadvantage with this approach. When two (or more) transactions on different nodes update different objects within the same container, this will indicate a possible conflict, since the container has been modified. However, there are no conflicts since the transaction updates different objects in the container. This can possibly be solved, but it is not clear at what cost in the sense of computational overhead. Checking for conflicts will always be performed whether there is a conflict or not. When a conflict is detected there has to be an extensive work to determine which objects caused the algorithm to detect the conflict, and thereafter determine whether there really is a conflict.
5.3.2 Page level

When applying the version vectors at page level (Elmasri & Navathe, 1994) there are some issues that must be fulfilled by the page management mechanism in tdbm (Brachman & Neufeld, 1992). Two pages stored at two different nodes have to be equivalent. That is: one object must reside on the same page on the different nodes and the object must be stored at the same position on the page at both nodes. If not, there are problems to find the objects in conflict, and there can be objects that are in conflict but the conflict is not detected. The version vector only indicates that there is a conflict on the page, not which objects that are involved in the conflict.

Some kind of directory must be kept over objects on the page. There is in fact such a directory already available in tdbm (Brachman & Neufeld, 1992). Nevertheless this method requires mechanisms to decide which objects that are in conflict before the conflict can be resolved.

5.3.3 Object level

On object level, a version vector is applied to each physical object in the database. This increases the amount of storage space needed, and that is a disadvantage. The advantage is that there cannot be any doubt about which objects that are in conflict. This saves the extra work that must be performed to identify the conflicting objects when version vectors are used at page or container level.

5.3.4 Attribute level

The last alternative is to apply a version vector to each attribute of an object, this however not possible in this case. At tdbm level the objects are not viewed as an abstract data type, with attributes and methods. It is viewed and handled as a Binary Large Object (BLOB) i.e., a chunk of binary data with no semantic meaning at all (Elmasri & Navathe, 1994). Therefore, it is very difficult to distinguish an object’s particular attributes. Therefore, to apply version vectors at an attribute level is not useful.

5.4 Data representation

For data representation of objects there are two things that must be investigated: i) how should the version vectors be stored and ii) how can the log filter be physically represented? In the following two sections these topics are discussed.

5.4.1 Representation of version vectors

Two different ways to store the version vectors for each object have been identified. In this section, these are reviewed and compared.

One method is to attach the version vector to the object when it is stored in the database i.e., the value in the database is exchanged with a value that contains both the object and its version vector. The effect is that this extra information must be handled every time the object is used. When an object is accessed, the version vector must be removed before the
Approach

An object can be used and modified. Also, when only the version vector is to be accessed the whole object must be fetched from the database.

When an object is updated the version vector must also be updated before the object is stored in the database.

There will be situations when only the version vector is of interest and others when only the actual object is of interest. This, however, is not such a serious problem in the sense of overhead. On the other hand, all information is always available when handling the updates and propagations.

Another solution is to store the version vectors in a database of their own. This database should only contain the version vectors and information about which objects they are attached to. This scheme is shown in figure 8. The meta data is, in this case, the version vector and the information about which object it is attached to.

With this method, it is possible to locate a version vector for an object directly by using tdbm’s hashing mechanism. In the previous method (see figure 7) it is possible to use hashing to find the version vector, but the object must also be handled.

Another advantage is that the version vector and the actual value of an object can be accessed and manipulated separately. The disadvantage is that there is one extra database that must be maintained.

The first approach is the one chosen. By attaching the version vector to the object it is possible to avoid having an extra database that must be maintained. Also, the object is often
used together with its version vector and therefore by choosing this approach it is possible
to get both the object and the version vector with just one access to the database. The dis-
advantage is that this extra information must be handled every time the object is accessed.

**Representation**

The version vectors are represented as an array where each position represents a node
number. With this representation it is possible to see an object’s version on each node. See
chapter 3 for a description of the conflict detection algorithm and the version vectors.

With this solution, however, requires knowledge of an upper bound on the number of nods
in the distributed system when a database is created. It might be troublesome to add nodes
dynamically. It is then necessary to update every object in the database with a new version
vector. It is possible to make this solution scalable however, by letting the version vector
array contain more elements than there are nodes at start-up. When initializing the system
a boundary is set on the number of elements in the version vector array that can be used.
The boundary equals the number of nodes in the system. When there is a need to add more
nodes the boundary can be increased. Of course, there is always an upper bound for what
is a reasonable length of the version vector array. This upper bound may vary from system
to system. The solution is, however, scalable to a certain extent and it is the one chosen in
this implementation.

