HÖGSKOLAN SKÖVDE

School of Humanities and Informatics
Examensarbete i datavetenskap 20p
(HS-IKI-MD-05-105)
D-level
Spring 2005

Conflict Detection in DeeDS

Markus Hoffmann

Conflict Detection in DeeDS
Submitted by Markus Hoffmann to Högskolan Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the School of Humanities and Informatics.

September 2005

I hereby certify that all material in this dissertation that is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.

Signature: _______________________________________

Supervisor: Sanny Gustavsson

Conflict Detection in DeeDS

Markus Hoffmann

Abstract

In distributed database systems, immediate global consistency of replicated data can be achieved by distributed commit protocols that are typically unpredictable. If real-time characteristics are necessary, such unpredictability has to be avoided. In a distributed real-time database, optimistic replication can be used to avoid unpredictable delays by allowing transactions to commit locally. The update of other nodes is performed as soon as possible. If optimistic replication is used, conflicts may occur since data can be changed locally without synchronously informing other nodes. To detect these conflicts, this thesis introduces a conflict detection approach for DeeDS, a distributed, active real-time database that supports a dynamic node set. A comparison of existing conflict detection approaches is performed, and it is found that the dynamic version vector approach is the best fitting approach. The main reason is that it can handle a dynamic node set with a minimum of additional conflict detection data. To show the realization of the approach in DeeDS, dynamic version vectors have been implemented. Additionally, conflict management in DeeDS is redesigned to allow separation of conflict detection and conflict resolution. This makes the software architecture more flexible and is a first step towards application specific conflict resolution.

Keywords: conflict detection, dynamic node set, DeeDS, dynamic version vectors, Bayou version vectors, Hash Histories, version stamps

Contents

1.Introduction

Today’s computer networks are growing rapidly in size because even small devices are connected with each other to exchange information. Processes controlled by computer systems are becoming more complex and there is often a need of communication between different parts of a system. An assembly line in a factory for example where many robots work together to produce a car needs a communication system so that the robots can exchange status reports to the controller software but also to other robots to signal them that a new item is ready and can be passed over. But not only the size of networks is changing. Wireless techniques like Bluetooth, Wireless LAN or UMTS make it possible to create dynamic networks in which nodes are not anymore statically connected. Ad hoc networks in which all nodes are equal and communicating directly with each other without a fixed infrastructure offer solutions for many new scenarios. In some of these scenarios data has to be stored and fetched in time. In the area of air traffic control for example approaches are discussed to give airplanes more control about the navigation so that they become autonomous. The long term goal that shall be reached is called free flight. *Free Flight* is defined as a “safe and efficient flight operating capability under instrument flight rules in which [pilots] have the freedom to select their path and speed in real time” by Task Force 3 of RTCA Inc., Washington, D.C. (Perry, 1997). By giving the airplanes more autonomy the number of flying airplanes could be increased to satisfy the increasing number of flight passengers. In such a scenario every airplane has to observe other planes around it via radar for example, to set the right course. Collected data like the position of other airplanes has to be accessed in real-time by applications like the collision detection because otherwise accidents can occur. To guarantee the timeliness of data access distributed real-time databases could be used. The advantage of using a distributed database instead of a not distributed one would be that airplanes see more than only their radar radius. They also get informed about the flight routes of other airplanes that are currently far away. This information could be used to plan the own flight path in a much better way because areas with a high airplane density could be detected in advance and so avoided. Additionally other information like thunderstorm warnings from other airplanes would be available for all.
At the University of Skövde the DRTS group is currently developing a prototype of a distributed real-time database called DeeDS (Distributed activE real-timE Database System). In this prototype optimistic replication is used for distributing the data in the network to avoid unpredictable waiting times. This means that updates to the database can be committed on a single node without immediately informing other nodes in the system. Inconsistencies might arise if replicas are changed that don’t include all former updates performed on replicas of the same object. Concurrently in this context means that the replicas were updated without including all previous updates. As a result of this different replicas of the same object include different values, making the database state globally inconsistent. To avoid this, conflict detection and resolution are needed to restore a globally consistent database state by detecting and resolving (e.g. merging) conflicting replica versions. This thesis describes techniques for conflict detection in environments with a dynamic node set. For conflict detection versioning approaches are used which in the first approaches could only work in a static environment in which the amount of nodes is not changing. The version vector approach of Parker & Ramos (1982) is such a static approach. Within the scope of this thesis the conflict detection based on the enhanced version vector approach by Lundström (1997) and conflict resolution in DeeDS has been redesigned to allow dynamic node sets. For this the dynamic version vector approach (Ratner et al., 1997) that can handle a dynamic node set has been implemented.

1.1. Thesis outline

The next chapter contains some basic information about replication models, consistency classes and versioning approaches. It also includes a short introduction to the DeeDS prototype. Chapter 3 introduces the problems and highlights the aims of this thesis. Chapter 4 describes the state of the art of current conflict detection approaches and gives reasons why the dynamic conflict detection approach has been chosen for DeeDS. The new design of the conflict detection and resolution and implementation details of the dynamic version vector approach in DeeDS are described in chapter 5. Finally chapter 6 highlights the results of this thesis and discusses them.

2.Background

This chapter gives an introduction to the field of conflict detection in optimistically replicated distributed real-time systems, and fundamental concepts that are used in this thesis.

2.1. Distributed real-time systems

As suggested by their name, distributed real-time systems are distributed systems which are subject to real-time constrains.
A distributed system is defined by Burns & Wellings (2001) as follows:
"…a system of multiple autonomous processing elements, cooperating in a common purpose or to achieve a common goal."
Reasons for building such systems could be better scalability, fault tolerance or a distributed application area where the processing elements are situated at different locations. In order to work together, the processing elements in the distributed system have to communicate with each other over networks, which often cause unknown delays and can slow down the whole distributed system. There is also a need to share data among the different processing elements. This can be done by different replication strategies, which are explained in chapter 2.3 in more detail. Additionally, the data on the processing elements should be consistent, according to the application requirements, throughout the whole system. Different levels of consistency have been suggested, some of which are described in chapter 2.4. If processing elements may work concurrently on replicas of the same data, situations can arise in which two nodes modify their local copy of the same object concurrent.
A real-time system is defined by Burns & Wellings (2001) as follows:
"…any information processing system which has to respond to externally generated input stimuli within a finite and specified period."
That means that it is not enough if the system just outputs the right result. It also has to deliver that result at a specific time or within a specific time period. Real-Time Systems are needed to monitor and control real world entities. The world does not wait and so the systems have to react on time. Application areas are, for example aircraft-control-systems or patient monitoring systems. For a hard real-time system, a correct result that is returned too late can endanger human life or lead to equipment damage with other serious consequences. So, real-time systems have to be designed to ensure timeliness of all outputs. To this end predictable behaviour and sufficient efficiency of all system components is the most important requirement. Meeting this requirement is made harder by many properties of a distributed system like unpredictable network delays, which causes many problems in distributed real-time systems.

2.2. Database systems

Database systems provide structured storage of data. Especially if huge amounts of data have to be stored and accessed regularly it is important that this can be done in an efficient way. Additionally many people should be able to work at the same time with the data stored in the database. To allow this database systems use the concept of transactions to ensure that operations of different users do not affect each other.

2.2.1.Transactions

Transactions can be seen as sets of operations that are working on the data stored in the database. If a transaction is executed values of the database are read or written. The idea of transactions is that operations that belong together and have to be executed without any influences from outside are encapsulated. To work correctly (see chapter 2.4) every database system should ensure the ACID properties (Elmasri, 1994). ACID stands for Atomicity, Consistency, Isolation and Durability. Below, these properties will be described in more detail.

Atomicity:	A transaction is seen as the smallest unit, meaning that either all operations of a transaction are executed successfully or not a single one is executed.
Consistency:	A transaction that starts from a consistent state does not lead to an inconsistent state. Either the transaction commits successfully and the new database state is consistent again or it is aborted and all the changes that are already made are undone to get back the original consistent state.
Isolation:	If many transactions are executed concurrently no transaction should be affected by another one. So, every transaction should be executed as if it was running alone.
Durability:	Once a transaction commits successfully its changes have to remain in the database indefinitely.

2.2.2. DeeDS

DeeDS (Andler et al., 1998) is a prototype of a distributed, active real-time database system. It aims to provide a data storage for real-time applications that may have hard or firm real-time requirements. As database DeeDS uses OBST (Object Management system of STONE) (Casais, 1992) and TDBM (DBM with transactions), which replaces the OBST storage manager. One main reason for introducing TDBM is to add support of nested transaction to DeeDS. TDBM is a transaction processing data store with a layered architecture (Brachmann & Neufeld, 1992), and provides DeeDS with the following (Eriksson, 1998):

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

To meet real-time constraints, all operations supported by DeeDS have to be predictable. This is ensured by avoiding delays for disk access, network communication and distributed commit through main memory residency, full replication and local commit of transactions. Local commit means that transactions are allowed to commit on a node by updating only the local database of that node. The other nodes are informed eventually. This behaviour avoids the unpredictable execution time of distributed commit protocols like the “Two Phase Commit Protocol” but also weakens global consistency. Instead of immediate global consistency DeeDS supports eventual global consistency (chapter 2.4). Local commit also introduces some concurrency problems like concurrent updates of different replicas belonging to the same object. To handle these problems, DeeDS uses conflict detection and forward conflict resolution (see chapter 2.5.4) which resolves conflicts without rolling back transactions (Gustavsson & Andler, 2005). Conflict resolution is done deterministically on all nodes so that global consistency is reached eventually if there are no new updates to the database. Local consistency at each node is ensured at all time by pessimistic concurrency control offered by OBST/TDBM.
To achieve better portability an extra layer called DOI (Deeds Operating systems Interface) is used between DeeDS and the operation system. This makes it possible to run Deeds on POSIX compliant systems like UNIX or LINUX but also on OSE Delta which is real-time

operating systems. In summary, DeeDS is a real-time database that ensures predictability at the cost of immediate global consistency.

2.3. Replication

In distributed systems replication can be used to achieve adequate performance and availability in data sharing (Ratner et al., 2001).
There are two fundamental strategies for data replication, namely pessimistic and optimistic replication. Furthermore, there are two different kinds of update propagation that can be used.

2.3.1.Pessimistic Replication

Pessimistic replication strategies aim to maintain consistency of all replicas at any point in time. All actions that could lead to an inconsistent state are avoided. This is often done at the cost of availability because replicas in the system have to agree to ensure consistency which leads to delays. Primary Copy (Stonebraker, 1979) and Voting (Gifford, 1979) are two strategies that ensure pessimistic replication (Barreto, 2003). The Primary Copy strategy assigns a master replica on which all operations have to be performed in the first place. The node containing the special replica then sends messages to all the other replicas to update these. If it fails, the remaining nodes elect a new master replica before continuing. The Voting strategy aims to reach consensus among all nodes on the operation which is allowed to be executed. To implement pessimistic replication, any strategy can be used that determines the action to be performed by all replicas.

2.3.2.Optimistic Replication

Optimistic replication strategies increase availability at the cost of consistency (Barreto, 2003). An operation does not have to wait for all replicas to agree. Instead it is allowed to be executed on a subset of replicas and is then replicated to the remaining nodes eventually. Thus, inconsistencies might occur if different replicas are updated concurrently. So the replica managers must contain a conflict detection and resolution to handle such inconsistencies in a way that leads to a consistent state. The different approaches for optimistic replication differ mostly in the way they detect and resolve inconsistent states. Chapter 2.5 gives a more detailed view of the fundamental versioning approaches that are used to detect inconsistent states.

2.3.3.Propagation types

State- and update propagation are the two possible types that can be used to inform other replicas in the system about changes. If state propagation is used the update messages include the new content of the replica. That means that a transaction is actually executed only once on one node and the result is send to update the other replicas. Update propagation in contrast sends the information about the committed transaction in the update message. This information is used to commit the transaction on every node that has to be updated. The advantage is that the information about the transactions can be kept somewhere and can be used if transactions have to be rolled back and re-executed. The disadvantage is the computational overhead since every node has to commit the same transaction. In real-time systems the worst case execution time has to be assumed for all calculations and so state transfer with only small computations shows a better behaviour for these systems.

2.4. Data Consistency

This chapter contains information about the main consistency notions that are used within this thesis. *Local consistency* means that the database of one node is in a consistent state. That means that a local transaction commits correctly or is completely removed. If a transaction is aborted all changes that have been done must be rolled back. *Global consistency* considers all replicas in the distributed system. If all replicas of each database object contain the same data at all points in time then the distributed system is globally consistent.
To be able to describe consistency guarantees models were introduced. The following section will describe the classes of consistency models.

Immediate Consistency (strong consistency guarantees)
Immediate consistency can be achieved by using replication protocols with strict serializability. This means that the changes of one transaction are made on all replicas before another transaction can read the updated data. In a distributed database system global consistency is a very interesting subject. A distributed database system is immediately globally consistent if all nodes in the system have the same view on the data. This requirement can only be ensured by pessimistic replication approaches that are explained in chapter 2.3.1.

Eventual Consistency (relaxed consistency guarantees)
Sometimes immediate global consistency is not necessary or is too expensive to achieve. A model for *eventual consistency* ensures that the databases of all nodes in the system become consistent at some point in time if no new updates are made to the database. It is possible that databases on different nodes are inconsistent at some points in time but this is a state that can be handled by applications which are aware of working on possibly inconsistent data. To achieve eventual consistency optimistic replication protocols can be used that show better availability since they do not have to wait until other nodes agree on a transaction commit.

2.5. Conflict detection & resolution concepts

If Optimistic Replication (chapter 2.3.2) is used it is possible that a node changes the content of one of its replicas without having all former updates integrated that have been made to the replica’s object by other nodes (not the newest version). This leads to a state in which different nodes have different values for the same object. If transactions are used an additional conflict type can occur that results from reading data that is not up-to-date. The mentioned conflicts lead to a global inconsistency of the distributed database. To reach global consistency which is absolutely necessary for every database the conflicts have to be detected and resolved. This chapter will describe the possible conflict types and then focus on the basic concepts that are necessary to understand the different conflict detection approaches. For every concept current projects are mentioned which show that these concept work also in practice. At the end two different conflict resolution concepts are described.

2.5.1.Conflict Types

There are two types of conflicts that are possible in distributed, optimistic replicated systems with transaction support namely write/write conflicts and read/write cycles. These two conflict types are described below in more detail.

Write/write conflict

Write/write conflicts occur if the content of a replica is modified which does not include all former changes that have been performed on other replicas belonging to the same object. As a result of this two replicas of the same object might contain different values. They are called conflicting replicas.

Read/write transaction cycle

In systems that support transactions read/write transaction cycles might occur. A *read/write transaction cycle* is a set of transactions with the characteristic that no transaction contains the newest versions for all the objects it contains in the read- and write-set. For a better understanding an example with a small transaction cycle that contains 2 transactions is given. A transaction T1 read an object that a second transaction T2 concurrently writes, and transaction T2 reads an object concurrently written by transaction T1. In other words this conflict occurs if an object is modified based on stale read data. If this conflict is not handled the consistency of the database can be violated. One example can be shown on a database with a constraint that two objects O1 and O2 are not allowed to be 0 at the same time. Considering two transactions T1 and T2 where T1 reads O1 to check if it is 0 and if this is not the case sets O2 to 0 and T2 checks O2 for 0 and if this is not the case sets O1 to 0.
If now T1 and T2 both check for 0 before one of them sets an object to 0 it seems to be no problem for both to set O1 and O2 respectively to 0. However this causes a database constraint violation because O1 and O2 are not allowed to be 0 at the same time. Figure 2.1 illustrates the described example. The numbers on the arrows represent the execution sequence. Read/write cycles can contain more than two transactions and that makes them sometimes hard to detect.

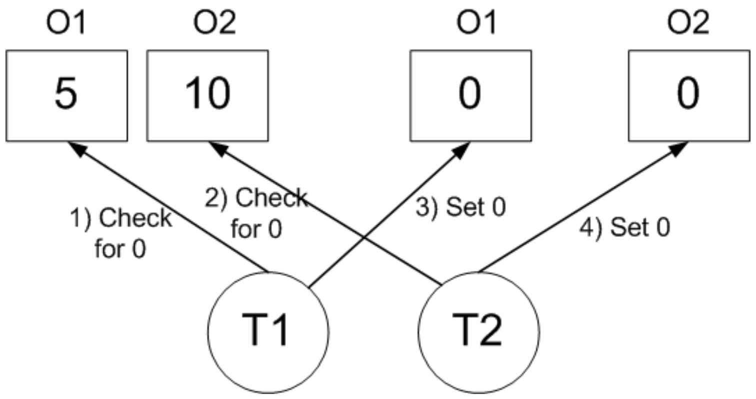


Figure 2.1: Constraint violation

2.5.2.Basic versioning concepts

There are two different fundamental versioning concepts that can be used to classify the available conflict detection approaches. The first versioning concept is version vectors.

A dynamic version vector (DVV) for a replica R is a set of pairs (N_R, C_R) where N_R is a unique node ID and C_R a counter belonging to N_R. The value of C_R indicates how many times node N_R has written to the version of R represented by the version vector.

In other words version vectors count the number of updates a replica suffers from each node in the system. This information is used as a version number that can be used to order actions on the replicas.
For example if one replica has been updated 3 times by a node N then this version of the

replica is newer than a version that shows only 2 updates from node N. Of course, normally there is not only one node writing to a replica. The conflict detection approaches that are using this concept are the basic version vector- (Parker et. al., 1982), Bayou version vector- (Almeida et al., 2002) and dynamic version vector approach (Ratner et al., 1997). The basic version vector approach has been used in the distributed file system Locus (Parker & Ramos, 1982), Bayou version vectors in the distributed database Bayou (Petersen et al., 1996) and the dynamic version vector approach is implemented in the distributed file system ROAM system (Reiher, 1999).
The second concept is the history based concept. History based approaches use the fact, that there is no need to know the exact number of changes a specific node did to a replica. Instead unique IDs are given to replica states. Every time the content of a replica changes it gets a new unique ID. By saving all the IDs of a replica (the replica history) a kind of versions identifier can be created that can be used to order actions. If one version (ID history) includes all the unique IDs of another version then this version is newer. Causal histories and hash histories are approaches that are based on this concept. The hash history approach was implemented and tested in a CVS system but till now there is no project that uses it.

2.5.3.The Log Filter concept

This concept can be used if additionally to write/write conflicts read/write transaction cycles have to be detected. The idea is to store information about every transaction that has been integrated into the database in a list. In more detail the stored information for each transaction consists of the version numbers of every object in the read and write set of the transaction. By knowing the version numbers of the objects a transaction saw when it was performed the transactions can be sorted. If the version numbers of all objects of a transaction T1 are newer than these of another one T2 than T1 is considered to be happened after T2. A read/write cycle is detected if the Log Filter list cannot be ordered. Figure 2.2 gives a simple example.
A Log Filter history can contain many transactions that are represented by T0 to T3. Each transaction reads and writes objects. At the time where these objects are accessed they have a specific version number. This version number is saved in the Log Filter list for every object. In the example O1, O2 and O3 represent the version numbers of read or written objects of the transaction. Now it can be seen that in the illustrated case T1, T2 and T3 cannot be ordered and a transaction cycle would be reported.

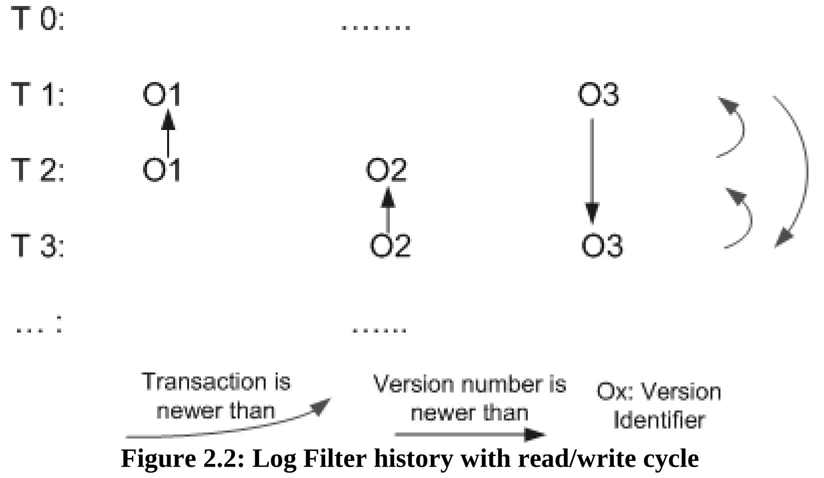


Figure 2.2: Log Filter history with read/write cycle

2.5.4.Conflict resolution types

There are two different conflict resolution types, namely backward and forward conflict resolution. Backward conflict resolution is the technique that comes from the database area. If transactions are conflicting they are rolled back and then executed again. If there are dependencies between transactions it might be necessary to roll back more than one transaction. In real-time systems rolling back should be avoided since it introduces unpredictable waiting times for rolling back the transactions. Instead of rollback forward conflict resolution can be used. In case of a conflict forward resolution tries to achieve a consistent state without undoing already committed transactions. One forward resolution technique is merging the two conflicting replicas in a deterministic way by, for example, calculating the average value. Forward conflict resolution is often used with state propagation (see chapter 2.3.3) because the information for rolling back transactions is often not available anymore.

3.Problem Description

In distributed systems with optimistic replication (see chapter 2.3.2), like DeeDS, conflicts have to be detected. Version vectors is one mechanism that can be used for that. But this mechanism is not able to handle large and dynamic numbers of nodes well. This is because the data structures used in these mechanisms grow with the number of nodes and also have to be changed during runtime when a node joins or leaves the network. There is also no implicit bound on the version vector sizes. For real-time systems these characteristics are even more problematic because they lead to unpredictable behaviour in conflict detection.

This chapter describes the above mentioned problem in more detail. It gives a motivation for why it is necessary to face this issue and shows the different steps to a solution.