**5.5 Log filter design**

The log filter is accessed and updated every time an update on a database object is per-
formed. It is required that handling of the log filter is efficient. Two approaches are dis-
cussed here for how this can be realized.

**5.5.1 Using a database file for log filter information**

One alternative is to store version vector sequences in a database of their own. This data-
base should also contain information about which objects they belong to. It is then possible
to use tdbm’s hashing mechanism to determine whether the actual read set is already
contained in the log filter and if so, extract the corresponding version vector sequences.
This operation is performed, according to the algorithm, every time an update is received.
The use of hashing to locate the version vector sequences is of course a big improvement
compared to a linear search through a list of objects, which is another alternative. The dis-
advantage is that there is one extra database that must be maintained and modification of
version vector sequences, in the sense of adding null vectors, is complicated. The
sequence must be fetched from the database, modified and stored back again. This leads to
processing overhead.

**5.5.2 Using lists for log filter information**

Another method is to use a list (Aho, Hopcroft & Ullman, 1983) to store the sequences
and information about to which set of objects they belong. A disadvantage is that it takes
time to find an entry in an unsorted list due to the fact that the search is based on linear
search. For example, when going trough the read set and trying to find the corresponding
entry in the log filter.
Approach

Nevertheless, if a list is used there are two different ways the version vector sequences can be represented in the log filter, with linked lists or arrays (Aho, Hopcroft & Ullman, 1983). With the linked list, each link will contain one version vector and the list constitute a version vector sequence. When using arrays, one array would represent a version vector sequence.

An advantage with lists is that it is easy to make them grow dynamically. It is easy to insert and add elements in a linked list. When inserting and removing elements in an array, the elements have to be moved accordingly (Aho, Hopcroft & Ullman, 1983).

5.5.3 Discussion result

It seems like the approach using lists seems dynamic and scalable compared to using a database file (Elmasri & Navathe, 1994; Aho, Hopcroft & Ullman, 1983). Most programming languages support operations for dynamic allocation of data segments in memory. This seems to be a more appropriate method to apply on the log filter implementation than working with entries with dynamic length in the database.

5.6 Software test

To test and verify the correctness of the replication protocol, eight test cases have been defined. These are reviewed in this section. These test cases are mainly designed to verify a correct behaviour of the replication protocol. They can also be used for regression tests of DeeDS. Regression testing is defined by Pressman (1994) as “Repeating past tests to ensure that modifications have not introduced faults into previously operational software”. The tests are then used during development of DeeDS to verify that changes to the system did not affect the system in a way that makes it work improperly.

5.6.1 The test process

The test process is based on the “black box” approach (Pressman, 1994). With this technique the tests are conducted on the interface of the software system, or in this case, the software module TDBM with the Replicator module attached to it. The approach is chosen to fit in with the research project Testing of Event-Triggered Real-Time Systems (TETReS) of the Distributed real-time systems research group at the university of Skövde (Andler, Birgisson & Mellin, 1999).

The test cases are run by a script (written in Perl). Input to the script is a test program, which implements the flow specified in the test case, and a file with expected outcome of the test. The test is run and the actual outcome is compared to the expected outcome. If the two outputs match, this is a successful test. The test script is run from the console of UNIX shell and the outputs are text based outputs to the console.

5.6.2 Test cases

In this section the test cases are reviewed with three statements: purpose, stimuli and expected outcome. In the test cases the term entry has been used to refer to something that is stored and manipulated in the database. The term object has been used earlier because
tdbm is used as storage manager for OBST, which is an “object management system”. Entry is, however, more general and is therefore used in the test case descriptions.

Test 1 - Store Entry

**Purpose:** To test that the Replication module does not cause values of stored entries to become corrupted.

**Stimuli:**
- A local node stores a new Entry in a database. (Entry shall be a parameter, as well as the database)
- The same local node reads the newly stored Entry and prints it to standard output.

**Expected outcome:** The stored Entry and the Entry printed to standard output shall be the same.

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 a database. (Entry shall be a parameter, as well as the database)
- The same local node reads the newly stored Entry and prints it to standard output.

**Expected outcome:** The Entry, that which the object is updated to, and the Entry printed to standard output shall be the same.

Test 3 - Store Entry with Version Vector

**Purpose:** To test that the version vector associated with an Entry is handled correctly when a new Entry is created. When a new Entry is created in the database a new version vector shall also be created and associated with the new Entry.

**Stimuli:**
- A local node stores a new Entry in an existing database. (Entry shall be a parameter, as well as the database)
- The same local node reads the newly stored Entry along with its version vector and prints it to standard output.

**Expected outcome:** The stored Entry and the Entry printed to standard output shall be the same. Also a version vector shall be displayed that hold as many fields as there are nodes.
in the system and the field in the vector, corresponding to the actual node, shall present a “1” to show that the object has been updated (created) on this node.

**Test 4 - Update Entry and Version Vector**

**Purpose:** To test that the version vector associated with an object is handled correctly when an existing object is updated. When an object is updated in the database, the field of its associated version vector that correspond to the current node shall be incremented.

**Stimuli:**
- A local node updates an existing Entry in a database. (Entry shall be a parameter, as well as the database)
- The same local node reads the newly stored Entry along with its version vector and prints it to standard output.

**Expected outcome:** The updated Entry and the Entry printed to standard output shall be the same. Also a version vector shall be displayed that hold as many fields as there are nodes in the system and the field in the vector, corresponding to the actual node, shall be incremented by one to show that the object has been updated on this node.

**Test 5 - Check Propagate Message at sending node**

**Purpose:** To test that the propagation message is correctly assembled related to the transaction performed.

**Stimuli:**
- A node stores a new Entry in a database. (Entry shall be a parameter, as well as the database)
- Compare the prepared message for propagation (locally) with an expected outcome.

**Expected outcome:** The resulting propagation message shall match the expected propagation message for a given transaction.

**Test 6 - Check Propagate Message at receiving node**

**Purpose:** To test that a propagated message is correctly transferred and interpreted at a receiving node.

**Stimuli:**
- A node stores a new Entry in a database. (Entry shall be a parameter, as well as the database)
- Another node receives this new Entry as a propagation message.
- Compare this message with an expected outcome.
**Expected outcome:** The received propagation message shall match the expected propagation message for a given transaction.

**Test 7 - Replication (create new Entry) without conflicts**

**Purpose:** To create a new entry in a remote database by using the propagation and replication mechanisms.

**Stimuli:**
- A node (n1) creates a new Entry in a database. (Entry shall be a parameter, as well as the database)
- Another node (n2) receives this new Entry as a propagation message.
- Create a local (for n2) version of Entry with the information found in the propagation message.
- Read the Entry from the receiving node's local database.
- Compare the Entry from n1 with the Entry from n2. (They shall match)

**Expected outcome:** The Entry shall be created on both nodes and shall be identical to each other.

**Test 8 - Replication (update existing Entry) without conflicts**

**Purpose:** To update an Entry in a remote database by using the propagation and replication mechanisms.

**Stimuli:**
- A node (n1) updates an existing Entry in a database. (Entry shall be a parameter, as well as the database)
- Another node (n2) receives this update as a propagation message.
- Update n2's local Entry with the information found in the propagation message.
- Read the Entry from the receiving node's local database.
- Compare the Entry from n1 with the Entry from n2. (They shall match)

**Expected outcome:** The Entry shall be updated on both nodes and shall be identical to each other.

**5.7 Summary and conclusions**

In this chapter, possible solutions to different implementation related problems have been investigated.
Approach

It has been argued that version vectors shall be applied to objects and not containers or pages.

The version vectors are attached to the objects in the database. With this, the use of an extra database is avoided. In this case, not much is saved by using a separate database for the version vectors, because objects and their version vectors are often used together. When putting all data (object and version vector) in the database in this manner, all information can be fetched using only one access to the database.

Furthermore it has been decided that the log filter should be implemented using lists. The other alternative would be to use a database file. Handling the filter in main memory, separated from the database, is considered more efficient.

The propagation of updates is made after local commit. The information that must be sent to the other nodes is the read set, the write set, version vector for each object in the read set and the new updated object. The complete object will be propagated.

Eight test cases for test and verification of the replication protocol have been defined and described with purposes, stimuli to run the test case and expected outcome. The test cases can be run independently, but also in a sequence.
6 Results

This section will expand the discussion from chapter 5 with more details on the implementation. Which modules that have been implemented and choice of data representations are reviewed.

In chapter 5 a more detailed design of the replication protocol presented in chapter 4.1 is given. The design consists of five software modules. The modules that are implemented are the Logger and Version Vector Handler. The Replicator and Propagator are partly implemented, but not enough to fully test propagation.