3.1. Motivation

Distributed systems like the Bayou database (Petersen et al., 1996) or the ROAM file system (Reiher et al., 1999) use a weak consistency model to achieve better throughput and availability. For distributed real-time databases like DeeDS, predictability is another very important reason for using such a model.
In a strict consistency model a transaction is only allowed to commit if all replicas of the changed objects have applied the new values. So there is a need to wait until all these replicas have sent an update confirmation message to ensure that all have made the changes. In a distributed system, this waiting time is unpredictable because it depends on many different things like the network traffic or the availability of the replicas.
Weak consistency implies that transactions do not need to change all replicas to commit. This avoids or at least reduces the unpredictable waiting time of the strict consistency model because the update of replicas on all other nodes is not included in the commit process of a transaction. For applications that are aware of global inconsistencies and know how to handle it, it is sufficient that the local replicas are updated and so the unpredictable waiting time for confirmation messages of other nodes can be completely avoided. But there are also some things which have to be considered if such optimistically committed transactions are used (transactions which commit only locally). There is a possibility of conflicts because of concurrency (see chapter 2.1). So, a mechanism is needed to detect and resolve such conflicts during the update process.
Conflict detection mechanisms based on basic version vectors can be used, but implementations of these typically do not scale well and have difficulties when handling a dynamic number of participating nodes (Ratner et al., 1997). This means that there are problems that have to be solved if these mechanisms are to be used in a dynamic environment.
In nearly every part of our world embedded systems are used and in many cases it is necessary to connect them so they can interact with each other. For example, the emerging technologies for ubiquitous computing require wireless ad-hoc networks consisting of many very flexible devices. In such scenarios nodes join or leave the network very often because of the restricted radio range, energy problems or other reasons like system failures. Even in such scenarios there is a need for real time processing in some cases. Especially in flood and fire warning or military systems where sensor signals have to be evaluated very fast, distributed real-time databases like DeeDS are needed. It is therefore necessary to develop a conflict detection which is able to handle a dynamic set of nodes for such a system.

3.2. Aims

a) Comparison of existing conflict detection approaches:

There already exist different approaches to conflict detection in distributed systems. So the first aim is to have a taxonomy of these approaches and a survey considering the advantages and disadvantages. There should also be an analysis if it is possible to create a hybrid solution.

Evaluation:
This aim is achieved if a comparison of the main conflict detection mechanisms exists. The comparison has to consider the questions mentioned in 3.3.a) and there should be a conclusion which expresses what parts of the mechanisms could be integrated in DeeDS and if there is the possibility of hybrid solutions.

b) Implementation in DeeDS:

The next aim is the implementation of an existing or a newly created approach for conflict detection which fits into the DeeDS environment and shows good behaviour towards scalability and the handling of dynamic node sets. If necessary, there should be an adaptation of existing data structures and algorithms to DeeDS to achieve this aim. The resulting implementation should be evaluated towards the criteria mentioned in 1.1b).

Evaluation:
This aim is reached when the conflict detection approach is implemented and tested for correctness. Additionally, there should be a discussion about how the implemented approach influences the conflict detection process.

c) Hooks to conflict resolution:

Because the conflict resolution depends on the application semantic the conflict detection should be able to call different conflict resolution mechanisms to resolve different conflicts. I.e., there should be a separation of conflict detection and conflict resolution so that both are independent and there should be hooks which make it possible for the conflict detection to call different conflict resolution mechanisms.

Evaluation:
The conflict detection and the conflict resolution have to be divided into two independent modules. Furthermore there have to be hooks which make it possible to use different conflict resolution mechanisms when a conflict is detected. At the end the two modules (conflict detection and conflict resolution) and the hooks have to be tested for correctness.

3.3. Objectives

In this chapter, each aim is divided into several objectives so that it is easier to understand what has to be done to reach the specific aim.

a) Comparison of existing conflict detection approaches:

First there should be an analysis of the most important approaches to conflict detection which are currently available. After this analysis a survey and a categorization of the existing approaches should be created and based on that work there should be a comparison of the approaches that are relevant for DeeDS. The comparison should be based on the following criteria:

1) **Handling of a dynamic node set.**
2) **Real-Time characteristics**
3) **Adaptability of the mechanism to DeeDS**

b) Implementation in DeeDS:

A plan for the integration of the chosen conflict detection approach in DeeDS is needed. In this plan it is necessary to determine the data structures and algorithms which should be used as a basis for the implementation. Following this plan should be an implementation of the chosen approach. The goal is to have a proof of concept that the theoretical constructs work in practise and that the new data structures and algorithms lead to improved behaviour. Lastly it should be discussed how the new approach improves the conflict detection in DeeDS. The discussion should include the following issues:

1) **size of conflict detection data structures**
2) **efficiency of the conflict detection algorithms**
3) **limitations of the conflict detection approach**

c) Hooks to conflict resolution:

First, a concept is needed which considers how to separate conflict detection from conflict resolution in a way that both of them are completely independent to allow the use of more than one conflict resolvers. The concept should also focus on the need to call different conflict resolution mechanisms from the conflict detection via hooks. Based on this concept there should be a working implementation based on the developed concept for DeeDS.

3.4. Delineations

The main part of this thesis is about conflict detection. Creating new conflict resolution mechanisms or policies is not in the scope of this thesis.

4.Conflict Detection

In this chapter different conflict detection approaches based on the concepts described in chapter 2.5 are compared. Reasons why the dynamic version vector approach (Ratner et al., 1997) is seen as best fitting for DeeDS (Andler et al., 1997) are given. Furthermore, the characteristics that have been regarded as crucial during the comparison process are highlighted.

4.1. Existing Approaches

This part of the chapter will give an overview of the existing classes of conflict detection approaches. For this the existing approaches will be analysed and then compared with each other.

4.1.1. Overview

As described in chapter 2.5.2 it is possible to divide the conflict detection approaches in two classes. The version vector based like Bayou version vectors (Douglas et al., 1995) and dynamic version vectors (Ratner et al. 1997), and the history based approaches like hash histories (Kang et al., 2003). The version vector based approaches go back to a first approach of Parker & Ramos (1982) who based their work on that of Lamport (1978), whereas the history based approaches are derived from causal histories (Schwarz et al., 1994). But there are also mixed approaches that show characteristics of both version vectors and version histories. Version stamps (Almeida et al., 2002) is an example of such an approach.
Figure 4.1 gives an overview of the different approaches and the concepts behind them.

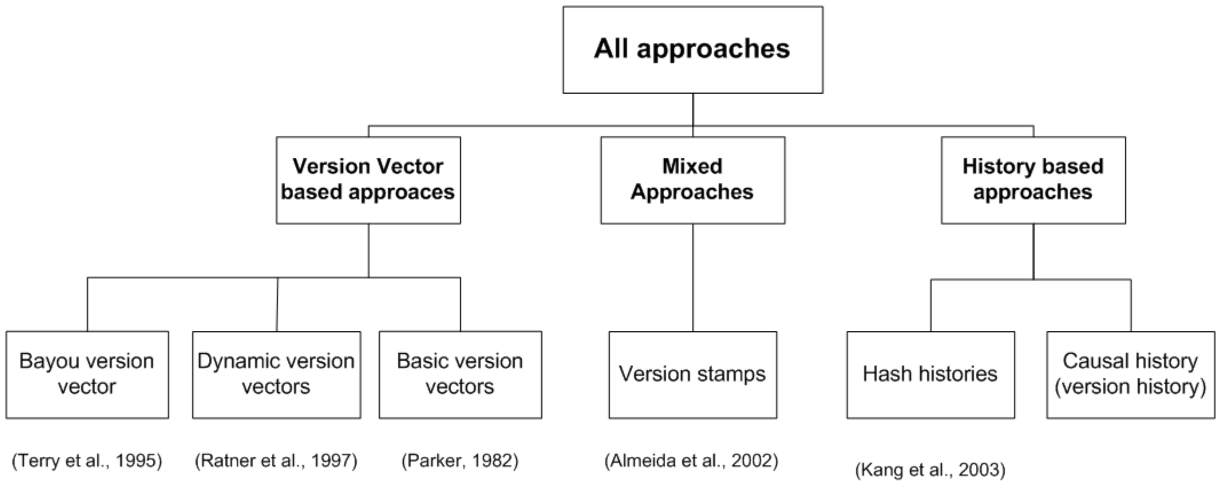


Fig. 4.1: Taxonomy of conflict detection mechanisms

4.1.2.Version vector based approaches

Basic version vectors

Basic version vectors were introduced by a paper of Parker & Ramos in the year 1982. It is the first version vector based approach and is used by the other version vector approaches which are modifications or extensions to it. The basic version vector approach only supports a fixed amount of participating nodes because the size of the version vector is predefined and every node has a fixed position which makes a direct access of entries possible. A version vector is a vector of counters, one for each node. Every replica in the system has its own version vector to describe its history of local changes and merged updates. If a node writes to

a replica the version vector of this replica is changed by increasing the counter associated with this node by 1. The version vector is also changed when two replicas merge. The aim is to know the exact number of writes to a specific replica of every node in the system. Two version vectors are compared by considering every pair of counters belonging to the same node. If the value of each counter in a version vector V1 is greater or equal than the corresponding value in a version vector V2 then V1 dominates V2. This means that V1 includes all changes of V2. But it is also possible that both version vectors have counters with higher values than the corresponding counters in the other version vector. As a result, none of these version vectors dominates the other. In this case there is a conflict because the replicas were changed by different nodes and neither includes all the updates of the other one. So there is a need to resolve the conflict. If optimistic replication (see chapter 2.3.1) is used the conflicting replicas could be, for example, merged. During this process the version vectors also have to be merged by giving every counter in the new version vector the maximum value of the corresponding old ones. At the end the new version vector has to dominate the two old version vectors.
To better understand version vectors the following example is given (Fig.: 4.2). There are three nodes which all have a copy of the same object. On node 1 the copy is called replica 1, on node 2 the copy is called replica 2 and so on.
The replica on node 1 has the version vector <1,0,0>. This means that node 1 has changed the content of this replica one time, but has seen no updates from nodes 2 and 3. At the beginning node 1 propagates its version to the other nodes in the system so that all nodes have the same versions. In the next step, node 1 and node 2 write concurrently to their local replicas. As a result of this the version vector of replica 1 changes to <2,0,0> and the vector of replica 2 to <1,1,0>. The replica on node 3 still has the version vector <1,0,0> because node 3 made no changes to it. Because node 1 and 2 wrote to their local replicas they now would start the propagation process. To keep this example simple we consider only the propagation messages of node 2. These messages are sent to node 1 and node 3. On these nodes the version vector of the local replicas is compared with the version vector of the propagation message to find out which version is newer. On node 3 the result of this comparison is that the version of the update message dominates the local replica and so includes all the local updates. That means that the local replica can simply be overwritten by the value included in the propagation message assumed that state propagation is used (see chapter 2.3.3). On node 1 the result of the comparison is different because node 1 also wrote to its local replica after the last update. This now leads to a conflict because neither version vector dominates the other one. Node 1’s local replica has the version vector <2,0,0> and the update message <1,1,0>. So the 2 is greater than the 1 but the 0 in the second entry of the local version vector is not greater than the 1 in the corresponding entry of the update message. This conflict has to be resolved and the two version vectors have to be merged. By taking the maximum values of both version vectors the resulting version vector is <2,1,0>.

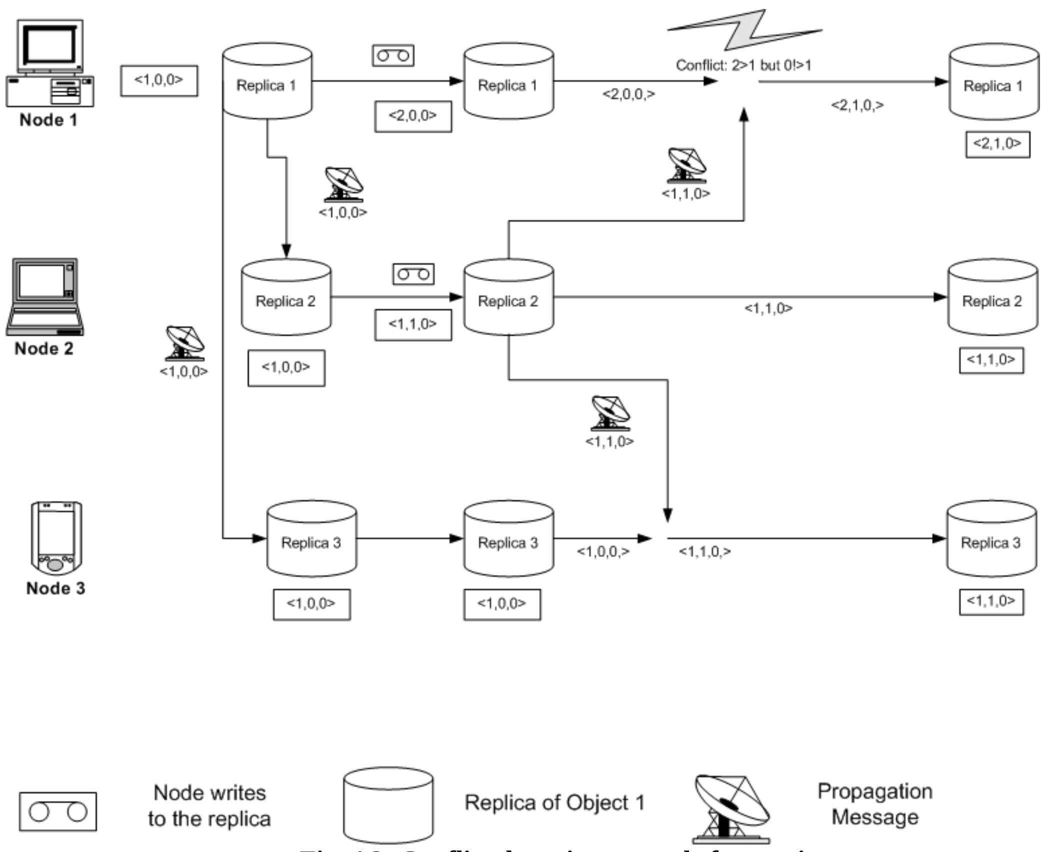


Fig. 4.2: Conflict detection example for version vectors

Bayou version vectors

The Bayou version vector approach expands the basic version vector approach by allowing nodes to join or leave the network. To this end dynamic version vectors data structures have to be introduced in which the nodes have no fixed position and have to be identified by an ID. This ID is created recursively so that every node has a unique ID. Joining and leaving of nodes is done with the help of the update operation that ensures that all nodes in the system eventually get information about the node leaving.
So, if one node leaves the network it sends an update message to at least one of the other nodes telling that it is not in the system anymore. If a node gets such an update message it deletes the associated entry in the version vector. If this decreased version vector V1 later is compared with another one V2 which still has the entry of the left node then this is recognized and the entry in the vector V2 is deleted. If a node is deleted its ID is never assigned again to a new node to avoid erroneous deletions. Thus, entries can be removed from the version vector without the need of reaching consensus of all nodes. Joining the network is done by sending a request to one of the participating nodes. A node that receives a join message creates an update message that adds a new entry to its version vector. The version ID of the new node is created by adding the version vector of the corresponding update message to the ID of the node that created the update. So even if many nodes are joining the network they all have unique ids.

Dynamic version vectors

The dynamic version vector approach supposes that only a small set of nodes write to a replica. For conflict detection only these writing nodes have to be in the version vector

because if a node just reads the replica there are no changes to the version vector. Since that it is not known in advance which nodes will be writers and which will only read the replica the version vector has to be very flexible. It is also possible that the set of writing nodes changes from time to time. So a dynamic version vector (DVV) has to offer the possibility to expand and compress itself depending on the number of writing nodes.
Like in Bayou version vectors there are no fixed positions for the nodes in a DVV. The entries are <ReplicaID:Counter> pairs where “ReplicaID” has to be a unique identifier for a node and “Counter” is the number of updates the node did to that replica. At first the DVVs of all replicas are empty. This changes when the first node N starts writing to its replica because then the DVV of that replica is expanded by adding a <ReplicaID:Counter> pair to it. In this case the “ReplicaID” is the unique ID node N got during the network join process. The Counter value is set to 1 the first time node N writes to this replica and is increased by one for every further update of node N. If only expansion would be allowed then the vector would become larger and larger and that would lead to a very bad performance. So there is also a need to compress the DVV. This is not as easy as expansion because it has to be ensured that all version vectors in the system are changed equally. Otherwise, the comparison of two version vectors would be useless. Before deleting the version vector entry of a node all other nodes should have received all the updates concerning that node.
The DVV approach uses a consensus algorithm (Ratner et al., 1997) to guarantee equal treatment of all version vectors. This algorithm decreases the counter value of one node in all version vectors by the same number. If one counter is zero after it has been decreased then the whole entry is removed and so the version vector compressed. Additionally, every version vector has a version number which is increased after every compression. So if two DVVs are compared and one of them has a newer version number than the other one, it is known that consensus is reached on all nodes and the old one can be compressed too. The version numbers of different version vectors can only differ by one. This is because consensus of all nodes has to be reached before compression can take place and a node rejects compression requests for a version vector entry if compression is already running on it. During the consensus process no operations are blocked so that no unpredictable waiting times are introduced in the conflict detection by this algorithm. More details about the DVV consensus algorithm are available in Ratner et al. (1997).

4.1.3.History based approaches

Causal Histories

The causal history of an event (R. Schwarz & F. Mattern, 1994) is a set of events that happened before this event. In the causal history approach a version history for every replica is maintained. Every time the value of a replica is updated, a new version number that has to be unique in the whole system is created and stored in the replica's history. The histories of the replicas are totally ordered to see the causal relationship between the different versions. The ordering is also very useful when throwing away old values, which are already integrated in all nodes, to shorten the history. To compare two replicas both histories are needed. A history H1 dominates the history H2 if the history of H2 includes all history entries of the history H1. In other words if H1 consists of the history H2 plus additional version numbers then H1 includes all the changes of H2. In this approach there is also the possibility that none of the histories dominates the other one because H1 is not a part of H2 and vice versa. In this case there is a conflict because new version numbers have been added concurrently which means that the replicas were changed concurrently. To solve the conflict the ordering of the histories could be used to find the most recent common ancestor and then process all the newer updates of both conflicting replicas. If update transfer is used it may even be possible to directly find out the exact deltas which have to be executed to get a new version which includes all the updates of both original replicas.
Now, an example is used to illustrate the use of version histories (see also Fig.: 4.3). There are no changes in the behaviour of the nodes but instead of version vectors, version histories are used.
First of all node 1 propagates its local replica to all the other nodes. There is one entry in the history, the version ID of that replica. It is important that this versionID is unique in the system. Now again, node 1 and 2 changes the values of their local replicas concurrently. Node 1 adds the version ID 346 and node 2 adds the version ID 653.
Now, node 2 propagates its replica to the other nodes. During the integration process of the propagation message the other nodes compare the history of that message with the history of their local replica. Node 3 didn't write to its local replica so its whole history consists of only one entry, the 457. This entry is included in the history of the update message and so the whole history is included. This means that the version of the propagation message dominates the version of the local replica and so the local replica can be overwritten without losing any information.
The conflict in node 1 is detected because neither history includes all version ids of the other history. After the conflict resolution the new history has to include all the version ids of both original histories. The order of the entries after the last common ancestor depends on the conflict resolution process.

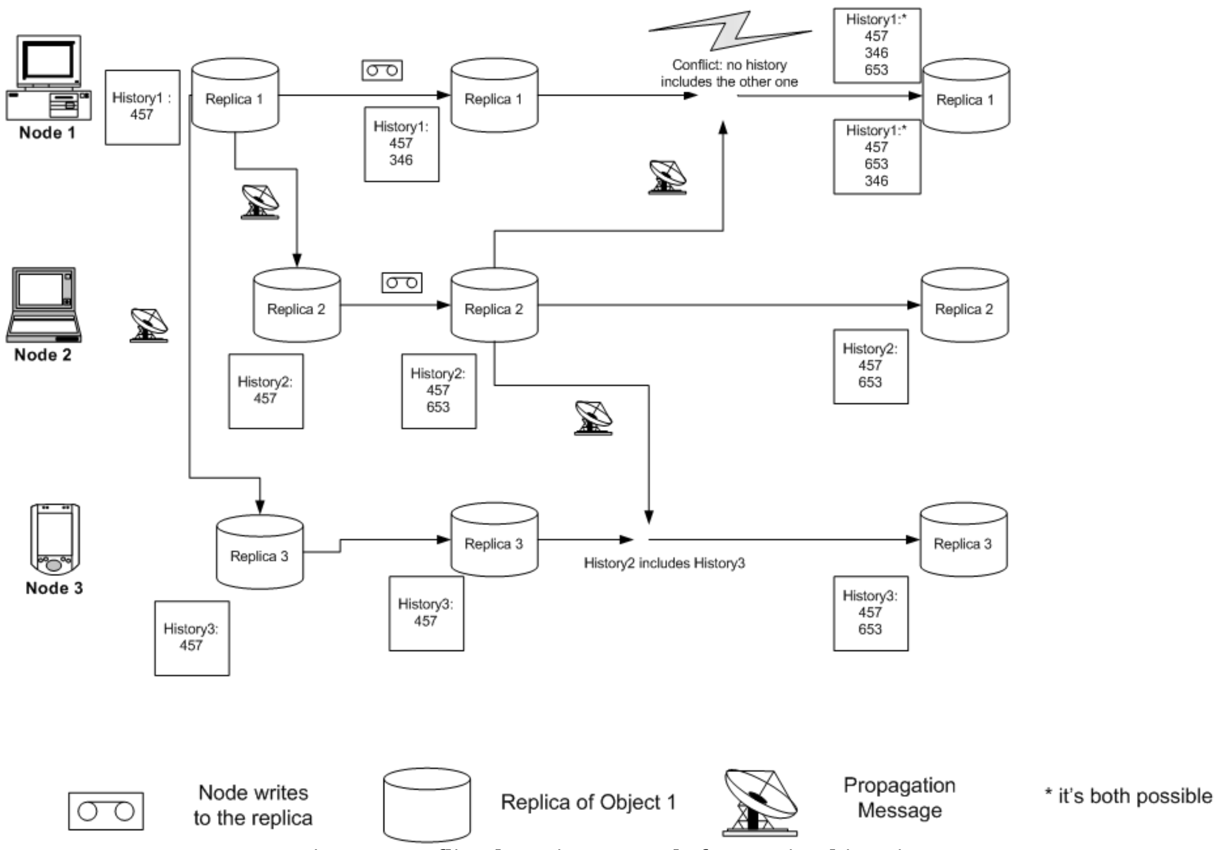


Fig.4.3: Conflict detection example for version histories

Hash Histories

The Hash history approach (Kang et al., 2003) is derived from the causal history approach. The main difference from causal histories is that the version numbers are created by calculating the hash values of the replica's contents. The idea behind this is that a conflict only exists if there are two different replica versions with different contents. In other words, if the contents of two replicas are the same there is no need to resolve the conflict. This is referred to as detection of coincidental equality because the replica versions are not causally related but their contents are identical. The hash function that calculates the hash values (version numbers) has to be chosen carefully because otherwise several replicas may get the same value (version number) even if their contents are not the same and that would have bad consequences for conflict detection. In the hash history approach common hashing algorithms like SHA-1 (National Institute of Standards and Technology Gaithersburg, 1995) are used which are optimized to calculate different hash values for different contents. Summarizing this approach it can be said that hash histories create a list of hash values that represent the different versions of the replicas' contents. By comparing two of these histories it can be established which updates are included in which replica and so conflicts can be detected.

4.1.4.Version Stamps

The version stamps approach is not completely new because it uses parts of version vectors and version histories. The new idea is that it is not necessary to keep all versions that appear in the system. It is enough to only keep the so called *frontier elements*. Frontier elements are version IDs which are currently used by at least one replica. So if one version ID is old and it

is overwritten on every node by a new version ID then it is not necessary to remember the old version ID anymore. But there is one thing that has to be ensured if old version IDs are discarded. A history needs to be included in the version ID. Otherwise it would be impossible to detect whether one version is older than any other or one version includes the other. To achieve this the approach uses a recursive naming scheme of all replicas. It ensures that every replica has a unique identifier even if it has the same content as other replicas. For this, a fork process is introduced which creates new version numbers every time a replica is copied. The new version numbers are created by appending a zero or a one at the end of the old version number. After forking a replica with the version ID 0 the two resulting versions are 00 and 01. By doing this it's possible to find out if one version is a predecessor of the current one by simply checking for prefixes. If a version ID is a prefix of another one then it is a predecessor of it. When two versions are merged their version numbers are also merged. Version ID 00 and the version ID 10 are merged to version ID 00+10, for example. In addition to the version ID every replica has to save an update ID. The update ID is the last version ID which suffered changes. A replica always maintains two components, the "update ID" and the "version ID". During the merge process of two replicas the update IDs are also merged like the version IDs. Because the forking and the merging processes lead to longer version IDs there is a need to decrease the size from time to time. This is done in the following way. If both replica identifiers are present in the same version stamps which were created through forking they can be compressed. For example, if a version stamp contains the sequence 00+01 it can be compressed to 0 because we know that they both were created from the replica with this number and are now merged again (otherwise these numbers were not in the same version stamp). In (Almeida et al., 2002) it is mentioned that the version stamps are seen as boolean expressions and compressed by using boolean algorithms.
To better understand the concept Fig. 4.4 illustrates the forking and merging processes. The deltas are write accesses to the replica which lead to a copy of the second component (version ID) into the first component (update ID).

$$\begin{tikzcd}
 & & & & & {[0 \mid 00]} \arrow[r, "\delta"] & {[00 \mid 00]} \arrow[ddr] & & \\
 & & & {[\mid 0]} \arrow[r, "\delta"] & {[0 \mid 0]} \arrow[ur] \arrow[dr] & & & & \\
{[\mid]} \arrow[r, "\delta"] & {[\mid]} \arrow[r, "\delta"] & {[\mid]} \arrow[ur] \arrow[dr] & & & {[0 \mid 01]} & & {[00+1 \mid 00+1]} \arrow[r, "\delta"] & {[00+1 \mid 00+1]} \\
 & & & {[\mid 1]} \arrow[r, "\delta"] & {[1 \mid 1]} \arrow[r, "\delta"] & {[1 \mid 1]} \arrow[urr] & & &
\end{tikzcd}$$

Figure 4.4 Version Stamps – forking and merging

Figure 4.5 shows the decreasing of the version and update ids. At the end the [011|01] becomes [01|01]. This is because the first component (update ID) has to be a prefix of the second one.

$$\begin{tikzcd}
 & & {[0 \mid 0100]} \arrow[r] & {[011 \mid 011+0100]} \arrow[r] & {[011 \mid 011+0100+0101]} \arrow[d, Rightarrow] \\
 & {[0 \mid 010]} \arrow[ur] \arrow[dr] & & & {[011 \mid 011+010]} \arrow[d, Rightarrow] \\
{[0 \mid 01]} \arrow[ur] \arrow[dr] & & {[0 \mid 0101]} \arrow[uurr] & & {[011 \mid 01]} \arrow[d, Rightarrow] \\
 & {[0 \mid 011]} \arrow[r, "\delta"] & {[011 \mid 011]} \arrow[r, "\delta"] & {[011 \mid 011]} \arrow[uuu] & {[01 \mid 01]}
\end{tikzcd}$$

Figure 4.5 Version Stamps – decreasing the size

Version Stamps are used in the PANASYNC project (Almeida et al., 2000) to recognize write/write conflicts between updates to files.

The example that is used in the other approaches can be also used here but there is a problem of version stamp decreasing if the replicas are not really merged but instead stay on different nodes and only exchange update messages. The example for this approach is shown in figure 4.6.
In this approach the fork process is used to copy the replicas from node 1 to the nodes 2 and 3. Then node 1 and 2 modify the replica which results in a copying of the version ID of the replica to the update ID (so that update ID and version ID are the same). Considering now the point at which the update message of replica 2 arrives on node 1. One update ID is 00 the other is 01 which means that none of both is a prefix of the other one. The result is a conflict like in the former cases. The problem arises after the merging of the version stamps to "<00+01/00+01>". Like shown in figure 4.5 this version stamp could be decreased to <0/0>. This would be no problem if replica 2 would have been removed from node 2 but unfortunately this is not the case because only update messages are sent. So the result is that replica 1 and replica 2 have the same content but different version stamps. One possibility to overcome this problem would be to remove replica 2 and merge it with replica 1 and fork the new version to create a new updated replica 2 but this whole procedure then has to be atomic. Additionally the decreasing mechanism would not work well anymore because the fork operation would increase again the size of the version stamp. It seems that this approach has been developed for a system in which one server has a main replica and other replicas are only created on clients during network partitions. If the partition is over these replicas are all merged again to the main replica on the server. Unfortunately this is not the case in a DeeDS system where there is no main server.

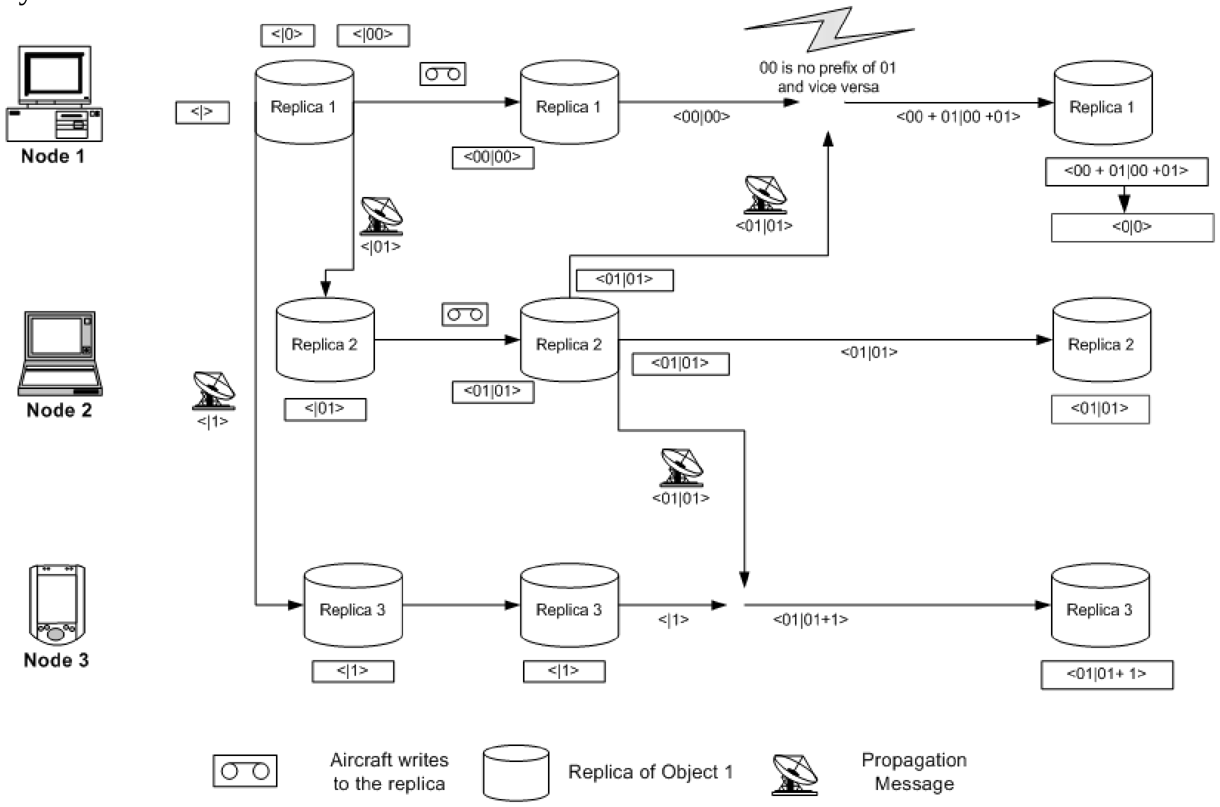


Figure 4.6 Conflict detection example for version stamps

4.1.5.Comparison

In this chapter the different approaches are compared with each other. The criteria are derived from the main goal of this thesis that is to find a conflict detection approach for an optimistically replicated real-time database (DeeDS) that can handle an environment with a dynamic node set.
As stated in chapter 3.3a) the criteria are:

1) **Handling of a dynamic node set**
2) **Real-Time characteristics**
3) **Adaptability of the mechanism to DeeDS**

The first criterion ensures that the chosen approach can handle a dynamic node set and so can be used in the requested environment. The second criterion ensures that the approach can be used under real-time conditions which means that it is predictable and sufficient efficient. The last criterion considers the fact that the conflict detection approach has to be implementable in DeeDS and so should not contain any characteristics that make this impossible. At the end a section "Additional important characteristics" has been added which includes important issues that fit in none of the above mentioned criteria.

Handling of a dynamic node set

In a dynamic environment nodes join and leave the network frequently. It is thus important that a conflict detection approach used in such an environment can add and remove nodes efficiently. In the basic version vector approach adding or deleting a node is not supported. This means that this approach cannot be used in an environment with a dynamic node set and so it can be disregarded. But there are other version vector based approaches that can cope with a dynamic node set namely Bayou version vectors and DVVs (dynamic version vectors). In the Bayou version vector approach there is a need to explicitly send a join or delete message to remove a node from the version vector whereas the DVV approach can implicitly remove entries from time to time via the DVV consensus algorithm. In the hash history and version stamp approaches, the nodes in the system have no direct influence on conflict detection because no node information is included in the versioning structures (hash history and version stamp respectively) that are assigned to replicas. The hash history approach uses hash values of the replicas' contents whereas the version stamps approach uses unique IDs of the replicas. Joining and leaving of nodes in the system thus does not influence the hash history and version stamps approaches in a way that data structures have to be changed.
All conflict detection approaches use IDs to identify replica versions or nodes. Especially if nodes frequently join or leave the system assigning such IDs is not a trivial task for some of the examined approaches.
Bayou version vectors need unique IDs for every node in the system. To ensure this a naming scheme is introduced that gives every node in the system the possibility of creating a new unique ID by taking its ID and appending a postfix. In the Bayou system IDs can never be reused because of the compression algorithm that just removes nodes from the version vector. The consequence of this is that they grow without bound over time especially if nodes are removed and added very often.
The DVV approach also needs unique identifiers for nodes but old IDs can be reused. The IDs can be created with any algorithm that ensures that these are unique. Hash Histories do not need unique IDs for nodes. This is an advantage when supporting a dynamic node set because the overhead of maintaining unique node IDs is avoided. The idea is to relate IDs to replica versions. Every time a replica is changed a new ID is created by calculating the hash value of

the replicas' contents and this is independent from the nodes in the system. If two replicas on different nodes have the same contents they also have the same ID (the hash value). Version stamps use unique identifiers on a replica basis and not on a node basis like the version vector based approaches. If a replica has to be copied to another node the identifier of this replica is split up into two new identifiers adding a 0 or a 1 to the old identifier. Thus identifiers grow with every copy process.

Summary: The basic version vector approach is not suitable for DeeDS because it cannot handle a dynamic node set and so is not able to solve the problem this thesis is faced with. The version vector based approaches can handle a dynamic node set but need a naming scheme that assigns unique IDs to every node which is not a trivial task in a dynamic node set. Thus the hash history and version stamps approaches fulfil this criterion best.

Real-Time characteristics

The most important characteristic of a conflict detection approach with regards to its usability in a real-time system like DeeDS is predictability. To achieve predictability the worst case execution times of the approaches are important because only by assuming this time it can be assured that the task is really finished. In systems where hard and soft real-time tasks are running together the average execution time could be also interesting because more execution time would be available for the soft real-time tasks (more soft real-time tasks could be run). This is the reason why the average case is also shortly discussed in this section.
The worst case execution time of the complete conflict detection process for one transaction is influenced by many different parameters, for example the read/write set size of transactions or the number of transactions already included in the conflict detection history (such as a Log Filter). For the evaluation of the different conflict detection approaches in this chapter only the worst case execution time of the comparison of two conflict detection data structures is examined because only this time is directly influenced by the different approaches and is the indication if one approach is better than another. The conflict data structures of the different approaches are partly very different. So, a direct comparison is only possible between the version vector based approaches. The comparison between the other approaches is more difficult and discussed at the end of this section. Regarding the VV based approaches, the number of elements in the VV determines the execution time of the comparison. This is because all the elements of two VV that are compared have to be compared. So the number of elements is an important criterion in assessing the approach.
Assuming that the total number of nodes that can join or leave the system is limited the number of VV entries in the Bayou - and DVV approaches are bounded. The worst case occurs if the VVs cannot be compressed due to network partitions or other disruptions.
In the worst case a Bayou version vector includes entries for all nodes that have ever been in the system. This happens in case of network partitions in which some nodes cannot be informed about the leaving of others. But since consensus of all nodes is not needed to remove an entry of a version vector only the version vectors of nodes that are unreachable in a network partition stay uncompressed until the partition is over. All other version vectors can be compressed. The reason why consensus is not needed to remove one entry is that identifiers of nodes are never reused.
A DVV of an object contains entries for all the nodes that ever wrote to replicas belonging to that object. This means that in comparison to the Bayou VV approach not all the VVs in the system include the same set of elements since not all nodes might write to all objects. To remove entries of nodes that stopped writing to the replica the DVV consensus algorithm has

been added, which leads to a smaller average version vector size. If executed regularly the algorithm is a very good optimization (Ratner et al., 1997).
Comparing the worst case conflict detection data structure sizes of the two approaches, the DVV approach shows better behaviour because only in a scenario where all nodes write to all objects the worst case of both approaches is the same. In all other cases the DVV approach contains smaller VVs than the Bayou VV.
The version stamp approach suffers from the problem mentioned in chapter 4.1.4 and that's why this approach cannot be used until a solution is found that leads to a predictable version stamp decreasing. Without this predictable version stamp decreasing algorithm it is not possible to discuss the worst case of this approach.
The conflict detection data structure of the hash history approach is a history containing all the hash values of former replica versions. A history shortening algorithm is necessary to cut the history from time to time to set an upper bound. But if there are no time bounds set for the replication it is hard to cut the history. In the worst case, when no time bounds exist, all entries would have to remain in the history and this would result in a very large conflict detection data structure. Many comparisons are then necessary to compare all the hash values of one history with another one.

Summary: Version Stamps cannot be used if the replicas remain on different nodes because then the compression of the version stamps does not work in an efficient way anymore (see section 4.1.4). The dynamic version vector approach shows better behaviour than the Bayou version vector approach. The comparison of the dynamic version vector approach with the hash history approach is difficult because the conflict detection data structures are very different. But assumed that over time there will be more object updates than writing nodes in the system the dynamic version vector approach is the better one.

Adaptability of the mechanism to DeeDS

For systems like DeeDS that support transactions write/write conflict detection alone is not sufficient. Additionally read/write cycles have to be detected.
One mechanism to detect read/write cycles are Log Filters (Parker & Ramos, 1982), which were developed to work with the basic version vector approach. However, Log Filters seem to be not completely correct as noted by, e.g., Davidson (1985). The reason is that sometimes cycles are reported that actually are no cycles. The problem lies in the Log Filters' history that contains all old transactions. To work correctly the whole Log Filter would have to be reordered in some cases which would result in a bad worst case conflict detection time and thus other mechanisms for read/write cycle detection have to be developed. Because no mechanisms have been found that could be used directly in DeeDS this section discusses the support of read/write cycle detection without focusing a specific mechanism. Since all conflict detection approaches offer a version identifier for object versions all of them can be adapted to work with read/write cycle detection mechanisms that use versioning information to detect the cycles. For every committed transaction the read/write cycle detection has to keep version identifiers of all objects in the read- and write set. So, a good approach for read/write cycle detection mechanism has to use small version identifiers. There are two factors that have an influence on the version identifier size and so also on the read/write cycle detection. The first is the amount of information that has to be stored in the version identifier. If for example only the identifiers of the writing nodes have to be included in the version identifier these will be, at least in the average case, smaller than version identifier that include the identifiers of all nodes in the system, since normally not all nodes are writing. The second factor are mechanisms that can decrease the version identifier size over time. In the last section ("Real-

Time Characteristics") the first factor was considered. This section focuses on the second factor.
In the version vector approach, the history data is represented by the entries in the version vector. A value in the version vector is increased every time a node writes to a replica. At first view the version vector does not need more space for this because only the value of a variable is increased, but if nothing is done the variables will eventually overflow. One solution to avoid overflow is to use bounded version vectors (Almeida et al., 2004) that substitute the integer counters by bounded stamps.
Another issue in the version vector based approaches is the number of elements in the version vectors. The original version vector approach only worked with a fixed number of elements in every vector. But approaches like the Bayou version vectors or the DVVs that support a dynamic node set have to implement a way to remove old entries of their version vectors. The Bayou version vector mechanisms only support removal of entries in the version vectors of nodes that have left the system whereas the DVV mechanisms also support removal of entries belonging to nodes that are still in the system but have stopped writing to the replica. The idea in the DVV approach is that only versions of active writing nodes should be included in the version vectors.
The Hash History approach keeps a history of all versions. To compare two different replica versions the whole history of these replicas has to be compared. So there is a need to send these histories over the network. To avoid slowing down the whole system after a while it is necessary to limit the size of the history. One way to achieve this is to truncate the history, for example by the age of the entries or to set a fixed size for it and replace old entries with new ones. But it has to be ensured that the time for truncation or the space of the history are chosen correctly so that no entries are deleted which are still needed for conflict detection to work correctly. In more detail, old versions would be falsely regarded as new versions because they are not included in the history and would be added as new ones. The mechanisms of the version stamp approach remove old data automatically if two branches are united again. The main problem arises when branches are never joined, like in DeeDS. In this case the Version Stamps would increase without bound.

Summary: The most important characteristic of a conflict detection approach that is used in DeeDS is that it supports read/write transaction cycle detection in an efficient way. For this small conflict detection data structures are important. The Bayou VV approach can remove VV entries of nodes that leave the system. The dynamic VV approach can even remove entries of nodes that are still in the system, but requires consensus for its VV compression algorithm. Hash histories need a truncation algorithm that ensure that no necessary data is discarded. This is not an easy task if there are no time bounds on the update distribution in the network. Version Stamps cannot be reduced in a system like DeeDS where the replicas remain on different nodes.

Additional important characteristics

In the Hash History approach two different replica versions could incorrectly be considered equal, since two different replica contents could lead to the same hash value. Well known hash algorithms are used to overcome this drawback but still it cannot be completely avoided, since the contents of the replicas are reduced to one hash value. In a safety critical system like an airplane control system, even one error of this kind could lead to catastrophic consequences. For example if a new position value of an airplane would have been the same hash value assigned than an old position of it then the other airplanes in the observation range drop this position update because they regard it as old. This leads to an inconsistent state in which the positions in the database do not represent the correct positions in reality. As a consequence, airplanes could crash.

4.1.6.Result

This section evaluates the characteristics of the different conflict detection approaches with regards to their use in DeeDS. An overview of the most important characteristics can be found in Table 4.1.

Hash histories have many advantages when it comes to ID management since the IDs do not have to be maintained and can be calculated at any time from the contents of the replica. But this approach also has disadvantages. The large amount of memory needed to use hash histories together with read/write cycle detection mechanisms like Log Filters that have to save many histories and also the longer processing time needed to calculate the hash values make this approach unusable in some systems especially where resources are very limited. It is also not possible to use this approach in safety critical systems since hash values are not necessarily unique and so a conflict might not be detected as a result of this.

Version Stamps behave well if replicas can be reunified from time to time. The identifiers are shortened automatically during reunification; hence the required memory space can be limited. The processor utilization is not very high since there are no costly extra calculations beside the required ones for marshalling/unmarshalling and the comparisons. However if the replicas cannot be reconciliated, which is the case if the replicas remain on different nodes then the identifiers get larger and larger. One possibility to handle this could be to implement a consensus algorithm that reconciliates all replicas in the system and shortens their identifiers. But in a distributed system that contains partitions all nodes are perhaps never in the same partition at the same time.

The version vector based approaches are widely used, particularly in distributed file systems (Ratner et al., 1997). The basic version vector approach (Parker & Ramos, 1982) is not feasible in environments with dynamic set of nodes because of the fixed vector size. The Bayou version vector approach (Almeida et al., 2002) supports version vectors with a dynamic size. It can be used together with read/write cycle detection mechanisms like Log Filters and needs no extra resources beside the ones for the version vector comparison and the marshalling/unmarshalling. The main drawback is the continuously increasing identifier size because old IDs cannot be reused.

The approach considered to be the best for DeeDS is dynamic version vectors (Ratner et al., 1997). It supports a dynamic version vector size and only includes elements for nodes that actually wrote to the object in the version vector, which is an advantage, especially in systems where there are many read-only nodes. The approach can decrease the number of entries as well as the counter values in the version vector from time to time by executing the DVV consensus algorithm.

This algorithm is really useful to keep the version vectors small but it also needs additional resources and introduces new control structures to the version vectors that increase their size by at least a version number to every version vector. These are some drawbacks of the algorithm but the fact that the DVV approach does not need the DVV consensus algorithm to work correctly opens the possibility to execute the algorithm only in times of low system load which makes it a good optimization of DVVs.

Properties	**Basic Version Vector**	**Dynamic Version Vector**	**Bayou Version Vector**	**Version Stamps**	**Hash Histories**
Joining or leaving of nodes is supported	no	yes	yes	yes	yes
Nodes have to be uniquely identified	yes	yes	yes	no	no
support for read/write conflict detection	good	good	good	resource intensive ^{1})	resource intensive ^{2})
Worst case conflict detection data size	all nodes in the system	all nodes ever wrote to the object	all nodes ever been in the system	unbounded ^{1})	ids of all updates

1) version stamps grow unbounded in size if the replicas stay on different nodes and are not reconciled
2) For every replica version a history of identifiers has to be saved. If this history is not shortened efficiently much memory space will be required.