The Logfilter module and the enhanced conflict detection algorithm have not been implemented. This affects the result in the way that the replication protocol can not be tested as a whole, and the log filter and conflict detection can not be tested or measured. Therefore a discussion about the complexity of the algorithm and a theoretical measure of growth of the log filter over time is given in section 6.3 below. The tests are reviewed in section 6.2.

6.1 Review of the Logger implementation

As stated in chapter 5 the first module of the Replicator module that is activated when a transaction is started is the Logger. The implementation of the TDBM-interface is extended with hooks that call the logger on several occasions during a transaction. When a transaction is started by an application a new log file is created. A log transaction is also started which handles storage of log values to the log file. The logger uses TDBM in itself to manage the log database. A database is used because of the implicit error handling and error recovery mechanism that is available in TDBM. The logger is the only module that is implemented in its whole.

Each object that is stored in the log has a special format. This format is described in figure 9 below.

<table>
<thead>
<tr>
<th>Header</th>
<th>Key</th>
<th>Version Vector</th>
<th>Value</th>
</tr>
</thead>
</table>

**Figure 9**: The format for the value that is stored in the log file.

As stated in section 5.3.4 the values (objects) stored in TDBM are BLOBs. That is, they do not have any semantic meaning at all in themselves. The ladleful BLOB is built from four parts.

- **header**: The header is a data structure which contains information about operation type (read or write), pathname (the pathname of the database to which the logged value belongs), key descriptor, key size, value descriptor and value size (the four last values are TDBM-specific internal data structures and is stored so that they can be used when updating the value on a remote node).
Results

- **key**: The key of the value that is being logged. The key is used to locate the value in the database of a remote node to which the value is replicated.
- **version vector**: The version vector of the value to be logged.
- **value**: The actual value.

The logger is activated when the following operations are executed by an application:

- **begin**: When a begin transaction call is made to TDBM a new log database is created by the logger. A new transaction is also created which is used to manage the log database.
- **fetch/store**: The value is added to the wildfowl. This is done by a store on the log file. The logger will call TDBM to store a new value in the log. The ladlefuls follows the format described above. If the operation executed is a fetch the value part of the BLOB is left out. The actual value is only interesting in case of a write when performing error recovery. The value is left out to save bandwidth in case of a low capacity communication link in a distributed system.
- **commit**: When the transaction is committed the log transaction is also committed, the log is closed and forwarded to the Propagator and Replicator on the local node.

The output of the Logger is *log message* which can be sent to remote nodes via the Propagator. The propagation is partly implemented but not enough to pass a test.

6.2 Test results

Only test cases 1-3 have been run. They ended with the expected result as described in section 5.6.2. The other test cases depends on fully developed propagation and conflict detection mechanisms.

That the test runs turned out to be successful, in the sense that they conformed to expected output, which gives an indication that the implementation is correct. However, the test cases are focused on regression testing which is applicable the most when integrating a finished system to verify that the different modules doesn’t affect each other so that they don’t behave as expected.

With test cases defined for unit testing (Pressman, 1994) a better indication and a better measure of the correctness of the implementation should be acquired.

6.3 Discussion

6.3.1 Implementation

In section 5.3, several granularities at which the version vectors can be applied was discussed. From that it is decided to apply the version vectors at the object level. This granularity is chosen since conflicts arise between two or more objects and not between pages or containers. If the attribute level had been used, it might have been necessary to move the conflict detection and resolution up one abstraction level to OBST. This because TDBM does not have meta data to understand the values (objects) which it stores.
6.3.2 Test

The implementation of the replication protocol has not been finished. If it was some test cases to measure efficiency of that implementation should be defined. Examples of such test cases would be:

- Measure how the version vector sequences scale over time
- Measure how the log filter scales over time
- Verify if the replication of data has any effect on the performance of the database. The update operations may be slowed down if there are large amounts of data to be replicated (Elmasri & Navathe, 1994).

Point 3 refers to a statement in (Elmasri & Navathe, 1994). This statement refers to systems with pessimistic concurrency control where full replication is used. DeeDS also uses a fully replicated database (Andler et. al., 1995), even though the consistency constraints are relaxed. It may, however, be relevant to investigate the effects of replication in a system with eventual consistency. The replicated updates must be handled in parallel with local transactions. The local transactions use pessimistic concurrency control (Andler et. al., 1995).

Also, running all test cases in sequence would be an interesting test so that is clear that an update to a database fully works.