Table 4.1. Characteristics of the conflict detection approaches

4.2. Dynamic version vectors in DeeDS

Dynamic version vectors (DVVs) together with the DVV consensus algorithm have been implemented successfully in the Roam system (Reiher, 1999), which is an optimistically replicated file system that can be used in mobile environments. In this chapter the differences of both systems are highlighted and parts of the DVV approach that have to be adapted to make this approach work with DeeDS are identified. Solutions for adapting the identified parts are also described.

4.2.1.Roam vs DeeDS

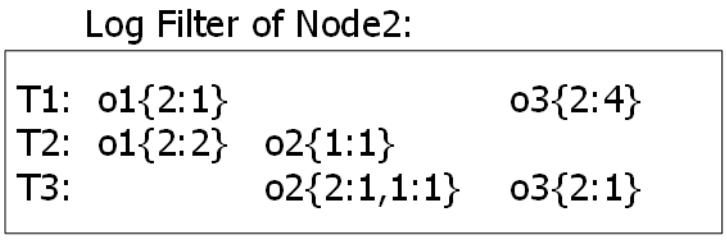
Both Roam and DeeDS are distributed and use optimistic replication to share their data. They are not dependent on a server infrastructure and keep working during network partitions. The most significant difference between the systems in terms of conflict detection is that DeeDS, in contrast to Roam, supports transactions. That means that the replication module of DeeDS has to support transactions. Conflict detection and resolution are used by both Roam and DeeDS to ensure eventual global consistency. In Roam it is sufficient that write/write conflicts (chapter 2.5.1) are detected but not in DeeDS. The support of transaction introduces new conflict types that have to be detected, such as the read/write cycle (chapter 2.5.1). For this, DeeDS has to use a read/write cycle detection mechanism, currently Log Filters. So, there is a need to bring the conflict detection approach of Roam together with read/write cycle detection. This can be either done by changing the conflict detection approach or by changing the read/write cycle detection mechanism. But it is not only the read/write cycle detection that has to be changed for adapting DVVs in DeeDS. There is also a need to change the data structure that represents the DVV and the algorithms that work on that data structure. DVVs are variable in size and so it is not possible to work with static data structures that can be maintained and accessed more efficiently than dynamic ones. An algorithm that has to be changed completely is the comparison algorithm because it is closely connected to the version vector structure. Because this algorithm is used very often during conflict detection it is important to optimize it for a specific conflict detection data structure.
Another difference between Roam and DeeDS is that the data exchange to preserve consistency is done pair wise between nodes in Roam and the version vectors are updated only during this pair wise reconciliation. The update of version vectors is done in the first phase by scanning the local files. If the modification date has been changed since the last scan the corresponding version vector is updated. In the DeeDS database there are no modification timestamps, so update message have to be sent to the other replicas that have to be informed right after each commit. The version vectors are also updated after each transaction commit.

4.2.2.Dynamic version vector adaptation

In the previous section the parts of the DVV approach that have to be adapted have been identified. Below solutions to adapt these parts are discussed. It has to be mentioned here that although Log Filters suffer from some flaws (see chapter 4.3) they have been adapted to work with DVVs. The reason is that no other read/write cycle detection mechanism has been found that could replace Log Filters in DeeDS at the moment and the results of this adaptation could be also helpful for the adaptation towards another read/write cycle detection. The development of a new conflict detection approach that is able to detect all kinds of conflicts has been started to replace Log Filters. But this isn't done within this work.

Adaptation of the dynamic version vectors
To integrate DVVs in DeeDS the data structure for the DVVs has to be defined. To be flexible and eliminate the marshalling and unmarshalling time a serialized data structure has been chosen. Since most of the operations on the DVV are comparisons the entries stored in that data structure are ordered to enable a more efficient compare algorithm. The compare algorithm also has to be aware of the missing entries in the DVV. Nodes that never have written to a replica are not included in the associated DVV. During comparison these missing entries are seen as a nodeID/counter pair whose counter is zero. If there is a need for a non-serialized version of the DVVs marshalling/unmarshalling algorithms have to be created that are specific to DVVs. All adaptations that have been described in this part have been implemented in DeeDS. More implementation related information can be found in chapter 5.

Adaptation of the DVV consensus algorithm
The DVV consensus algorithm optimizes the DVV approach by reducing the number of entries within one version vector as well as decreasing the counters in the vector. Entries of nodes that stop writing to an object can be removed from the version vector of this object. But even if a node keeps on writing to an object its counter can be decreased in the version vector of that object. This increases the performance of the system because the data that has to be sent over the network and the memory space requirements are reduced. Algorithms like the compare algorithm also show better performance because they have to compare fewer entries. This algorithm works well in a system like Roam where it is possible to compress all DVVs in the system. In DeeDS, however, there are Log Filters on every node that also include DVVs. To ensure correct conflict detection it is necessary to compress the DVVs in the Log Filter in the same way as all the other DVVs in the system. Since there are DVVs in the Log Filter that represent replica versions included in the read- or write-set of former transactions the DVV consensus algorithm could lead to negative counter values in these DVVs. If these entries just would be removed, relations among version vector sequences would be lost. The following example shall lead to a better understanding of this issue. The three transactions T1, T2 and T3 (figure 4.4) can not be ordered in the Log Filter because they form a read/write cycle. T1 dominates T3, T3 dominates T2 and T2 dominates T1. Looking at the Log Filter of an arbitrary node in a system like DeeDS (called node 2 in the example) it could contain such a read/write cycle (figure 4.6) which would be detected by running a read/write cycle detection. The elements in the example can be read as following: *object-ID{node-ID:counter-value}.*


Log Filter of Node2:

T1:	o1{2:1}		o3{2:4}
T2:	o1{2:2}	o2{1:1}	
T3:		o2{2:1,1:1}	o3{2:1}

Figure 4. 6 Log Filter without compression

Assuming now, that T2 has been integrated into all replicas of object 1 (o1) in the system the version vector of every of these replicas contains an entry belonging to node 2 with at least a counter value of 2. This makes it possible for the DVV consensus algorithm to decrease the counter by 2. As mentioned above the version vectors in the Log Filter also have to be compressed by the DVV consensus algorithm. If version vector entries would be removed whose counter values are 0 or negative the version vectors of object 1 (o1) would disappear and the Log Filter would look like shown in figure 4.7. But without the version vectors of o1 the three transactions do not form a cycle anymore. They can be ordered because the dependability between T1 and T2 has been removed. So, if a read/write cycle detection would be performed no cycle would be detected which would be incorrect.

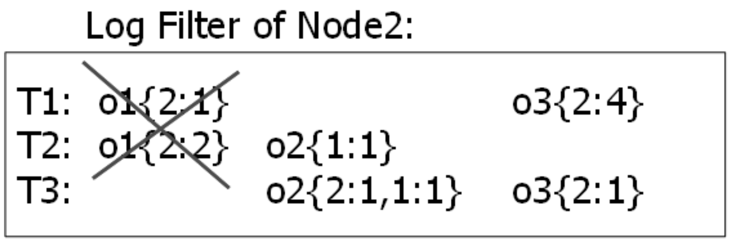


Figure 4. 7: Compressed Log Filter

One solution to adapt the Log Filter to work with the DVV consensus algorithm is to allow negative counter values in the Log Filter, but then the DVVs in the Log Filter could not be compressed. Another solution is to keep the relations between the transactions on an object granularity. In other words, the dependencies of the transactions in the log filter would be saved within the log filter data structure and so relations between already integrated transactions would not disappear. This would allow compression of the Log Filter but a new Log Filter data structure would be necessary.

4.3. Read/Write cycle detection

Initially, it has been assumed that Log Filters could satisfactorily be used to detect read/write transaction cycles. Unfortunately this is not the case because this approach detects not all conflicts and also falsely detects conflicts (Davidson et al., 1985).
However, transaction graphs (Davidsson et al., 1985) can be adapted for DeeDS via modifications that allow continuous graph construction using dynamic version vectors. Due to the limited time this work is out of scope of this thesis.

5.Implementation of dynamic version vectors in DeeDS

In this chapter, the implementation of the dynamic version vector (DVV) approach for DeeDS is described in more detail. The replication module has been redesigned to separate conflict detection from conflict resolution. This is not required to implement DVVs but is necessary to support application-specific conflict resolution. The new design is one step towards supporting more than one conflict detection mechanism during runtime.

5.1. Software Design

This section describes the new design of the replication module. It first shows where the replication module is situated within DeeDS and then describes the different components of the module in detail.

Overview

The Replication Module is situated between TDBM, which is the data storage system in DeeDS and DOI which acts as an interface to real-time operating systems like OSE Delta (Fig. 5.1). Both TDBM and DOI are described in more detail in chapter 2.2.2. The applications which run on top of TDBM use functions in TDBM to create and commit transactions. Within TDBM there are hooks to the replication module. For example, the commit function notifies the replication module so that it can replicate the committed transaction to all other nodes.

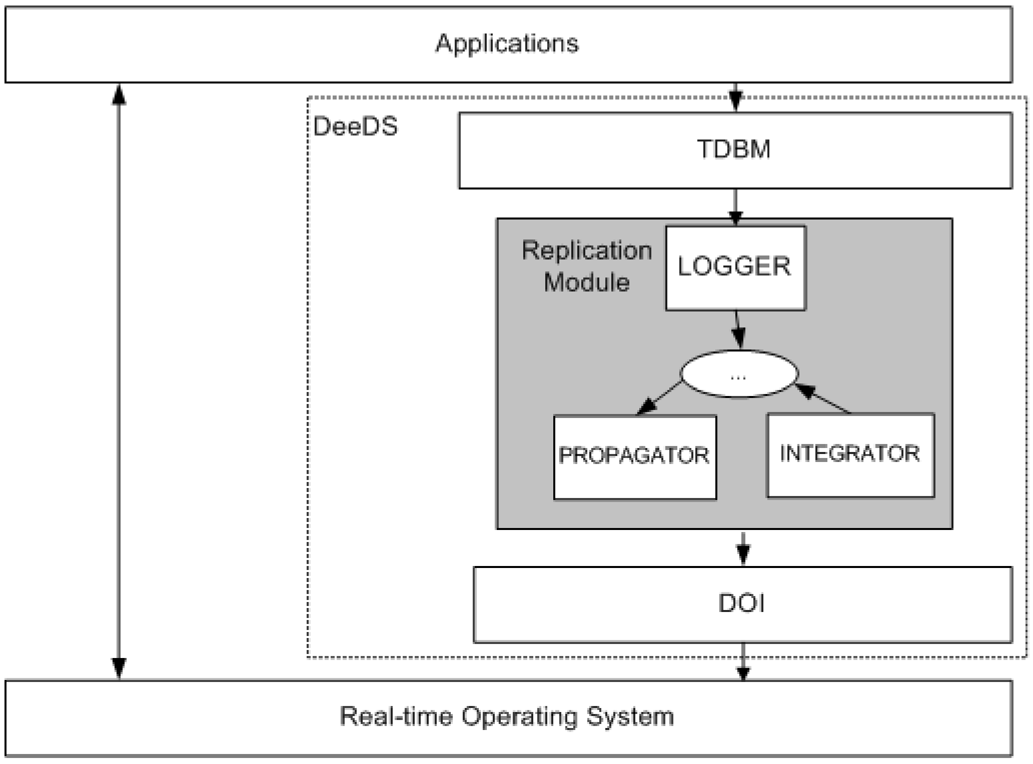


Fig.5. 1 Overview of the DeeDS system

The interface between the Replication Module and TDBM is the Logger, which contains all the functions that can be called from within TDBM. The Propagator and the Integrator are interfaces to the network and use DOI functions to send and receive messages.
Figure 5.2 gives a detailed overview of the relations among the components in the Replication Module.

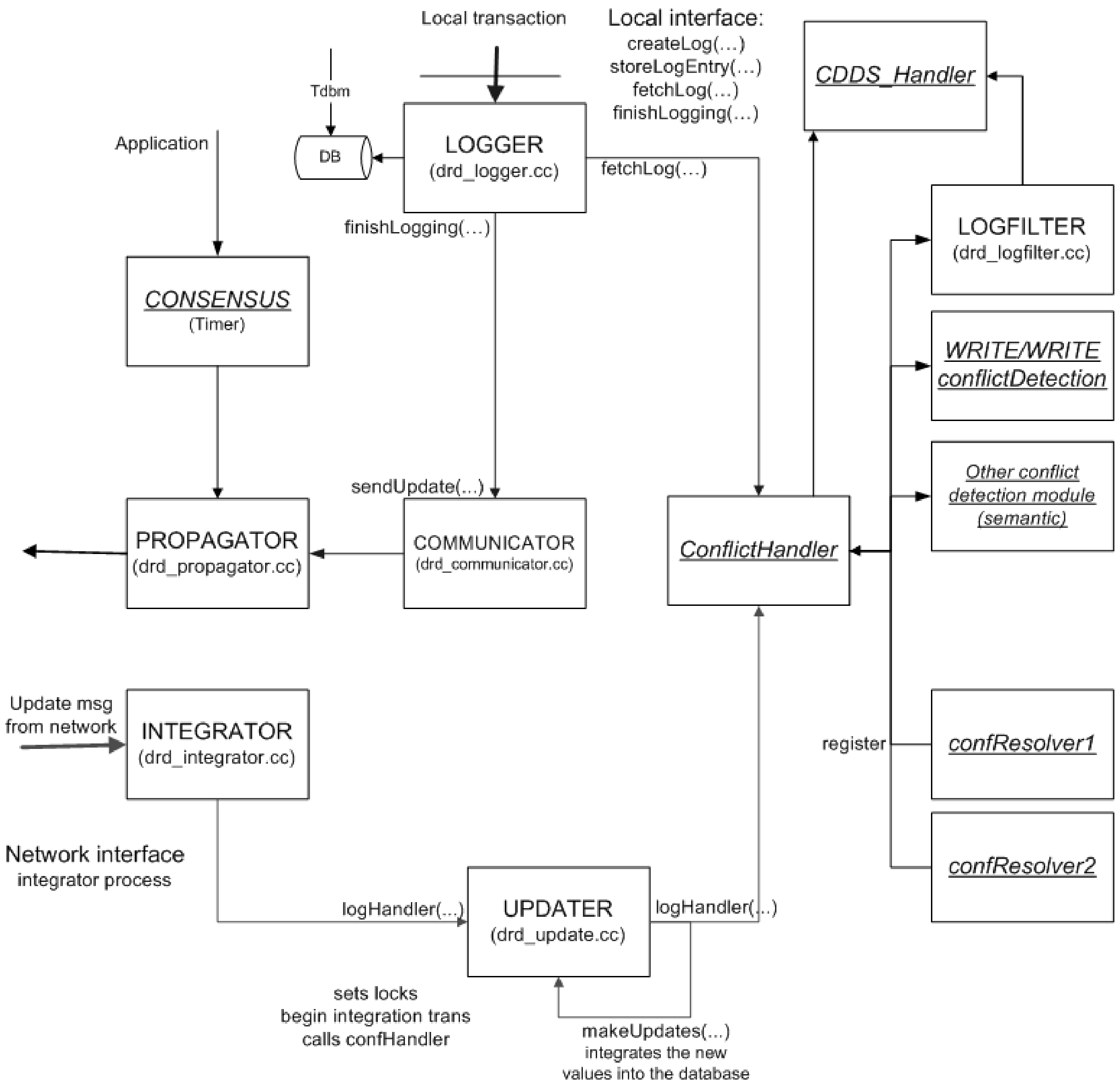


Fig.5. 2 The Replication Module

Logger

The logger is called from TDBM every time an operation on the database is performed by an application. The Logger creates a log for every transaction in the system containing the data that is read or written, and the Conflict Detection Data Structure (CDDS) (see 5.2.2), which is needed to detect conflicts. If a transaction is committed locally the logger creates a byte stream from the transaction’s log and passes it to the Propagator, which sends it to all other nodes that have to be informed. It also updates the local conflict detection data structures by calling the Conflict Handler so that these also stay up to date.

Conflict Handler

The conflict handler is accessed by the Updater. It gets all information about the incoming update message as an unchanged byte stream and gives back a vector including all keys and values that have to be stored in the local database. To efficiently detect and resolve conflicts this Byte Stream is parsed and a Conflict Detection Data Structure (CDDS) is created by calling the CDDS Handler. This CDDS is then passed to the conflict detection component, which is at the moment the Log Filter but the design is open for more than one conflict detection component. If there are more than one conflict detection component, a decision policy has to be implemented which selects the right conflict detection component for every transaction that has to be integrated. If conflicts are detected, the Conflict Handler is notified and so can start conflict resolution. Like for conflict detection the design supports multiple components for conflict resolution. If forward conflict resolution is used the resolution of conflicts is often application dependent and so it is important that multiple conflict resolvers are supported. The idea is that applications can integrate their own conflict resolution components in DeeDS. Also in this case there is a need to add a policy which selects the correct conflict resolver. After all conflicts are detected and resolved the result, keys and values that were changed are saved in a vector which is given back to the Updater.

Updater

The Updater integrates transactions that have been performed on remote nodes into the local database. Before the actual integration takes place it has to be checked if the new transaction conflicts with any former transaction which has already been integrated. This conflict detection is performed by the Conflict Handler, which is called from the Updater before the write-set of the new transaction is written to the database. Storing the changed values finishes this function and thus also the integration process of the new transaction.

Integrator

The Integrator receives incoming messages and delegates them to the required components. In the case of conflict detection and resolution it receives the update messages, which include transactions that were performed on other nodes and have to be integrated to achieve eventual consistency. If such an update message is received by the Integrator the, Updater is called.

Communicator

The communicator has no special task at the moment. But this component could be used, for example, by the consensus algorithm to send messages over the network. The communicator passes all messages it gets to the propagator that actually sends them.

Propagator

The Propagator is used to send update messages to other nodes. If the local node commits a transaction, the other nodes in the system have to be informed to support eventual consistency of the whole system. The contents of an update message, which have been serialized by the logger, is sent to the Propagator, which then creates the message and sends it to every node in the system that has to be updated.

CDDS Handler

The Conflict Detection Data Structure Handler (CDDS Handler) hides everything which is specific for the Conflict Detection Data Structure (CDDS). The reason for this is to get a cleaner design that separates the code for maintaining the CDDS from the code for resolving and detecting conflicts. This also enables easy changing of the CDDS because the CDDS Handler simply has to be replaced by another handler that supports all the operations to maintain the new CDDS. The CDDS Handler contains procedures for marshalling, unmarshalling and updating the CDDS. For this thesis a CDDS Handler that supports DVVs has been created.

Log Filter

The Log Filter is used for detecting write/write conflicts and read/write cycles.
It stores the CDDSs of all transactions that have been performed on the database in its data structures, which are described in chapter 5.2. After the integration process of a new transaction the Log Filter checks if all the CDDS it contains can be ordered. If they can not be ordered there has to be a read/write cycle. Write/write conflicts are automatically detected when a new CDDS is added to the Log Filter by the CDDS compare function explained in chapter 5.3 in more detail. As mentioned before, Log Filters are not working correctly and that’s why this component has to be replaced by a component including the implementation of a correct read/write cycle detection mechanism.

Resolver

A conflict resolver is called after a conflict has been detected. As described before the only way of resolving a conflict in a system like DeeDS is by forward resolution. Rolling back is not possible because the transaction has already optimistically committed on other nodes. The conflict thus has to be deterministically resolved to ensure that all the nodes get the same value after integrating the same transactions. For this thesis an average resolver for write/write conflict resolution has been implemented in DeeDS. The resolver takes the values of conflicting updates and calculates the average value of these. Additionally a new CDDS is created that dominates all conflicting CDDS. This new CDDS together with the calculated average contains all the information of the original conflicting updates and is returned to the Conflict Handler. The fact that the average resolver calculates the average makes it only feasible for numerical values. The resolver currently implemented in DeeDS accepts only integer values as inputs but could easily extended to accept any numerical values.

5.2. Data Structures

In this chapter the data structures used for conflict detection will be described in more detail. In the first part, serialized data structures, which are used for inter node communication and data storage, are explained. The second part addresses non-serialized data structures that are used for information exchange between different components in one node.

5.2.1.Serialized data structures

Dynamic version vector

A DVV is stored with every value in the database to keep information about already included updates. To store the vector in a database it has to be in a serialized form.

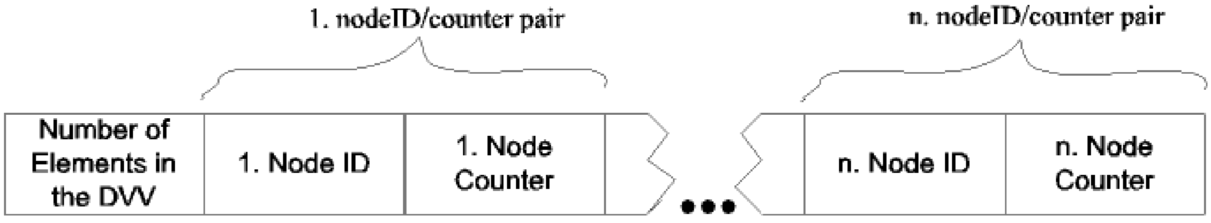


Fig.5. 3 Serialized dynamic version vector

Because the DVV does not have a fixed size the serialized form of it (Fig.5.3) starts with its size (in number of elements). This number is used by the compare algorithm that is implemented in the CDDS Handler to find the end of the version vector. After the size, nodeID/counter pairs are stored containing information about how many times the associated replica has been written by a specific node. Every value stored in the database is prefixed by a serialized CDDS (currently a DVV). If the user accesses a value in the database this CDDS is removed and only the value is returned to the user. If the value is written by an application the CDDS is updated and both CDDS and value are stored in the database. CDDS maintenance is completely hidden from the application, which only stores and reads values.

Update Stream

The update stream is included in every update message that is sent by the propagator to inform other nodes about a transaction. It contains all information necessary for a remote node to integrate the updates of the transaction to its local database. Within a node, this data structure is used to inform the conflict detection mechanisms that a new local transaction has been committed. Fig. 5.4 shows the structure of the update stream that starts with the number of objects included in the read- and write-set of the transaction.

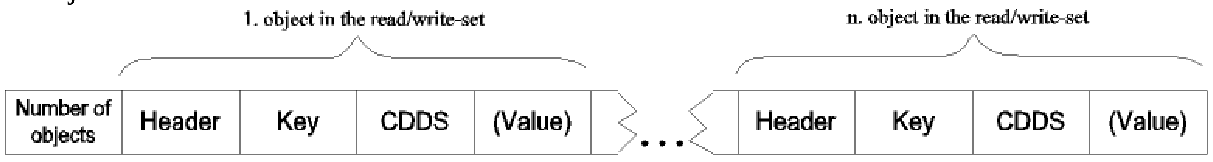

1. object in the read/write-set
n. object in the read/write-set
Number of objects
Header
Key
CDDS
(Value)
Header
Key
CDDS
(Value)

Fig.5. 4 Serialized CDDS Sequence

For every object a header, key and CDDS is stored. The value is appended if this object is in the write-set of the transaction. The header contains the key, CDDS and value sizes as well as an indicator in which set (read or write) the object is included. Also the pathname of the TDBM file is stored in the header.

5.2.2.Non-Serialized data structures

Conflict detection data structure
A conflict detection data structure is created during the unmarshalling process. It includes versioning information like a version vector or a hash history depending on which approach is used. The versioning data structure is stored in a serialized form to keep the design open for different versioning approaches. It is possible to create a specialized CDDS that is more efficient, but then all functions that operate on this specialized data structure should be hidden in the CDDS Handler. For this thesis the DVV approach has been implemented and so a serialized DVV (5.2.1) is used.

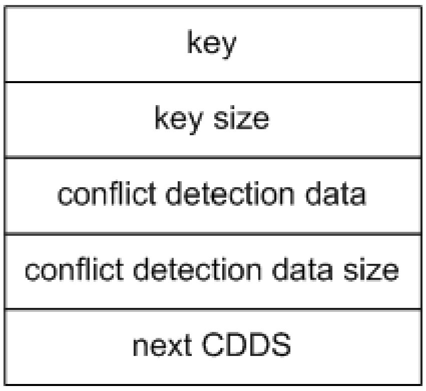


Fig.5. 5 Conflict Detection Data Structure (CDDS)

The different fields of the CDDS are shown in Fig. 5.5. The “key” field contains the identifier of the object to which the CDDS belongs. Because different types of object identifiers are supported in DeeDS, the size of the key has to be stored. The conflict detection data field contains a pointer to the serialized versioning data structure like a serialized DVV. These data structures have no fixed size if they support a dynamic node set and so the next field contains the size of the versioning data structure. The last field contains a pointer to the next CDDS. This is used for storing more than one CDDS in a CDDS sequence.

Conflict detection data structure sequence
A *conflict detection data structure sequence* contains all information about a transaction that is needed for conflict detection. It includes the CDDSs of all objects in the read- and write-set of the represented transaction. This data structure contains three fields (Fig.5.6): pathname, head and next. Pathname contains the full path of the TDBM file to which the objects in the read- and write-set belong. The head field contains a pointer to the first CDDS in the sequence. The next pointer is used only in the Log Filter to create a linked list of sequences.

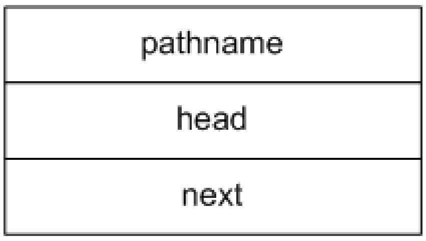


Fig.5. 6 Conflict Detection Data Structure Sequence

5.3. Algorithms

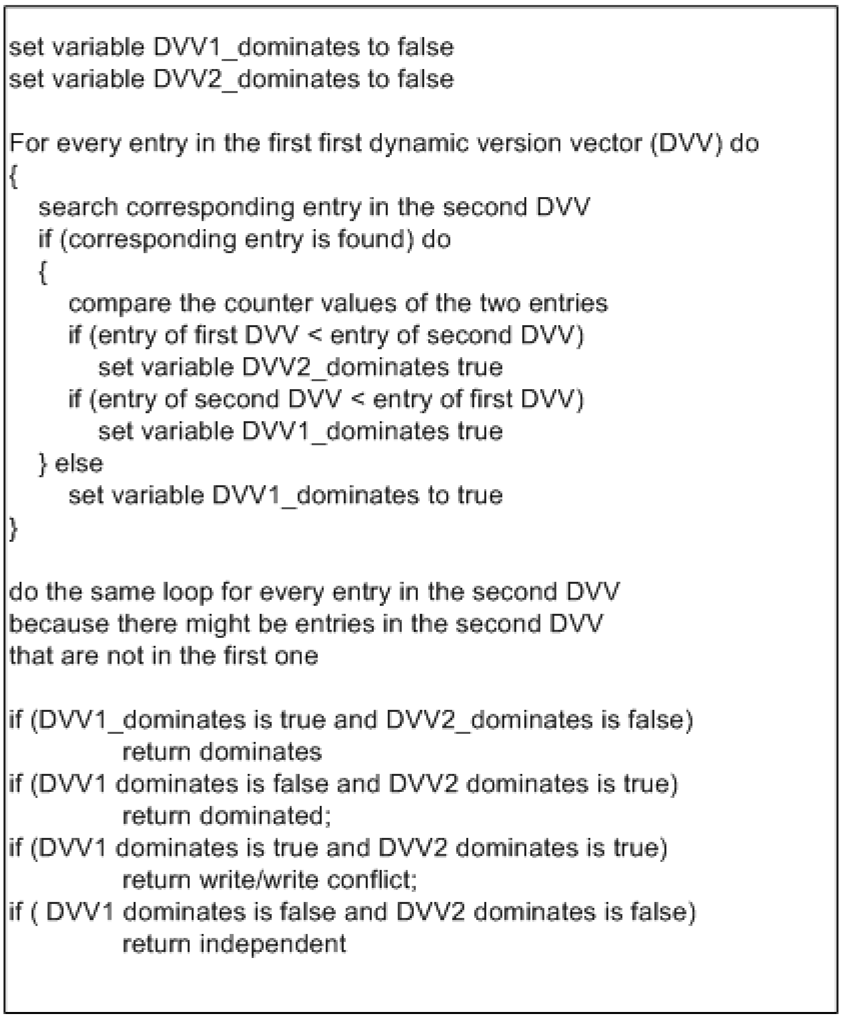
Algorithm for comparing dynamic version vectors
During conflict detection CDDSs have to be compared to determine which updates are already included in a replica version. Since the compare algorithm is dependent on the type of CDDS it is implemented in the CDDS Handler. In the rest of this section the compare algorithm will be explained for DVVs.
The compare algorithm compares two DVVs by comparing all the version vector pairs of one DVV with the corresponding ones of the other DVV. Two pairs are corresponding to each other if their node ID is equal. If during comparison a pair (N_n, C_n), where n and m are replica identifiers, has no corresponding pair at the other DVV, (N_m=N_n, 0) is taken as the corresponding pair during comparison. So if a pair for a specific node ID is missing this means that this node has not written to the replica and is dominated by all pairs with counters greater than zero. After the compare algorithm has compared all corresponding pairs it outputs the type of relation between the two DVVs. The relation types are described below.

A dynamic version vector (DVV) V_n dominates another dynamic version vector V_m if and only if for all $N_{n,i} = N_{m,j}$ $C_{n,i} > C_{m,j}$,where $N_{n,i}$ & $N_{n,j}$ are node-IDs in V_n and V_m and $C_{n,I}$ & $C_{m,j}$ the corresponding counters.
The dominate relation between two DVVs can be seen as a happened after. So if a DVV V_n dominates another one V_m than V_n represents a newer version of the object than V_m.

Two dynamic version vectors V_n and V_m are equal if their sets of version vector pairs are equal.
If this is the case, the two version vectors represent the same version of a replica. This means that all updates included in the replica represented by V_n are also included in the replica represented by V_m.

Two dynamic version vectors are in conflict if there is at least one pair that dominates the corresponding pair in the other dynamic version vector and additionally at least one pair that is dominated by the corresponding pair.
This represents a write/write conflict, which arises when two different nodes have updated their replica of the same object without first integrating the other’s updates (see chapter 2.5.1).

These are all possible results of the comparison algorithm. The pseudo-code of the algorithm can be seen in Fig.: 5.7.



```
set variable DVV1_dominates to false
set variable DVV2_dominates to false

For every entry in the first first dynamic version vector (DVV) do
{
   search corresponding entry in the second DVV
   if (corresponding entry is found) do
   {
      compare the counter values of the two entries
      if (entry of first DVV < entry of second DVV)
         set variable DVV2_dominates true
      if (entry of second DVV < entry of first DVV)
         set variable DVV1_dominates true
   } else
      set variable DVV1_dominates to true
}

do the same loop for every entry in the second DVV
because there might be entries in the second DVV
that are not in the first one

if (DVV1_dominates is true and DVV2_dominates is false)
            return dominates
if (DVV1 dominates is false and DVV2 dominates is true)
            return dominated;
if (DVV1 dominates is true and DVV2 dominates is true)
            return write/write conflict;
if ( DVV1 dominates is false and DVV2 dominates is false)
            return independent
```

Fig.5. 7 Dynamic version vector compare function

Algorithm for comparing dynamic version vectors sequences
The compare algorithm for DVVs compares all corresponding version vector pairs of two version vector sequences. So if both DVV sequences have the same object in their read- or write-set there is a matching pair that has to be compared by the algorithm.
After all corresponding pairs have been compared the algorithm returns the relation between the two version vector sequences similarly to the DVV comparison. If a write/write conflict is found during one of the DVV comparisons this conflict has to be resolved, and then the sequences have to be compared again. A DVV sequence can dominate or be dominated by another sequence. During the comparison of sequences a new type of conflict can occur namely a read/write cycle. This type of conflict arises if a DVV of sequence S1 dominates a corresponding one in sequence S2 and another DVV in sequence S1 is dominated by a corresponding DVV in sequence S2. Figure 5.8 gives an example of DVVs creating a read/write cycle. If a DVV of one sequence has no corresponding one on another sequence no comparison has to be performed. In the case where there are no corresponding pairs of DVVs of two sequences these sequences are independent. Independent sequences have no causal relation and can be placed in any order.

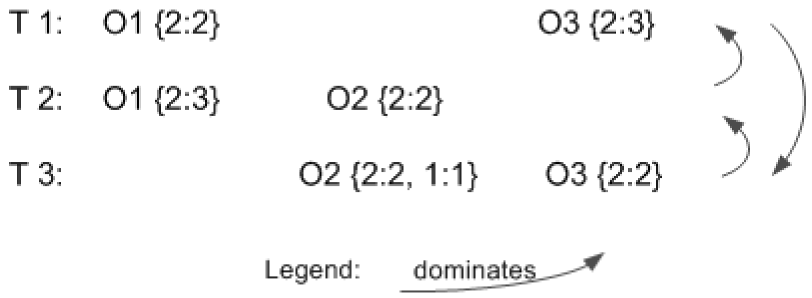


Fig. 5.8: Read/Write cycle

5.4. Examples

This Chapter shows the behaviour of the Conflict Detection and Resolution by two examples. In the first example, a local transaction is committed, and in the second example an update message is integrated into the local database.

Transaction commits locally

This example focuses on the actions that take place when a local transaction commits. Fig.5.3 gives an overview of the whole process. The logger is informed if a new transaction is created in TDBM. It creates a log where all information about the read- and write-operations performed by the transaction is stored.
When the transaction is committed the logger is informed about the commit and performs the following actions. First, it fetches the log it created before from the database. After this it creates a byte stream that contains all the operations of the transaction and additionally all data structures that are necessary for conflict detection. In DeeDS these data structures are appended to every value in the database, and are automatically fetched when the value is accessed. The created byte stream is used for two things. First, it is used to update the local conflict detection component. In our example this is the Log Filter. The components for conflict detection and resolution are never accessed directly. All access to these data structures is made through the Conflict Handler which gives the possibility to easily change the detection and resolution mechanisms. The Conflict Handler also parses the byte stream and creates the Conflict Detection Data Structure (CDDS), which is more efficient to handle

during comparison than the original byte stream.
The second thing the Logger does with the byte stream is send it to the Propagator, which creates a propagation message. This message is sent to all nodes in the system that have to be informed about this transaction.

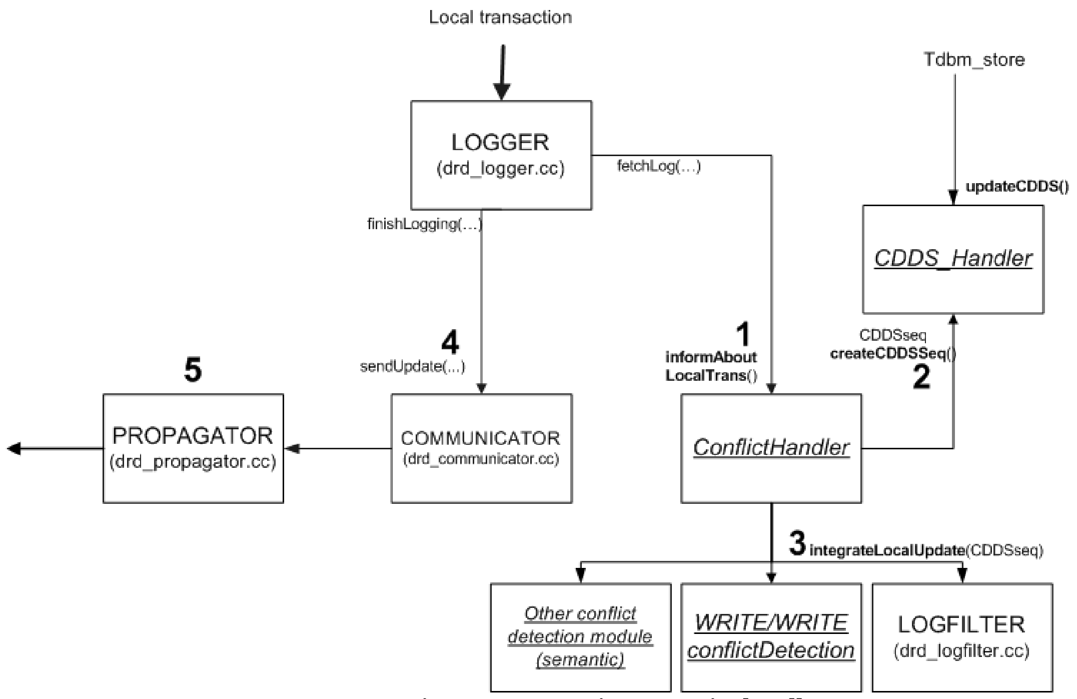


Fig.5. 9 Transaction commits locally

Transaction integration

This example shows what happens when an update message arrives (Fig.5.10). Update messages always include exactly one transaction that was committed on another node. The message first arrives at the integrator, which checks what type the message has. If it is an update message for the database it redirects the contents as a byte stream to the Updater. The updater sets the locks for the database and then starts an integration transaction. There are differences between an integration transaction and a local transaction. An integration transaction, for example, is not replicated. After this, the Conflict Handler is called, which parses the byte stream and creates the CDDS by using a procedure of the CDDS Handler. This data structure is used in all the following operations because it contains all the information of the original update message. To detect write/write conflicts and read/write cycles the Conflict Handler passes the CDDS to a conflict detection component which is in our case the Log Filter. The Log Filter checks if the new transaction introduces new conflicts by adding the CDDS to the history graph and then checking for cycles in it. If conflicts occur, these are reported to the Conflict Handler, which then calls the required conflict resolvers. In our case this is the average resolver that takes integers as input values and calculates the average value. Next, the conflict resolver notifies the Conflict Handler whether the conflict was resolved. If all went right the conflict resolver also returns the resolved value. The Conflict Handler waits until it has received all the resolved values and then returns these to the Updater, which integrates them in the local database.

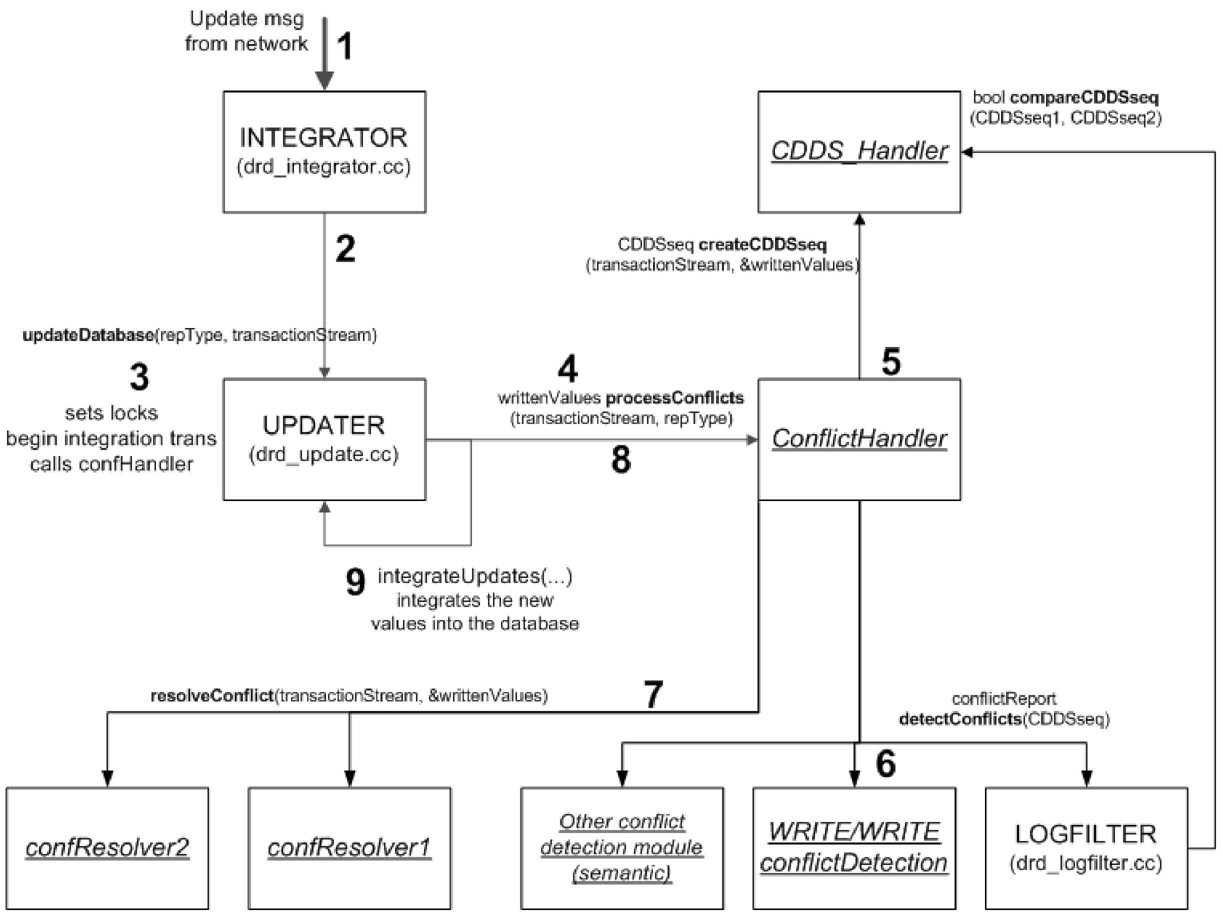


Fig.5. 10 Transaction has to be integrated

5.5. Test Cases

Within the scope of this thesis the DVV approach has been integrated in DeeDS. To increase reliability regression tests were created that help to ensure correct functionality of the different parts of the implementation. It shall also be assured that data stored in the database is not corrupted by the conflict detection module. The regression tests can also be used for controlling further changes within the conflict detection module of DeeDS. By executing these tests after every change, faults that were introduced with the change can be found and removed. Appendix A contains more information about the different test cases that were created.

6.Discussion & Conclusion

This chapter summarizes the work done for this thesis and points out the results. There is also a discussion about the chosen conflict detection approach and its implementation. The second part discusses the design of the replication module that divides the conflict detection from the conflict resolution.

6.1. Summary

In chapter 3 three aims are defined for this work. The first aim is to find a conflict detection approach that can handle a large and dynamic node set. The second is to implement the best fitting conflict detection approach and the last aim is to divide conflict detection from conflict resolution. This makes it possible for applications to integrate their own conflict resolution mechanisms in DeeDS. The last two aims are discussed together in this chapter because both of them have been influenced the new conflict detection design.

6.1.1.Conflict Detection Approach

The dynamic version vector (DVV) approach has been considered to be the best fitting approach for a distributed, active, real-time database with transaction support like DeeDS. This decision has been made for the following reasons:

1) **A dynamic node set is supported.**
2) **The conflict detection data structures are small in the worst- and average case.**
3) **The approach can be adapted to efficially handle transactions.**
4) **Optimizations for the average case are available (DVV consensus algorithm)**

The first reason seems to be trivial but it is an absolutely necessary criterion for a conflict detection approach to be able to work in an environment with a dynamic node set. So, all approaches with a fixed conflict detection data structure size like the basic version vector approach are not suitable for DeeDS. The second reason concerns the size of the conflict detection data structure that also affects the runtime of the conflict detection algorithms. Firstly, a limited size is required for predictability reasons. Since the size of version stamps increase with the number of replicas and cannot be decreased if replicas are never really merged this approach cannot be used until a solution has been found to fix this. Because of efficiency reasons it is also required that there are as few entries as possible in the conflict detection data structure. In the DVV approach only the writing nodes are included which is a real advantage in environments where most of the nodes are only readers (Ratner et al., 1997). DeeDS supports transactions and so conflict detection for read/write cycles also has to be performed. It has been mentioned that no correctly working read/write cycle mechanism has been found that can be used in DeeDS. But since such a mechanism is absolutely necessary in DeeDS and the chosen detection approach has to work with it the transaction handling was one comparison criteria. The version vector based approaches show the best characteristics towards read/write cycle detection because their average version identifier can be kept small which is very important since many version identifiers have to be saved by the read/write cycle detection (chapter 4.1.5). The other approaches are also adaptable to the read/write cycle detection but the resource usage would be higher. Basic version vectors excluded, no system has been found where one of the conflict detection approaches has been implemented together with read/write transaction cycle detection. Finally it is good to have the possibility to optimize the approach which can be done by adding the DVV consensus algorithm to the DVV approach to reach better efficiency since the average size of the conflict data structures

is reduced.
The DVV approach shows good characteristics for handling a large dynamic node set, can be easily adapted to a read/write conflict detection mechanism like Log Filters and is efficient in the average case. But the approach also has some issues that should be mentioned and discussed:

1) **If an unbounded writing node set is considered the version vector is unbounded.**
2) **From where and when should the DVV consensus algorithm be started.**
3) **Read/write cycle detection has to be changed to work with the conensus algorithm.**

In an environment with a dynamic node set in which the system has to run forever or at least for a long time it might be the case that the writing node set is very large or even unbounded since new nodes join the network over time. In this case the DVVs would be unbounded if nothing is done. One possibility to avoid this unbounded increasing is to execute the DVV consensus algorithm in a predictable way to limit the size of the version vectors. But then the DVV consensus algorithm might have to be executed also in times of high system load. In general, the DVV consensus algorithm has to be examined in more detail. The time of execution, the node that starts the algorithm and the amount of entries that should be compressed are variables of interest. The DVV consensus algorithm has not been implemented during this work due to time constraints but it would be of interest to show in practice how the mentioned variables influence the behaviour of the algorithm. Ratner et al. (1997) mentions, that the place from where the DVV consensus algorithm is started or the time and frequency of the execution might have a huge impact on the algorithm.
As described in chapter 4.1.5, the DVV approach shows good characteristics towards read/write cycle detection support. If the DVV consensus algorithm is integrated, the used read/write cycle detection approach, like Log Filters, has to be adapted. In addition to the version vectors stored in the database, all version vectors that are kept in the read/write cycle detection data structures have to be compressed by the DVV consensus algorithm to ensure correct behaviour. As shown in section “Adaptation of the DVV consensus algorithm” in chapter 4.2.2, the DVV consensus algorithm cannot be run on just the Log Filter, because relations among transactions could get lost. So, an additional feature of a new read/write cycle detection mechanism would be to keep the relations in the data structures separated from the versioning identifiers.