The test cases are also written with the regression tests of DeeDS in mind, which doesn’t make them fully suitable for unit testing (Pressman, 1994). The regression tests are more suitable to verify that a module doesn’t behave incorrectly after changes to the other parts of the system.

Unit testing (Pressman, 1994) aims to actively find and eliminate errors in the software during development. Therefore there should be some more test cases defined that focus on unit testing to acquire a better measure of the correctness of the implementation.

6.3.3 Complexity of the algorithm

The log filter should be implemented using linked lists. The log filter will grow constantly while there are transactions performed in the system. In the following discussion the complexity of the algorithm, and its relation to the log filter will be discussed.

As stated in the problem description (see chapter 4) there should be tests to verify the predictability and feasibility of the algorithm. However due to the fact that the implementation is not completely done, a theoretical analysis of the behaviour of the conflict detection algorithm is made here instead.

To be able to estimate the execution time and predictability of the algorithm and its implementation, the complexity of the different parts has to be investigated.

The critical parts of the algorithm are the UNION operation, the sorting of sets, and the comparisons of version vector sequences when detecting conflicts. Each of these is exe-
In Aho et. al. (1974) a UNION operation is described which take two disjoint sets and returns their union. This operation has a complexity of $O(n)$. In the conflict detection algorithm, the sets are not disjoint which implies that before an element can be merged with another set, it has to be determined if it is already in that set. If the union operation is expressed like $\text{UNION}(A, B, C)$, where $A$, $B$, $C$ are sets and $A$ and $B$ are merged to a set $C$, the number of comparisons that have to be made is $n \times m$. It is a linear search.

The sorting operation is carried out after the sets from the log filter and the read set of the particular update are merged using a UNION operation. It is necessary that the sets are sorted in the implementation when determining if and where a version vector should be extended with null vectors. For this, the quick sort algorithm is used. According to Aho et. al. (1974) the quick sort algorithm has a complexity of $O(n \log n)$, where $n$ is the number of elements to be sorted.

When comparing the version vector sequences, each element in a version vector is compared to the corresponding element in the version vector in the other version vector sequence. This is repeated for every vector in the sequence and for each sequence in the log filter. The number of sequences in the log filter is denoted by $l$.

To determine if there is a conflict or not, it has to be checked whether the version vector sequence of the update dominates all other version vector sequences, or is dominated by every other sequence. Put another way, the sequences must be sorted. If this holds, there are no conflicts.

If the version vector sequence in the update dominates all other version vector sequences, there is no need to compare the other version vector sequences with the new one. If not, each version vector sequence that is not dominated must be compared to the new version vector sequence to determine if that is the one that is dominant of the two.

In the worst case this leads to that there has to be $2(n \times m \times l)$ comparisons to determine whether there are a conflict or not. The complexity is therefore $O(n \times m \times l)$. This is explained with an example.
Consider figure 10. Again \( n \) is the number of nodes in the system (and thereby the number of elements in each vector) and \( m \) is the number of objects in the update. The number of sequences in the log filter is determined by \( l \). To compare one sequence with another, \( n \times m \) comparisons are needed. To compare a sequence with every other there has to be \( n \times m \times l \) comparisons.

If it is found that a version vector does not dominate another version vector, this shall be compared to the first one (as stated above). This leads to that there are \((n \times m \times l) + (n \times m \times l)\). If this has to be done for every sequence the result would be \((n \times m \times l) + (n \times m \times l) = nml + nml = 2nml\). This is implies that the number of comparisons grows with as a linear function.

The above examples shows the result of one update, and it is only the number of comparisons made that is calculated. Further, there are some more operations executed for every step taken in the lists.

The number of comparisons that is required depends on the number of nodes in the system, the number of objects in each update and the number of sequences in the log filter. The number of nodes is constant during the lifetime of the system and the number of objects in each update can not be assumed to be constant but it would probably not increase or decrease by large amounts.

The only thing that grows constantly is the number of sequences in the log filter. Therefore the log filter must be drained after a specified time as described by Lundström. This time is \( 2 \times T_{rep} \) where \( T_{rep} \) is the time it takes for an update to replicated, i.e. propagation, conflict detection and conflict resolution (Lundström, 1997). By doing this the growth of the log filter can be made predictable.
7 Conclusions

To revisit the objectives, this project is based on Johan Lundström’s work on how to get a bounded-delay replication in DeeDS.

A design of the replication protocol and its division into software modules has been carried out. Most of the implementation is done on the logging part and the handling of version vectors in the database.