6.1.2.Implementation of the new conflict detection design

The first decision for the implementation was whether to use the existing code for conflict detection in DeeDS or to create a completely new design for conflict detection. To not rely on the existing code and to be able to realize all new ideas like the encapsulation of the used CDDS approach the choice has been a completely new design. After finishing the design the existing code was inspected and fragments which could be easily modified to work in the new design were identified. The main features of the new design are the following:

1) **encapsulation of the used conflict detection approach**
2) **division of the conflict detection and conflict resolution**

The first feature makes it easy to replace the used conflict detection approach. This is achieved by hiding all approach-specific functionality in one class. To change the used approach a new class with the same interface has to be created which can then be used to replace the existing one.

The second feature of the new design is a first step towards application-specific resolution support. Conflict detection and conflict resolution are divided into two separate parts by introducing a conflict detection interface and a conflict resolution interface. The conflict handler performs the conflict detection and if necessary also the conflict resolution by using these interfaces. Register and deregister functions in the conflict handler are prepared but not yet implemented to allow applications running on DeeDS to introduce their own resolution functions.
One problem that has to be solved if multiple conflict detection and resolution functions are used is the decision maker that has to choose the right conflict detection or resolution function. The task of creating such a decision maker is not trivial. It first has to be decided on which granularity it makes sense to choose which function to call. A conflict detection or conflict resolution function could be associated with an application, a transaction, or with something of finer granularity like an object. Secondly a data structure has to be created that keeps information about which conflict detection or resolution function to use for which replica version or conflict. Because this information is needed every time a transaction is integrated the data structure should be optimized towards efficient accessibility. Lastly, the necessary information the decision maker needs should be made available. Possibly, the conflict detection data structure has to be extended for this to include, for example, application specific information. The DVV approach has been implemented in DeeDS and tested for correctness.
For the new design one class has been created which includes the DVV approach. This class has been integrated in DeeDS and tested for correctness.

6.2. Discussion

6.2.1.Version vector data structure

In the current implementation, a serialized data structure is used for the DVVs. In general, serialized data structures are not very efficient to work with but the decision to use it was made for the following reasons:

1) **conflict detection data type independence**
2) **no marshalling/unmarshalling**

The new design separates the conflict detection data structures and its specific functions completely from the rest of the conflict detection mechanisms. To not limit the possible conflict detection approaches which can be used with DeeDS only Byte Streams are exchanged between the different components of the conflict detection. An additional reason was that only serialized data types can be sent over the network and stored in the database. So if non serialized data types would be used there would be marshalling/unmarshalling overhead which can be avoided or at least reduced by using serialized forms.
If there is a need for a more efficient data structure the conflict handler could be used as a place to create such a data structure. For testing reasons a map has been created representing a DVV.
During the literature study of this thesis it has been found out that a lot of research has been done in the field of write/write conflict detection. Unfortunately there are only few resources available that focus on read/write cycle detection. One explanation could be that this kind of conflict is not present in many distributed systems like file systems. Another reason could be that transaction support is not included in many distributed systems.

6.3. Further work

To optimize the DVV approach the DVV consensus algorithm should be implemented in DeeDS. This algorithm can increase the efficiency of the whole conflict detection (Ratner et al., 1997) especially if the database is running a long time and many updates are made the version vectors have to be compressed from time to time. Additional information that is needed for the algorithm could be stored in the database itself. Because the consensus algorithm can be seen as an optimization it doesn't prevent the overflow of counter values in all cases. Thus another mechanism like "wrapping around" is needed to limit the counter size. To offer application specific resolution support the decision maker which chooses the conflict resolution function and the register and deregister functions should be implemented. Since Log Filters do not work correctly the already initiated development of a new mechanism to detect read/write cycles should be continued.

Acknowledgements

First of all, I want to thank my supervisor Sanny Gustavsson for guiding me through all this work, and being there for me, even at the weekends, to discuss problems and new ideas.
I would like to thank Sten F. Andler for all his feedback that helped improving my work and also for making this master thesis possible. In deed the whole Distributed Real-Time research group is really great and I want to thank every member of it, especially Henrik Grimm for his programming help when I saw no way out, Gunnar Mathiason for motivating me so much and Robert Nilsson for reading through my thesis.
I also want to thank Elizabeth Özdemir and Jutta Mülle who made this fantastic stay in Skövde possible and for helping wherever it was possible for them. Furthermore I want to thank my friend Daniel Jagszent for all the important discussions and hints and being everytime available for me. Of course I also want to mention my parents and my brothers and thank them for all their support. Last but not least I want to thank Anka and Sascha for all the nice trips, my neighbour Angeliki for going for a walk with me in times I couldn’t think anymore, Ana, Oliver, Stella and Magda making me laugh all the time, and family Björkenor who invited me for Christmas.

Bibliography

J. B. Almeida, P. S. Almeida, C. Baquero, Departmento de Informática, Universiddade de Minho, Portugal. Bounded Version Vectors, 2004

P. S. Almeida, C. Baquero, V. Fonte, Departamento de Informática, Universidade do Minho Largo do Paco, Portugal.Version Stamps – Decentralized Version Vectors, Proceedings of the 22nd international conference on distributed computing systems (ICDCS), pages 544--551. 2002. IEEE Computer Society.

P. S. Almeida, C. Baquero, and V. Fonte. Panasync: Dependency tracking among file copies. In Paulo Guedes, editor, Ninth ACM SIGOPS European Workshop, pages 7-12. DIKU – University of Copenhagen, 2000.

S. F. Andler, J. Hansson, J. Mellin, J. Eriksson, and B. Eftring. An overview of the DeeDS real-time database architecture. In Proc. 6th Int'lWorkshop on Parallel and Distributed Real-Time Systems, 1998.

J. P. Barreto, Information sharing in mobile networks: a survey on replication Strategies, Instituto Superior Técnico, Inesc-ID Lisboa, September 2003

B. Brachman, G. Neufeld, TDBM: A DBM Library with Atomic Transactions, In Proc. USENIX, San Antonio, 1992

E. Casais, M. Ranft, B. Schiefer, D. Theobald, W. Zimmer, STONE – An Overview, Forschungszentrum Informatik (FZI), Germany, 1992

S. Davidson, H. Gracia-Molina, D. Skeen, Consistency in partitioned networks, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, 1985

R. Elmasri, S. Navathe, Fundamentals of Database Systems (2nd ed.), Redwood City, USA, Benjamin/Cummings Publishing Company, Inc., 1994

D. Eriksson, How to implement Bounded-Delay replication in DeeDS, B.Sc. dissertation, Department of Computer Science, Högskolan i Skövde, 1998.

D. K. Gifford. Weighted voting for replicated data. In Proceedings of the Seventh Symposium on Operating System Principles SOSP 7, pages 150-162, Asilomar Conference Grounds, Pacific Grove CA, 1979. ACM, New York.

S. Gustavsson, S. F. Andler, Continous Consistency Management in Distributed Real-Time Databases with Multiple Writers of Replicated Data, Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'05), Denver, Co, USA, 2005

B. B. Kang, R. Wilensky and J. Kubiatowicz, CS Division, University of California at Berkley.The Hash History Approach for Reconciling Mutual Inconsistency, 2003

L. Lamport, Time, Clocks and the ordering of events in a distributed system Comms. ACM, Vol. 21, No. 7, pp. 558-65, 1978

J. Lundström, A conflict detection and resolution mechanism for bounded-delay replication, M.Sc. dissertation, Department of Computer Science, Högskolan i Skövde, 1997

National Institute of Standards and Technology. FIPS PUB 180-1: Secure Hash Standard. National Institute for Standards and Technology, Gaithersburg, MD, USA, April 1995. Supersedes FIPS PUB 180 1993 May 11.

D. S. Parker, R. A. Ramos, A distributed file system architecture supporting high availability, University of California, Los Angeles, 1982.

T.S. Perry, In Search of the Future of Air Traffic Control, *IEEE Spectrum*, 34(8):18-34, 1997.

K. Petersen, M. Spreitzer, D. Terry and M. Theimer, Bayou: Replicated Database Services for World-wide Applications, In Proceedings of the 7th ACM SIGOPS European Workshop, 1996.

D. Ratner, P. Reiher, G. J. Popek, Department of Computer Science University of California, Los Angeles. Dynamic Version Vector Maintenance, 1997

D. Ratner, P. Reiher, G. J. Popek, and G. H. Kuenning. Replication requirements in mobile environmnets. Mobile Networks and Applications, 6(6):525-533, 2001

P. Reiher, D. Ratner and G. Popek. ROAM: A scalable replication system for mobile and distributed computing, University of California, Los Angeles, 1998.

R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. Distributed Computing, 7(3):149-174, 1994, 1994

M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. IEEE Transactions on Software Engineering, SE-5:188-194, 1979.

D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer and C. H. Hauser. Managing Update Conflicts in Bayou, a weekly connected replicated storage system, SOSP 95

Appendix A - Regression Tests

1) Test 1 - Node creates a replica

Purpose: To test that a dynamic version vector is created correctly and that the data is not corrupted by the conflict detection
Trigger: A node stores a new entry locally in an existing database
Expected outcome: The output is equal to the input and a version vector is created which only contains the writing node

2) Test 2 - Node updates a replica

Purpose: To test that a dynamic version vector is updated correctly and that the data is not corrupted.
Trigger: A node updates an existing value in the database

a) The node writes for the first time to that replica
Expected outcome: A new entry is added in the version vector.
b) The node already has an entry in the version vector
Expected outcome: The existing entry is increased by one.

3) Test 3 – Check the message marshalling

Purpose: To ensure that the update message is created correctly.
Trigger: A transaction is committed in the node
Expected outcome: The propagation message is created correctly with the correct version vectors appended

4) Test 4 – Check the message unmarshalling

Purpose: To ensure that conflict detection data structure is created correctly out of the incoming update message
Trigger: An update message arrives at a node.
Expected outcome: The arriving message is transferred correctly to the conflict detection data structure which is used by the conflict detection mechanism.

5) Test 5 – Logfilter

Purpose: The logfilter should insert the version vectors in the right order.
Trigger: a) The node writes a value into the database
b) An update message arrives at the integrator
Expected outcome: The version vectors are ordered with the help of the domination relation.

6) Test 6 – Write/write conflicts

Purpose: All write/write conflicts should be detected and resolved
Trigger: An update message arrives at the integrator
a) detection

Expected outcome: all write/write conflicts are detected
b) resolution
Expected outcome: all write/write conflicts are resolved.

7) Test 7 – Read/write conflicts

Purpose: All read/write conflicts should be detected
Trigger: An update message arrives at the integrator
Expected outcome: all read/write conflicts are detected

8) Test 8 – Read/write cycles

Purpose: All read/write cycles should be detected
Trigger: An update message arrives at the integrator
Expected outcome: all read/write cycles are detected

Appendix B – Conflict Detection & Resolution Interface

Conflict Detector:

There are two functions that have to be implemented. The function detectConflicts is called to check a conflict detection data structure sequence. It takes a conflict detection data structure sequence as input parameter and returns a conflict report (see drd_CDDS.h in Appendix C for data type details).

drd_ConflictReport* detectConflicts(drd_CDDSseq *sequence)

The second function is used to integrate a local transaction in the conflict detector.
A boolean is returned if the integration succeeded. The input parameter is the sequence that has to be integrated.

doi_Boolean integrateLocalUpdate(drd_CDDSseq *sequence)

Conflict Resolver:

The conflict resolver has to implement a function with the following parameters and register it at the Conflict Handler. At the moment there is only one interface for write/write conflict resolvers. But the interface could be extended to also include read/write conflict detection.
For more details about the data types, see drd_CDDS.h in Appendix C.

Return value: doi_Boolean
This value indicates if the conflict could be resolved.

Function Parameters:

drd_CDDS *cddsVector	Pointer to the conflict detection data structure of the conflicting replicas (input)
Datum *values	Pointers to the values of the conflicting replicas (input)
int numberOfValues	number of replicas that are in conflict (input)
drd_CDDS *mergedCDDS	includes the merged conflict detection data structure after resolution (output)
Datum *mergedValue	includes the merged value after the resolution (output)

Appendix C – Source Code

Updater (update.cc)

```
/*---------------------------------------------------------------------------
 FUNCTION

   drd_updateDatabase

 DESCRIPTION

           This function is called from the integrator and guides the
           integration process. It delegates the incoming Stream to the
           Conflict Handler to detect and resolve the conflicts and after that
           calls the local "integrateUpdates" function to store the written values in the database.

 SYNOPSIS
 INPUT
           drd_replicationType repType     the replication type which is used for the incoming transaction
(update message)
           char *transactionStream                     the content of the incoming update message
(containing all information about the
                                                                  transaction)
 OUTPUT
 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS

---------------------------------------------------------------------------*/

void drd_updateDatabase(drd_replicationType repType, char *transactionStream)
{
           Tid *tid;
           Dbm *dbm;
           string pathname;
           drd_LogHeader *firstHeader = new drd_LogHeader;
           Datum *writtenValuesOut;
           int numberOfUpdates;
           DbmRc drc;


           //Get the pathname out of the stream
           //for this we parse the first header and save it in the variable firstHeader
           memcpy (firstHeader, &transactionStream[sizeof(long)], sizeof(drd_LogHeader));
           pathname.assign(firstHeader->pathname);

/* open database to set locks */
           dtd_CHECK(DbmOpen(pathname.c_str(), DBM_VOLATILE, NULL, &dbm, NULL));
           //replicationType = drd_Replication_asap;
           }
	// begin integration transaction
	dtd_CHECK( dtd_begin_ext(NULL, &tid, drd_LocalTrans));
	tid->drd_logInfo.transType = drd_LocalTrans;

	// perform the conflict detection and resolution
	if (!processConflicts(repType, transactionStream, &writtenValuesOut, &numberOfUpdates,
dbm, tid))
		std::cout << "Error conflicts could not be processed !" << std::endl;

	cout << "Number of updates: " << numberOfUpdates << endl;
	// integrate the writtenValuesOut to the database
	drc = integrateUpdates(pathname, dbm, tid, writtenValuesOut, numberOfUpdates);

}

/*-----------------------------------------------------------------------
 FUNCTION

   integrateUpdates

 DESCRIPTION

	This function stores the values of the writtenValuesOut vector to the database.
	It finishes the integration of an incoming update message.

 SYNOPSIS
 INPUT
	const string& pathname		the pathname of the TDBM file
	Dbm *dbm					the TDBM file handle
	Tid *tid				the id of the transaction
	Datum *writtenValues		the writtenValues vector containing the datums which have
							to be insert into the database (first
datum is the 1.key, second datum is the 1.value
							third datum is the 2.key....)
	int objnumber				the number of objects in the writtenValues vector
 OUTPUT
	DbmRc					return value (at the moment is not used)
 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS

-----------------------------------------------------------------------*/

DbmRc integrateUpdates(const string& pathname, Dbm *dbm, Tid *tid, Datum *writtenValues, int
objnumber)
{
  int i,j = 0;
  DbmRc rc;

  /*stores the values in the database */
  printf("STORES VALUE IN DATABASE \n");

  j=0;
  for(i = 0; i < objnumber; i++) {
    dtd_CHECK( DbmStore(dbm, tid, writtenValues[j], writtenValues[j+1], DBM_REPLACE) );
    j=j+2;
  }

  //printf("SETTING READONLYTRANS \n");
  tid->drd_logInfo.readOnlyTrans = doi_false;

  //printf("COMMIT TRANSACTION \n");
  if ((rc = DbmCommit(tid)) != DBM_OK) {
    return rc;
  }

  //printf("CLOSING DATABASE \n");
  if ((rc = DbmClose(dbm)) != DBM_OK)
  {
    return rc;
  }

  drd_signalIntegrationEvent(pathname, objnumber, writtenValues);

  //free memory
  destroy_Datum(writtenValues);

  cout << "stop" << endl;
  return rc;
}
```

Conflict Handler (drd_ConflictHandler.cc)

```
/*---------------------------------------------------------------------------
 FUNCTION

   registerResolver

 DESCRIPTION

   This function registers a resolver for a special conflict

 SYNOPSIS
 INPUT
 OUTPUT
 RETURNS

    boolean
      showing if the registration was successful

 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
---------------------------------------------------------------------------*/

doi_Boolean registerResolver()
{
	return doi_true;
}


/*---------------------------------------------------------------------------
 FUNCTION

   deregisterResolver

 DESCRIPTION

   This function deregister a resolver for a special conflict conflict

 SYNOPSIS
 INPUT
 OUTPUT
 RETURNS

    boolean
      showing if the deregistration was successful

 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
---------------------------------------------------------------------------*/

doi_Boolean deregisterResolver()
{
        return doi_true;
}


/*------------------------------------------------------------------------
 FUNCTION

 processConflicts

 DESCRIPTION

   Checks a new Update from outside for conflicts and resolves these

 SYNOPSIS
 INPUT
        drd_replicationType repType:    This is the replication type used for this transaction.
                                        At the moment there is only one replication Type ASAP
        char *transactionStream:        This Stream is the content of the update message and contains
                                        all the information for the read and write set of the transaction
                                        which shall be integrated.

 OUTPUT
        Datum **writtenValuesOut:       gives back a vector including all the written values which
                                        have to be updated in the local database.
        int *numberOfUpdates:           gives back the number of updates which are stored in the
                                        writtenValuesOut vector
        Dbm *dbm:                       gives back the dbm handle which shows which database file
                                        has been accessed
        Tid *tid:                       gives back the transaction ID
 RETURNS

Boolean                                 showing if all of the conflicts could be removed

 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
------------------------------------------------------------------------*/


doi_Boolean processConflicts(drd_replicationType repType, char *transactionStream, Datum
**writtenValuesOut, int *numberOfUpdates, Dbm *dbm, Tid *tid)
{
        drd_LogFilterHandler* lfh;
        drd_CDDSseq* seq;
        drd_ConflictReport* cr;

        switch(repType)
        {
        case ASAP://give back the logfilter for handling conflicts of this replication type
                                lfh = drd_LogFilterHandler::giveBackLogFilterHandler(ASAP);
                                break;
        case BOUNDED://give back the logfilter for handling conflicts of this replication type
                               lfh = drd_LogFilterHandler::giveBackLogFilterHandler(BOUNDED);
                                break;
        default : return doi_false;
                        cout << "Error in informAboutLocalTrans - unknown replication type ! " ;
        }
        //parse the transactionStream to get the CDDS sequence
        seq = drd_CDDSHandler::createCDDSSeq(transactionStream, writtenValuesOut,
        numberOfUpdates);
        //check for conflicts
        //lock database
        //lock the logfilter

        cr = lfh->detectConflicts(seq);

        //resolve the different conflicts

        if (cr->WWConflict)
        {//resolve WW conflict
                cout << "WW conflict detected" << endl;
                //a write/write conflict is detected

                if (!handleWWConflict( cr, seq, writtenValuesOut, *numberOfUpdates, dbm, tid))
                {
                        cout << "Error during conflict resolution (WW error)" << endl;
                        return doi_false;
                }
                //conflict detection again...
        }
        if (cr->RWConflict)
        { //resolve read/write conflict
                cout << "Read/Write Conflict detected" << endl;
        }
        if (cr->RWCycle)
        { //resolve read/write cycle
                cout << "Read/Write cycle detected" <<endl;
        }

        // free memory
        destroy_drd_ConflictReport(cr);
        return doi_true;  //written values....
}


/*---------------------------------------------------------------------------
 FUNCTION

  informAboutLocalTrans

 DESCRIPTION

  This function informs the conflict handler about a new local transaction
  so that the conflict detection data can be updated

 SYNOPSIS
 INPUT
        drd_replicationType repType             indicates the replication type of the transaction
        char *transactionStream             the same ByteStream which is sent over the network
                                            to inform the other nodes about a new transaction.
                                            Here it is used to update the Conflict Detection
                                            component
 OUTPUT
 RETURNS

   boolean
     indicating if the conflict detection datastructure could be updated

 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
---------------------------------------------------------------------------*/

doi_Boolean informAboutLocalTrans(drd_replicationType repType, char *transactionStream)
{
        drd_LogFilterHandler* lfh;
        drd_CDDSseq* seq;
        Datum *writtenValuesOut;
        int numberOfUpdates;
        doi_Boolean status = doi_true;

        //get the LogFilterHandler depending on the replication type
        switch(repType)
        {
        case ASAP: lfh = drd_LogFilterHandler::giveBackLogFilterHandler(ASAP);
                                break;
        case BOUNDED: lfh = drd_LogFilterHandler::giveBackLogFilterHandler(BOUNDED);
                                break;
        default : status =doi_false;
                                cout << "Error in informAboutLocalTrans - unknown replication
type !" ; //Error
        }
        //Create a CDDS sequence
        seq = drd_CDDSHandler::createCDDSSeq(transactionStream, &writtenValuesOut,
        &numberOfUpdates);
        //Call the function in the Logfilter Handler to update it
        lfh->integrateLocalUpdate(seq);
        return status;
}


/*---------------------------------------------------------------------------
 FUNCTION

  handleWWConflict

 DESCRIPTION


 SYNOPSIS
 INPUT
        drd_ConflictReport *rc          the error report which contains the conflicting CDDS objects
        drd_CDDSseq *sequence       the CDDS sequence which caused the conflict (which should
                                    be integrated to the logfilter)
        Datum **writtenValues       the vector of the written values. Needed to change the values
                                    which are integrated to the database (all entries in this vector
                                    are later simply stored in the database by the Updater)
        int numberOfUpdates         the number of updates in the writtenValues vector
        Dbm *dbm                    the handle for the TDBM file
        Tid *tid                    the id of the transaction
 OUTPUT
        Datum **writtenValues       contains the updated values after the procedure finished
 RETURNS
        doi_Boolean                                 indicating that all conflicts could be resolved
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
---------------------------------------------------------------------------*/
doi_Boolean handleWWConflict( drd_ConflictReport *rc, drd_CDDSseq *sequence, Datum
**writtenValues, int numberOfUpdates, Dbm *dbm, Tid *tid)
{
        //resolveWWConflict( drd_CDDS *cddsVector, Datum *values, int  numberOfValues,
drd_CDDS *mergedCDDS, Datum *mergedValue)
        int searchedIndex = -1;
        Datum databaseValue;

        drd_CDDS cdds1;
        drd_CDDS cdds2;
        Datum* v1;
        Datum* v2;
        long value1;
        long value2;
        int lengthValue;
        drd_CDDS *tempCddsVector;
        Datum *tempValueVector;
        drd_CDDS *mergedCDDS = new drd_CDDS;
        Datum *mergedValue = new Datum;
        //should be initialized every time that all pointers are set to 0 -- unused pointers are then not be
destroyed
        init_drd_CDDS(mergedCDDS);

        // search the conflicting datum in the writtenValues vector
        for (int i=0; i < numberOfUpdates; i++)
        {
                if ( areKeysEqual( (*writtenValues)[2*i].dptr, rc->conflictingCDDS->key) )
                        searchedIndex = i;
        }
        //fetch the old value out of the database
        dtd_CHECK(DbmFetch(dbm, tid, (*writtenValues)[2*searchedIndex], &databaseValue));

   //split the Datum in CDDS and Value
   drd_CDDSHandler::splitDatum(&((*writtenValues)[2*searchedIndex+1]), &cdds1, &v1,
&lengthValue);
   drd_CDDSHandler::splitDatum(&databaseValue, &cdds2, &v2, &lengthValue );
   value1 = *(long*)v1->dptr;
   value2 = *(long*)v2->dptr;
```

```
    //build the vectors which are needed for the resolver
    // !!! The key is not set in the CDDS !!! we do not need this at the moment for the conflict resolution
    tempCddsVector = (drd_CDDS*) doi_allocMem(2*sizeof(drd_CDDS));
    memcpy( tempCddsVector, &cdds1, sizeof(drd_CDDS) );
    memcpy(  &tempCddsVector[1], &cdds2, sizeof(drd_CDDS) );


    tempValueVector = (Datum*) doi_allocMem(2*sizeof(Datum));
    memcpy( tempValueVector, v1, sizeof(Datum) );
    memcpy( &tempValueVector[1], v2, sizeof(Datum) );

    //free memory
    delete v1;
    delete v2;



    //call the resolving function in the resolver
    if (!resolveWWConflict( tempCddsVector, tempValueVector, 2, mergedCDDS, mergedValue))
            return doi_false;

    //update of the writtenValues vector
    memcpy((*writtenValues)[2*searchedIndex+1].dptr, mergedCDDS->cddata, mergedCDDS-
>cddataSize);
    memcpy(&(*writtenValues)[2*searchedIndex+1].dptr[mergedCDDS->cddataSize], mergedValue-
>dptr, sizeof(long));
    (*writtenValues)[2*searchedIndex+1].dsize = mergedCDDS->cddataSize + sizeof(long);

    //update of the current CDDS sequence
    memcpy(rc->currentCDDS->cddata, mergedCDDS->cddata, mergedCDDS->cddataSize);
    rc->currentCDDS->cddataSize = mergedCDDS->cddataSize;

    //free the memory
    if (mergedCDDS != 0)
            delete mergedCDDS;
            if (mergedCDDS != 0)
                    delete mergedValue;


            return doi_true;
}

/*-----------------------------------------------------------------------
 FUNCTION

   areKeysEqual

 DESCRIPTION


 SYNOPSIS
 INPUT
            char *key1                  the first char*
```

```
        char *key2              the second char*
 OUTPUT
 RETURNS
        doi_Boolean     indicates if the chars are equal (true) or not (false)
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
---------------------------------------------------------------------------*/
//checks if two chars are the same
doi_Boolean areKeysEqual (char *key1, char *key2)
{
        if (strcmp(key1, key2) == 0)
                return doi_true;
        else
                return doi_false;
}
```

Conflict Handler (drd_ConflictHandler.cc)

```
/*---------------------------------------------------------------------------
 FUNCTION

   compare

 DESCRIPTION

  This function compares two CDDS. (e.g. two dynamic Version Vectors).
  This is needed during the integration process of an remote Update.

 SYNOPSIS
 INPUT
        char *cdds1_cddata
                This is a pointer to the first CDDS. The CDDS itself are
                ByteStreams which contain at first the number of entries
                and then the entry value pairs in a total order (from small to
                big node identifiers).

        char *cdds2_cddata
                see cdds1_cddata

 OUTPUT
 RETURNS
        drd_Rc
                This is the internal status report for the conflict detection
                which indicates in which relationship the two CDDS are.
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS

---------------------------------------------------------------------------*/
drd_Rc drd_CDDSHandler::compare(char *cdds1_cddata, char *cdds2_cddata)
{
        long numberOfEntriesCDDS = 0;
        drd_searchResult* sr;
        drd_Rc status;
        long nodeNumber;
        int pos =0;
        long compareValue1 = 0;
        long compareValue2 = 0;
        doi_Boolean CDDS1dominates = doi_false;
        doi_Boolean CDDS2dominates = doi_false;


        // reading the number of entries in the CDDS
        memcpy(&numberOfEntriesCDDS, cdds1_cddata, sizeof(long));
        // move the position pointer in the ByteStream further
        pos += sizeof(long);

        // going through the first CDDS and compare all entries
        // with the ones from the second one
        for ( int i=0; i<numberOfEntriesCDDS; i++)
        {
                memcpy(&nodeNumber, &cdds1_cddata[pos], sizeof(long));
                pos += sizeof(long);
                memcpy(&compareValue1, &cdds1_cddata[pos], sizeof(long));
                pos += sizeof(long);
                // this function is used to find the nodeidentifier
                sr = searchDVV(cdds2_cddata, nodeNumber);
                //if the entry is available in the 2. CDDS then compare it
                if(!(sr->newEntry))
                {
                        memcpy(&compareValue2, &cdds2_cddata[sr->index*2*sizeof(long)], sizeof(long));
//(sr->index-1)*2*sizeof(long)+sizeof(long) !counter! +sizeof(long)!nodeidentifier!
                        if (compareValue1 < compareValue2)
                                CDDS2dominates = doi_true;
                        if (compareValue1 > compareValue2)
                                CDDS1dominates = doi_true;
                }
                else CDDS1dominates = doi_true;  //if the node is not available in the second CDDS then the
first one dominates this
        }

        // going through the second CDDS and compare all entries with the ones from the first one
        // this is necessary because there is the possibility that in the second CDDS are entries
        // which are not in the first and would not be compared at all-- a more efficient way would be
        // to calculate the intersection of both CDDS and then compare these (here some are compared doubble
        pos = 0;
        memcpy(&numberOfEntriesCDDS, cdds2_cddata, sizeof(long));
        pos += sizeof(long);
        for ( int i=0; i<numberOfEntriesCDDS; i++)
        {
                memcpy(&nodeNumber, &cdds2_cddata[pos], sizeof(long));
                pos += sizeof(long);
                memcpy(&compareValue2, &cdds2_cddata[pos], sizeof(long));
                pos += sizeof(long);
                // this function can be used to find the nodeidentifier
                sr = searchDVV(cdds1_cddata, nodeNumber);
                //if the entry is available in the 2. CDDS then compare it
                if(!(sr->newEntry))
                {
                        memcpy(&compareValue1, &cdds1_cddata[sr->index*2*sizeof(long)], sizeof(long));
//(sr->index-1)*2*sizeof(long)+sizeof(long) !counter! +sizeof(long)!nodeidentifier!
                        if (compareValue1 < compareValue2)
                                CDDS2dominates = doi_true;
                        if (compareValue1 > compareValue2)
                                CDDS1dominates = doi_true;
                }
                else CDDS2dominates = doi_true;  //if the node is not available in the first CDDS then the
second one dominates this
        }

        // If all entries are compared we have to find out the relation the
        // two CDDS have to each other. For better understanding one example is
        // given here: If CDDS1 dominates at least one entry and CDDS2 dominates
        // none than the whole CDDS1 dominates CDDS2.
        // If both CDDS1 and CDDS2 dominate at least one entry than there is a
        // write/write conflict.
        if (CDDS1dominates && !CDDS2dominates)
                status = DRD_DOMINATES;
        if (!CDDS1dominates && CDDS2dominates)
                status = DRD_DOMINATED;
        if (CDDS1dominates && CDDS2dominates)
                status = DRD_WW_CONFLICT;
        if (!CDDS1dominates && !CDDS2dominates) //can only happen if they are completely equal
                status = DRD_INDEPENDENT;
        return status;
}


/*------------------------------------------------------------------------
 FUNCTION

   compare

 DESCRIPTION

 This function compares two CDDS sequences. This is needed during the integration process of
 a remote update.

 SYNOPSIS
 INPUT

 cddsSeq1     the first CDDS sequence
 cddsSeq2     the second CDDS sequence

 OUTPUT
 RETURNS
        drd_Rc  This is a status report which is given back to indicate
                        which realtions exists between the two sequences.
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS

------------------------------------------------------------------------*/
drd_Rc drd_CDDSHandler::compare(drd_CDDSseq *cddsSeq1, drd_CDDSseq *cddsSeq2, drd_CDDS
*conflictingCDDS)
{
        drd_CDDS *cddsSeq1_current = cddsSeq1->head;
        drd_CDDS *cddsSeq2_current = cddsSeq2->head;
        doi_Boolean cdds1Dominates = doi_false;
        doi_Boolean cdds2Dominates = doi_false;
        drd_Rc status;

        //if there is the possibility that one sequence can span over multiple
        //TDBM files than there should be an additional check.

        //a sequence must contain at least one element
        while (cddsSeq1_current != 0)
        {   // the sequence has to be ordered from small keys to huge keys...
                // if the key from seq1 is greater then we have to move to the next element in seq2
                while (cddsSeq2_current != 0 && *(long*)(cddsSeq1_current->key) >=
*(long*)(cddsSeq2_current->key) )
                {
                        if (keyCompare( cddsSeq1_current->key, cddsSeq2_current->key, cddsSeq1_current-
>keySize, cddsSeq2_current->keySize))
                        {
                                //if the keys are equal which means that the same object was accessed by
                                //both transactions (CDDS sequences) they have to be compared.
                                status = compare (cddsSeq1_current->cddata, cddsSeq2_current->cddata);
                                switch(status)
                                {
                                        case DRD_WW_CONFLICT:       memcpy(conflictingCDDS,
cddsSeq2_current, sizeof(drd_CDDS));
```

```
                                        memcpy(&conflictingCDDS[sizeof(drd_CDDS)], cddsSeq1_current,
sizeof(drd_CDDS));

                                        return status;

                                        break;
                                                  case DRD_DOMINATED:                    cdds2Dominates =
doi_true; //that means that cdds2 dominates cdds1

                                        break;
                                                  case DRD_DOMINATES:                    cdds1Dominates =
doi_true;

                                        break;
                                                  case DRD_INDEPENDENT:        break;
                                                  default:
                    cout << "Error -- unknown error type during compare !" << endl;
                                        }
                              }
                              // go to the next cdds in the sequence2
                              cddsSeq2_current = cddsSeq2_current->next;
                    }
                    // go to the next CDDS in sequence1
                    if (cddsSeq1_current->next != 0 )
                    {
                              cddsSeq1_current = cddsSeq1_current->next;
                    }
                    else
                              break;
                    // set back the current CDDS of sequence 2 ( to the head)
                    cddsSeq2_current = cddsSeq2->head;
          }

          // After all entries are compared the relation report has to be
          // created.
          if (cdds1Dominates && cdds2Dominates)
                    status = DRD_RW_CONFLICT;
          if (cdds1Dominates && !cdds2Dominates)
                    status = DRD_DOMINATES;
          if (!cdds1Dominates && cdds2Dominates)
                    status = DRD_DOMINATED;
          if (!cdds1Dominates && !cdds2Dominates)
                    status = DRD_INDEPENDENT;
          return status;
}


/*---------------------------------------------------------------------------
 FUNCTION

   createCDDSSeq

 DESCRIPTION

 This function parses a transactionStream and gives back
 a vector of writtenValues and a CDDS-Sequence.

 SYNOPSIS
 INPUT
 ransactionStream:   the stream which shall be parsed

 OUTPUT

 writtenValuesOut    :   a vector of written values which is needed by the update.cc
 numberOfUpdates  :  the number of values in the writtenValuesOut Vector
                                                                     One value needs two entries in the Vector
(key, value).
                                                                     So the size of the writtenValueOut is
2*numberOfUpdates*SavedStructure

 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS

The function is used both in the localUpdate process and the remoteUpdate process.
In the localUpdate process it is used to parse the outgoing transactionByteStream to
get a CDDS sequence which can be appended at the end of the LogFilter or another
conflict detection module.
In the remoteUpdate process it is used to parse the incoming transactionStream to
get the CDDS sequence which is used to detect conflicts and to get an array of writtenValues
which the update.cc needs to integrate the written values to the database.

---------------------------------------------------------------------------*/
drd_CDDSseq* drd_CDDSHandler::createCDDSSeq(char *transactionStream,Datum **writtenValuesOut, int
*numberOfUpdates)
{
        long position = 0;
        long noObjs;
        long arrPos = 0;
        char pathname[MAX_PATH_LENGTH];
        Datum* writtenValues;
        drd_CDDSseq* newSeq;
        drd_CDDS* tempCDDS = new drd_CDDS;
        init_drd_CDDS(tempCDDS);


        //At the head of the transactionStream there is a long which contains the number of objects
        //in the stream
        memcpy(&noObjs, transactionStream, sizeof(long));
        //noObjs = ntohl(noObjs);
        position = sizeof(long);

         /* Allocate an array the size of the number of objects. The key and
   * the value will then be put in pairs in this vector. Later used
   * when performing the update in the database by steping through the
   * array and put the values in the database. */
        writtenValues = (Datum *) doi_allocMem(2 * noObjs * sizeof(Datum));
        for(int i = 0; i < 2 * noObjs; i++) {
    writtenValues[i].dptr = 0;
    }

   /* Create a sequence to which dynamic version vectors can be added. */
   newSeq=(drd_CDDSseq *)doi_allocMem(sizeof(drd_CDDSseq));
   newSeq->head=0;
   newSeq->next=0;
   //initialize the pathname vector (necessary because of the compare pathname
   //function which compares all 128 letters
```

```
        for ( int m=0; m< MAX_PATH_LENGTH; m++)
                newSeq->pathname[m] = ' ';

/* Get info from log */
for (int k = 0; k < noObjs; k++) {
  drd_LogHeader header;
  memcpy(&header, transactionStream + position, sizeof(drd_LogHeader));

  memcpy(&pathname, header.pathname, sizeof(char[MAX_PATH_LENGTH]));
  position += sizeof(drd_LogHeader);

  if (header.opType == drd_Read) {

    //if the access type is drd_Read there is only the key and the
    //CDDS in the ByteStream because the value is not changed.

    //build the new dynamic version vector of this entry
    //copy all values out of the transactionStream
    memcpy(&newSeq->pathname, &pathname, sizeof(char[MAX_PATH_LENGTH]));
    tempCDDS->keySize = header.keySize;
    tempCDDS->key = (char*) doi_allocMem(tempCDDS->keySize);
    memcpy(tempCDDS->key, &transactionStream[position], tempCDDS->keySize);
    // jump over the key in the stream to the beginning of the CDDS
    position += header.keySize;

    tempCDDS->cddata = (char*) doi_allocMem(header.cddataSize);
    memcpy(tempCDDS->cddata, &transactionStream[position], header.cddataSize);
    /* Copy the dynamic version vector tempCDDS to the version vector sequence. */

    //drd_ntohVV(tempVV);
    //add DVV to the Sequence
    addCDDStoSeq(tempCDDS, newSeq);
    position += header.cddataSize;
  }
  else if (header.opType == drd_Write) {

    //if the access type is drd_Write there is the key, the
    //CDDS and the value in the ByteStream. So these variables have
    //to be parsed out of the Stream. In the written case the
    //entry also has to be added to the writtenValues Vector
    //which includes all the written values which have to be later
    //integrated to the local database.

    /* Copy the key to an array position */
    writtenValues[arrPos].dsize = header.keySize;
    writtenValues[arrPos].desc = header.keyDesc;
    writtenValues[arrPos].dptr = (char*) doi_allocMem(header.keySize);
    memcpy(writtenValues[arrPos].dptr, &transactionStream[position], header.keySize);
    arrPos++;

    //build the new dynamic version vector of this entry
    //copy all values out of the transactionStream
    memcpy(&newSeq->pathname, &pathname, sizeof(char[MAX_PATH_LENGTH]));
    tempCDDS->keySize = header.keySize;
    tempCDDS->key = (char*) doi_allocMem(tempCDDS->keySize);
    memcpy(tempCDDS->key, &transactionStream[position], tempCDDS->keySize);
    // jump over the key in the stream to the beginning of the CDDS
    position += header.keySize;

    tempCDDS->cddataSize = header.cddataSize;
```

```
      tempCDDS->cddata = (char*) doi_allocMem(tempCDDS->cddataSize);
      memcpy(tempCDDS->cddata, &transactionStream[position], tempCDDS->cddataSize);
      /* Copy the dynamic version vector tempCDDS to the version vector sequence. */
      addCDDStoSeq(tempCDDS, newSeq);; //add DVV to sequence

      /* Copy the value to an array position */
      writtenValues[arrPos].dsize = header.valueSize;
      writtenValues[arrPos].desc = header.valueDesc;
      writtenValues[arrPos].dptr = (char*) doi_allocMem(header.valueSize);
      memcpy(writtenValues[arrPos].dptr, &transactionStream[position], header.valueSize);
      position += header.valueSize;
      arrPos++;
    }
    else {
      fprintf(stderr, "ERROR - drd_createVVSequence: Unknown opType\n");
      return NULL;
    }
  }
  *writtenValuesOut = writtenValues;
  *numberOfUpdates = arrPos / 2;

  //free memory
  // this tempCDDS points to the last element in the vvSeq .. so we are not allowed to delete it...!!!!
  //destroy_drd_CDDS(tempCDDS);

  cout << "returning sequence" << endl;
  return newSeq;

}


/*-------------------------------------------------------------------------
 FUNCTION

   updateCDDS

 DESCRIPTION

 This function updates the CDDS structure of a value in the database.
 If it is a new value which has no datastructure then a cdds is created and
 saved at the beginning of the value.

 SYNOPSIS
 INPUT

 currentValue:    the new value which the application wants to store into the database
 oldEntry:                              the old entry of the database which contains the old CDDS
                                                          if there is no oldEntry for this currentValue then the
oldEntry->dptr pointer is
                                                          set to 0.
 NodeNumber:   this is the nodeidentifier which is needed to update some CDDS (e.g.: version vectors)

 OUTPUT

 cddsSize                        the size of the cdds data structure in the value

 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
 currentValue can be used to calculate the new hashValue if HashHistories are used as CDDS
---------------------------------------------------------------------------*/

Datum* drd_CDDSHandler::updateCDDS(Datum *currentValue, Datum *oldEntry, long NodeNumber, Tid
*tid, Dbm *dbm, Datum *key, long *cddsSize)
{
        Datum *newEntry = new Datum;
        const long newDVVElementCounter = 1;
        const long newDVVCounter = 1;
   drd_searchResult* sr;
   long numberOfEntries = 0;
   long counterToIncrease = 0;
   long pos = 0;

  /* Optimization: Perform a lookup to see if this value is already updated in
   * this transaction. If it is, the version vector does not need to
   * be increased.  This could be done by checking if the value is
   * stored in the log as a write operation. */

        newEntry->desc = currentValue->desc;

        /* Check if it is a new entry in the database for which we have to create a new dynamic version vector
*/
   if(oldEntry->dptr == 0) {
        /* Create dynamic version vector because it is a new entry*/
        newEntry->dsize = 3*sizeof(long) + currentValue->dsize;
        newEntry->dptr  = (char*) doi_allocMem(newEntry->dsize);
      memcpy(newEntry->dptr, &newDVVElementCounter, sizeof(long));
      memcpy(&newEntry->dptr[sizeof(long)], &NodeNumber, sizeof(long));
      memcpy(&newEntry->dptr[2*sizeof(long)], &newDVVCounter, sizeof(long));
      memcpy(&newEntry->dptr[3*sizeof(long)], currentValue->dptr, currentValue->dsize);
      numberOfEntries++; //this is important to calculate the cddsSize at the end
   }else
   {  // the updated value was already in the database...so the dynamic VV already exists
     // first search through the dynamic VV of the old entry to see if this node already wrote to that replica
     // (the list is ordered by nodeidentifiers)
     sr = searchDVV(oldEntry->dptr, NodeNumber);
     if(sr->newEntry)
     {     //at the moment the node has no entry in the dynamic Version Vector (DVV)
                //so we have to insert the nodenumber and a 1 (as countervalue)  at the pointer

                //we now know how long the new dvv will be
                //copy the old number of entries
                memcpy(&numberOfEntries, oldEntry->dptr, sizeof(long));
                //increase the number of DVV entries
                numberOfEntries++;
                //2*noe for the entries +  sizeof(long) for the entrycounter at the beginning
                newEntry->dsize = 2*numberOfEntries*sizeof(long) + sizeof(long) + currentValue->dsize;
                newEntry->dptr = (char*) doi_allocMem(newEntry->dsize);

                memcpy(newEntry->dptr, &numberOfEntries, sizeof(long));
                pos = sizeof(long);
                if (sr->index > 1)
                {
                        memcpy(&newEntry->dptr[pos], &oldEntry->dptr[sizeof(long)], 2*(sr->index -
1)*sizeof(long)); //2* because there are two longs per entry
                        pos += 2*(sr->index - 1) * sizeof(long);
                }
                memcpy(&newEntry->dptr[pos], &NodeNumber, sizeof(long));
```

```
                        pos += sizeof(long);
                        memcpy(&newEntry->dptr[pos], &newDVVCounter, sizeof(long));
                        pos += sizeof(long);
                        //copy the rest of the old dvv -- the length is: 2*(numberOfEntries - 1 - (sr->index - 1)
=numberOfEntries - sr->index)
                        //the 2* is because every entry needs two longs nodeid + counter
                if (numberOfEntries - sr->index > 0)
                {   //the +sizeof(long) is because of the elementcounter of the old dvv
                        memcpy(&newEntry->dptr[pos], &oldEntry->dptr[(sr->index - 1)*sizeof(long)+sizeof(long)],
2*(numberOfEntries - sr->index)*sizeof(long));
                        pos += 2*(numberOfEntries - sr->index)*sizeof(long);
                }
                //copy the value
                memcpy(&newEntry->dptr[pos], currentValue->dptr, currentValue->dsize);
        }else
        {     //we just have to increase an entry
                        //we increase the entry in the old DVV and then just copy it to the new entry
                        //calculate the position of the counter in the old dvv
                        pos = sr->index * 2*sizeof(long);  // sizeof(long) for the elementCounter + index-1 *
2*sizeof(long) for the entry pair + sizeof(long) for the key
                        memcpy(&counterToIncrease, &oldEntry->dptr[pos], sizeof(long));
                        counterToIncrease++;
                        memcpy(&oldEntry->dptr[pos], &counterToIncrease, sizeof(long));
                        //we now know how long the new dvv will be
                        //copy the old number of entries
                        memcpy(&numberOfEntries, oldEntry->dptr, sizeof(long));
                        newEntry->dsize = (2* numberOfEntries + 1)*sizeof(long) + currentValue->dsize;
                        newEntry->dptr = (char*) doi_allocMem(newEntry->dsize);

                        //copy dvv and the value to the newEntry
                        //copy the dvv
                        memcpy(newEntry->dptr, oldEntry->dptr, (2* numberOfEntries + 1)*sizeof(long));
                        pos = (2* numberOfEntries + 1)*sizeof(long);
                        //copy the value
                        memcpy(&newEntry->dptr[pos], currentValue->dptr, currentValue->dsize);
        }
    }
    //calculate the size of the CDDS
    *cddsSize = 2* sizeof(long) * numberOfEntries + sizeof(long); //... + element counter

    return newEntry;
}

/*---------------------------------------------------------------------------
 FUNCTION

   removeCDDS

 DESCRIPTION

  This function is called from the TDBM interface. It is used to remove
  the conflict detection data structure from a value fetched from the database so that the
  conflict detection data structure is hidden from the caller of DbmFetch.

 SYNOPSIS
 INPUT
 OUTPUT
 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
---------------------------------------------------------------------------*/
void drd_CDDSHandler::removeCDDS(Datum *value, long *cddsSize)
{
  long numberOfVectorEntries;
  // read the number of entries in the dynamic version vector (it is at the beginnig of the value)
  memcpy(&numberOfVectorEntries, value->dptr, sizeof(long));
  // every entry in the dynamic version vector bytestream consists of
  // two longs. One long is for the node-identifier and one is for the counter.
  // So the whole dynamic version vector structure is 2*number of entries *sizeof(long) + 1 sizeof(long)
  // for the entry counter at the beginning of the dynamic version vector
  value->dsize -= (2*numberOfVectorEntries+1)*sizeof(long);
  value->dptr += (2*numberOfVectorEntries+1)*sizeof(long);
  *cddsSize = (2*numberOfVectorEntries+1)*sizeof(long);
}

/*---------------------------------------------------------------------------
 FUNCTION

   mergeCDDS

 DESCRIPTION
	This procedure merges two different Version Vectors together. This is
	necessary during the conflict resolution. The resolved value needs
	a new Version Vector which dominates the two original ones.
 SYNOPSIS
 INPUT
	char *cdds1		Version Vector 1 which has to be merged
	char *cdds2					Version Vector 2
 OUTPUT
	char **resultCDDS		This is the merged Confict Detection Data Structure
	int *resultCddsSize This is the new size of the merged Conflict Detection Data Structure
						(in the Dynamic Version Vector approach the size is not
constant)
 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS

---------------------------------------------------------------------------*/
void drd_CDDSHandler::mergeCDDS(char *cdds1, char *cdds2, char **resultCDDS, int *resultCddsSize)
{
	long numberOfEntriesCDDS1=0;
	long numberOfEntriesCDDS2=0;
	long nodeid1;
	long nodeid2;
	long counter1;
	long counter2;
	long pos1;
	long pos2;
	long value1;
	long value2;
	long resultvalue;
	int resultpos;
	long numberOfResultElements = 0;
	doi_Boolean run = doi_true;


	memcpy(&numberOfEntriesCDDS1, cdds1, sizeof(long));
	memcpy(&numberOfEntriesCDDS2, cdds2, sizeof(long));
	//allocate so much memory that all entries of both cdds fit into the resulting vector
	//it is possible that we do not need all the space because some entries might be in both the same (or only
with different value)
	*resultCDDS = (char*) doi_allocMem(
(numberOfEntriesCDDS1+numberOfEntriesCDDS2)*2*sizeof(long));
	resultpos = sizeof(long); //keep the first long empty for the elementcounter
	pos1 = sizeof(long);
	pos2 = sizeof(long);
	counter1 = 0;
	counter2 = 0;
	while(run)
	{
		if ( counter1 < numberOfEntriesCDDS1 && counter2 < numberOfEntriesCDDS2)
		{
			memcpy(&nodeid1, &cdds1[pos1], sizeof(long));
			memcpy(&nodeid2, &cdds2[pos2], sizeof(long));
			//compare the two keys
			if (nodeid1 == nodeid2)
			{//the keys are the same so sum up the counters and add it to the new dynamic VV
				memcpy(&value1, &cdds1[pos1+sizeof(long)], sizeof(long));
				memcpy(&value2, &cdds2[pos2+sizeof(long)], sizeof(long));
				resultvalue = value1 + value2;
				//copy the key and value (counter) to the resultCDDS
				memcpy(&(*resultCDDS)[resultpos], &nodeid1, sizeof(long));
				resultpos += sizeof(long);
				memcpy(&(*resultCDDS)[resultpos], &resultvalue, sizeof(long));
				resultpos += sizeof(long);
				//increasing the numberOfResultElements by 1
				numberOfResultElements++;

				//no move both position pointers to the next entry and increase the counters
				pos1 += 2*sizeof(long);
				pos2 += 2*sizeof(long);
				counter1++;
				counter2++;
			}else
			{
				if (nodeid1 < nodeid2)
				{//nodeid2 is greater than nodeid1
					memcpy(&value1, &cdds1[pos1+sizeof(long)], sizeof(long));
					//copy the key and value (counter) to the resultCDDS
					memcpy(&(*resultCDDS)[resultpos], &nodeid1, sizeof(long));
					resultpos += sizeof(long);
					memcpy(&(*resultCDDS)[resultpos], &value1, sizeof(long));
					resultpos += sizeof(long);
					//increase the numberOfResultElements by 1
					numberOfResultElements++;

					//only increase the counter and move the position pointer of node1
					pos1 += 2*sizeof(long);
					counter1++;

				}else
				{   //nodeid1 is greater than nodeid2
					memcpy(&value2, &cdds2[pos2+sizeof(long)], sizeof(long));
					//copy the key and value (counter) to the resultCDDS
					memcpy(&(*resultCDDS)[resultpos], &nodeid2, sizeof(long));
					resultpos += sizeof(long);
					memcpy(&(*resultCDDS)[resultpos], &value2, sizeof(long));
```

```
                                        resultpos += sizeof(long);
                                        // also increase the numberOfResultElements by 1
                                        numberOfResultElements++;

                                        //only increase the counter and move the position pointer of node1
                                        pos2 += 2*sizeof(long);
                                        counter2++;

                                }

                        }

                }else
                {       //at least one dynamic version vector has no more elements
                        //find out which one it is and put the elements of the other one simply at the end of the
resultCDDS
                        //if both have no elements anymore nothing is added to the resultCDDS
                        if (counter1 >= numberOfEntriesCDDS1) // CDDS1 has no new elements any more
                        {
                                memcpy( &(*resultCDDS)[resultpos], &cdds2[pos2],
(numberOfEntriesCDDS2 - counter2)*2*sizeof(long) );
                                resultpos += (numberOfEntriesCDDS2 - counter2)*2*sizeof(long);
                                //increase numberOfResultElements by the amount of remaining entries
                                numberOfResultElements += (numberOfEntriesCDDS2 - counter2);
                        }
                        if (counter2 >= numberOfEntriesCDDS2) // CDDS2 has no new elements any more
                        {
                                memcpy( &(*resultCDDS)[resultpos], &cdds1[pos1],
(numberOfEntriesCDDS1 - counter1)*2*sizeof(long) );
                                resultpos += (numberOfEntriesCDDS1 - counter1)*2*sizeof(long);
                                //increase numberOfResultElements by the amount of remaining entries
                                numberOfResultElements += (numberOfEntriesCDDS1 - counter1);
                        }
                        run = doi_false;
                }
        }// end of while
        // add the numberOfResultElements to the beginning of the resultCDDS
        memcpy( (*resultCDDS), &numberOfResultElements, sizeof(long) );
        *resultCddsSize = resultpos;
}

/*---------------------------------------------------------------------------
 FUNCTION

  splitDatum

 DESCRIPTION
        During conflict resolution it is necessary to split a datum of the
        database into the two components namely CDDS and value. This is
        necessary to merge the CDDS and to calculate a new value out of the
        original ones.

 SYNOPSIS
 INPUT
        Datum *datumValue   the input data which has to be split into its different components
 OUTPUT
        drd_CDDS *cdds                          This is the Dynamic Version Vector of the datum
(in our case)
        Datum **outputValue                     This is the value of the datum
        int *lengthOfValue                      This is the length of the value
 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
                                the drd_CDDS has to be destroyed outside
---------------------------------------------------------------------------*/
void drd_CDDSHandler::splitDatum(Datum *datumValue, drd_CDDS *cdds, Datum **outputValue, int
*lengthOfValue)
{
        long elementCount;
        int cddsEndpointer; //pointer which points to the end of the CDDS
        //the first Bytes of the ByteStream saved in the datum
        //contain the number of entries in the CDDS.
        //So this number is read out and stored in the elementCount variable
        memcpy(&elementCount, datumValue->dptr, sizeof(long));
        //by knowing the number of elements we can calculate the size of the
        //CDDS and set the cddsEndpointer which then also points to the beginning
        //of the value
        cddsEndpointer = sizeof(long) * 2 * elementCount + sizeof(long);
        //copy the CDDS out of the stream
        cdds->cddata = (char*) doi_allocMem( cddsEndpointer );
        memcpy( cdds->cddata, datumValue->dptr, cddsEndpointer );
        cdds->cddataSize = cddsEndpointer;
        //calculating the length of the value
        *lengthOfValue = datumValue->dsize - cddsEndpointer;
        //copy the value out of the Stream
        *outputValue = new Datum;
        (*outputValue)->dptr = (char*)doi_allocMem(*lengthOfValue);    //eingefï¿½gt..
        (*outputValue)->dsize = *lengthOfValue;
        memcpy( (*outputValue)->dptr, &datumValue->dptr[cddsEndpointer], *lengthOfValue );
}



/*---------------------------------------------------------------------------
 FUNCTION

   orderKeyList

 DESCRIPTION
        This function orders a list of keys (presented by datums). This is necessary because the
        Version Vector sequences have to be ordered for better performance during the comparison
        The ordering of the keys takes place before the update ByteStream is created.
        ( before marshalling --- on the sending node)
 SYNOPSIS
 INPUT
        Datum *keyList                  contains all the keys which have to be ordered
        int numberOfKeys                the number of keys which are in the list
 OUTPUT
        Datum *keyList                  is also an output parameter because after calling
                                                this procedure it contains the ordered
keys.
 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
---------------------------------------------------------------------------*/

doi_Boolean drd_CDDSHandler::orderKeyList(Datum *keyList, int numberOfKeys)
{
	Datum tempDatum;
	doi_Boolean changes = doi_true;
	if (numberOfKeys > 1)
	{
		//as long as values have to be changed do
		while(changes == doi_true)
		{
			changes = doi_false;
			for ( int i =0; i< numberOfKeys-1; i++)
			{
				if (*(long*)(keyList[i].dptr) > *(long*)(keyList[i+1].dptr) )
				{
					tempDatum = keyList[i];
					keyList[i] = keyList[i+1];
					keyList[i+1] = tempDatum;
					changes = doi_true;
				}
			}
		}
	}
	return doi_true;
}



/*---------------------------------------------------------------------------
 FUNCTION

   searchDVV

 DESCRIPTION

  This function searches through the Bytestream of a dynamic version vector
  to find an entry for a special node. The entries are ordered by the nodeidentifiers.

 SYNOPSIS
 INPUT

 dvv:   the Bytestream of the dynamic version vector
 node: the nodeidentifier of the node to search for

 OUTPUT
 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS

 This function is needed in the updateCDDS(...) function in the drd_CDDSHandler.
 There the entry of a specific node has to be increased. So you need this function
 to find the node-entry.
---------------------------------------------------------------------------*/


drd_searchResult* drd_CDDSHandler::searchDVV(char* dvv, long node)
{
	long numberOfEntries =0;
	long currentNodeIdentifier =0;
	long pos = 0;
	drd_searchResult* sr = new drd_searchResult ;
```

```
        memcpy(&numberOfEntries, dvv, sizeof(long));
        pos += sizeof(long);
        for (int i =0; i<numberOfEntries ; i++)
        {
                memcpy(&currentNodeIdentifier, &dvv[pos], sizeof(long));
                //check if the nodeidentifiers are the same
                if (currentNodeIdentifier == node)
                { //we found the entry for the node
                        sr->newEntry = doi_false;
                        sr->index = i+1;
                        return sr;
                }

                //check if the nodeidentifier in the dvv is higher than the node we search
                if (currentNodeIdentifier > node)
                {//the node has to be insert before the currentNodeIdentifier
                        sr->newEntry = doi_true;
                        sr->index = i+1;  //+1 for cause the index i starts from 0
                        return sr;
                }
                //jump to the next entry
                pos += 2* sizeof(long);
        }
        sr->newEntry = doi_true;
        sr->index = numberOfEntries+1;
        return sr;
}

/*---------------------------------------------------------------------------
 FUNCTION

   addCDDStoSeq

 DESCRIPTION
 This function adds a Conflict Detection Data Structure to the end of an
 CDDS sequence.

 SYNOPSIS
 INPUT

 newCDDS:   the Conflict Detection Data Structure (e.g.: a dynamic Version Vector)
 sequence:   the CDDS sequence to which the newCDDS should be added

 OUTPUT
 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS
 This function is used within the createCDDSSeq(...) function of the drd_CDDSHandler

---------------------------------------------------------------------------*/

void drd_CDDSHandler::addCDDStoSeq(drd_CDDS* newCDDS, drd_CDDSseq* sequence)
{
  drd_CDDS *tempCDDS;
  //if the sequence is empty put the CDDS to the head
  if(sequence->head == 0) {
    tempCDDS = sequence->head = (drd_CDDS*) doi_allocMem(sizeof(drd_CDDS));
  }
  else {
```

```
              //if the sequence is not empty start at the head and go to the next
              //sequence until this is 0. Then append the new sequence at the end.
    tempCDDS=sequence->head;
    while(tempCDDS->next != 0)
      tempCDDS=tempCDDS->next;
    tempCDDS = tempCDDS->next = (drd_CDDS*) doi_allocMem(sizeof(drd_CDDS));
  }

   /* Copy the new version vector into the allocated position. */
  memcpy(tempCDDS, newCDDS, sizeof(drd_CDDS));
  tempCDDS->next=0; //the last element must always be 0

}

/*---------------------------------------------------------------------------
 FUNCTION

   keyCompare

 DESCRIPTION
 This function compare two keys

 SYNOPSIS
 INPUT
 OUTPUT
 RETURNS
 EXCEPTIONS
 ERRORHANDLER POSSIBILITIES
 REMARKS

---------------------------------------------------------------------------*/

doi_Boolean drd_CDDSHandler::keyCompare(char* key1, char* key2, int key1size, int key2size)
{
              if (key1size != key2size)
                            return doi_false;

              for (int i = 0; i<key1size; i++)
              {
                            if (key1[i] != key2[i])
                                          return doi_false;
              }

              return doi_true;
}
```