Different ways have been investigated to determine how the enhanced conflict detection algorithm described by Lundström (1997) can be implemented. It is investigated how data can be represented and the conclusion drawn from that is that the log filter should be represented using a linked list, and each object has its version vector attached to it.

An analysis has been carried out to estimate the number of operations that has to be carried out, to be able to estimate the execution time and predictability. The result of that analysis is that the algorithm grows with the complexity of $O(nm_l)$ where $n$ is the number of nodes in the system (and thereby the length of the version vector), $m$ is maximum length of a version vector sequence and $l$ is the number of updates made.

The feasibility of the conflict detection algorithm is dependent on the applications that use DeeDS. The factors such as number of objects, number of nodes and number of updates made per second is individual for each application.

Some of the defined test cases have been run. They produced the expected outcome, which was successful. To make a more thorough and rigorous verification some test cases should be defined that addressed unit testing.

7.1 Contributions

A software design of a replication protocol for use in general, but for DeeDS in particular, to replicate data among nodes in a distributed real-time system has been defined. The design is partly implemented and tested. With the design of the replication protocol the implementation of a replication mechanism in DeeDS has been brought one step closer.

The replication protocol uses the enhanced version vector algorithm for conflict detection. This raises the question of how version vectors shall be handled in the system. An argumentation about at what granularity the version vectors should be applied has been performed. A discussion of a implementation related design of the log filter as it is described in Lundström (1999) has been done, and an analysis of the complexity of the enhanced version vector algorithm has been performed.

Test cases have been defined to make it possible to easily regression test the replication protocol when it is integrated with DeeDS.

7.2 Future work

The implementation of the replication protocol is not completely done. Only the logging part and parts of the propagation mechanism of the protocol have been implemented. A future work would be to complete the implementation with a log filter implementation and
Conclusions

conflict resolution. With a complete implementation, a more thorough investigation of the behaviour of the replication mechanism can be performed. A complete implementation includes implementation of the handling of the log filter and the conflict detection algorithm as described in chapter 3. With this, empirical tests and measurements can be performed to measure the performance and feasibility in a fully distributed environment. If that turns out well this protocol may be a well suited protocol for distributed real-time database systems based on eventual consistency.

Testing the algorithm for efficiency can be achieved when the implementation is complete with propagation of updates and running on the target system. Then it is possible to test if it is feasible enough for the user applications that will use DeeDS.
Conclusions

Acknowledgments

There are some people I would like to thank for their support during the work.
First of all Joakim Eriksson, my supervisor, for his support, help, and for always being available and lending me some of his time to discuss problems and ideas.
Johan Lundström for leading me into this work.
Prof. Sten Andler for his support on special issues concerning Johan Lundström's work.
Prof. Sten Andler, Mats Lundin, and Jonas Mellin for reading my report and giving valuable comments and suggestions.
Sanny Gustavsson and Marcus Brohede for discussions concerning the implementation and for all help regarding the testcases and for pushing me on.
Conclusions

References


Conclusions


Appendix A: Enhanced version vector algorithm

In this appendix, the enhanced version vector algorithm used as the conflict detection algorithm is shown as it is described in Lundström (Lundström, 1997). This algorithm is based on Parker and Ramos’ (1982) version vector algorithm.

1. Initially, $LF = \emptyset$

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

   3. if $S$ is contained in some set $S' = obj_1 \ldots obj_n$ in $LF$ already, attach the version vector sequence $[\ldots, <v^1_j>, \ldots, <v^m_j>, \ldots] \to S'$ where $<v^i_j>$ is the version vector for object $obj^i_j$ (the value $j$ is contained in the OID for the object), and the $\ldots$ indicates that null vectors should be used.

4. If $S$ is not already contained in $LF$, incorporate $S$ into $LF$ using a UNION-FIND algorithm [PR82]. The algorithm is as follows:

   $S_T := \emptyset$

   for $obj$ in $S$ do

       begin

       $S_{obj} := \text{FIND}(obj)$

       if ($S_{obj} = \emptyset$) then ($S_{obj} = obj$) // It is a new object

       UNION($S_{obj}$, $S_T$, $S_T$)

   end

   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 $obj$.

5. After an update sequence has been incorporated, the sequence should be checked for conflict by executing the following:

   $S := \text{FIND}(obj)$

   if incompatible($S$)

   then if $S(\text{WRITE}_\text{SET}) \supseteq S_{\log\text{filter}}(\text{WRITE}_\text{SET})$

       return (WW conflict)

   else return (RW Conflict)

   else return (OK)