Log Filter (drd_LogFilter.cc)

```
//constructor

drd_LogFilterHandler::drd_LogFilterHandler()
{
  lf.noSequences = 0; //initializing the logFilter
  lf.first = 0;
  lf.last = 0;
}


/*------------------------------------------------------------------------------ FUNCTION

giveBackLogFilterHandler

DESCRIPTION

This function gives back an instance of a specific Logfilter. For example an ASAP logfilter.

SYNOPSIS INPUT

repType   the replication type

OUTPUT RETURNS drd_LogFilterHandler*          the reference to the Log Filter EXCEPTIONS
ERRORHANDLER POSSIBILITIES REMARKS
----------------------------------------------------------------------------*/
drd_LogFilterHandler * drd_LogFilterHandler::giveBackLogFilterHandler( drd_replicationType repType )
{

  switch ( repType )
  {
    case ASAP:
      if ( asap == 0 )
        asap = new drd_LogFilterHandler();
      return asap;
    break;
    case BOUNDED:
      if ( bounded == 0 )
        bounded = new drd_LogFilterHandler();
      return bounded;
    break;
    default:
      return 0;
  }
}


/*------------------------------------------------------------------------------ FUNCTION

detectConflicts

DESCRIPTION

This functions detects conflicts and gives back a short report to the caller

SYNOPSIS INPUT
```

sequence the CDDS sequence which shall be checked for conflicts

OUTPUT RETURNS drd_ConflictReport* the conflict report which contains all information about conflicts that occured so
that the Conflict Handler gets informed. EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS

```
---------------------------------------------------------------------------*/
drd_ConflictReport * drd_LogFilterHandler::detectConflicts( drd_CDDSseq * sequence )
{ //test
  drd_CDDS * conflictingCDDS = ( drd_CDDS * ) doi_allocMem( 2 * sizeof( drd_CDDS ) );;
  drd_CDDSseq * tempLast = 0;
  drd_CDDSseq * tempNext;
  drd_CDDSseq * insertSeqAfter = 0;
  drd_ConflictReport * confReport = new drd_ConflictReport;
  // to set the pointers to 0 -- so destructor knows which things to delete
  init_drd_ConflictReport( confReport );

  doi_Boolean dominatedByAll = doi_true;
  doi_Boolean dominated = doi_false; //to recognize cycles - if the sequence was dominated and suddenly
dominates => conflict
  //to check if the insertSeqAfter is empty is not ok because if the first sequence in the lf dominates the sequence
there is
  //this variable is still null but the sequence was dominated

  drd_Rc rc;
  tempNext = lf.first;
  if ( tempNext != 0 )
  {
    while ( tempNext != 0 )
    {
      if ( areStringsEqual( sequence->pathname, tempNext->pathname ) )
      { //compare the two conflict detection data structures
        rc = drd_CDDSHandler::compare( sequence, tempNext, conflictingCDDS );

        switch ( rc )
        {
          case DRD_DOMINATES: // the sequence dominates tempNext
            // we have to check for cycles
            if ( dominated )
            { // rw cycle conflict detected (the sequence was dominated and now dominates
              confReport->WWConflict = doi_false;
              confReport->RWConflict = doi_false;
              confReport->RWCycle = doi_true;
              //give back the position where it should be insert
              confReport->conflictingSeq = insertSeqAfter;
              return confReport;
            }
            dominatedByAll = doi_false;
          break;
          case DRD_DOMINATED:
            if ( insertSeqAfter == 0 ) // the sequence is dominated by tempNext

              insertSeqAfter = tempLast; //if it is the first time the sequence is dominated
            dominated = doi_true;
          break;
          case DRD_WW_CONFLICT:
            confReport->WWConflict = doi_true; // write write conflict -- report that
            confReport->RWConflict = doi_false;
            confReport->RWCycle = doi_false;
            confReport->conflictingCDDS = ( drd_CDDS * ) doi_allocMem( sizeof( drd_CDDS ) );
            confReport->currentCDDS = ( drd_CDDS * ) doi_allocMem( sizeof( drd_CDDS ) );
            //the currentCDDS conflicts with this CDDS from the logfilter
            memcpy( confReport->conflictingCDDS, conflictingCDDS, sizeof( drd_CDDS ) );
            //the second CDDS in the variable conflicting CDDS is the currentCDDS
            memcpy( confReport->currentCDDS, & conflictingCDDS[sizeof( drd_CDDS )], sizeof( drd_CDDS ) );
            return confReport;
          break;
          case DRD_RW_CONFLICT:
            confReport->WWConflict = doi_false;
            confReport->RWConflict = doi_true;
            confReport->RWCycle = doi_false;
            confReport->conflictingSeq = tempNext;
            return confReport;
          break;
          case DRD_INDEPENDENT: //it doesn't matter how to order these two conflict sequences
            //TODO we have to check if the sequences are equal...then we do not have to add them
            //
                                                    confReport->WWConflict = doi_false;
            //
                                                    confReport->RWConflict = doi_false;
            //
                                                    confReport->RWCycle = doi_false;
            //
                                                    return confReport;
            if ( insertSeqAfter == 0 );
            insertSeqAfter = tempLast;
          break;
          default: //Error
            std::cout << "Error status not recognized" << std::endl;
        }
      }
      // go to the next sequence in the logfilter
      tempLast = tempNext;
      tempNext = tempNext->next;
    } // end of while

    if ( insertSeqAfter == 0 ) //if the sequence dominated all the sequences in the logfilter
      insertSeqAfter = lf.last;
    if ( dominatedByAll )
      insertSeqAfter = 0;
    insertSequence( insertSeqAfter, sequence );
    confReport->WWConflict = doi_false;
    confReport->RWConflict = doi_false;
    confReport->RWCycle = doi_false;

  }
  else
  { //there are no entries in the logfilter -- so we put the sequence to the top
    lf.first = sequence;
    sequence->next = 0;
    lf.last = lf.first;
    lf.noSequences++;
    confReport->WWConflict = doi_false;
    confReport->RWConflict = doi_false;
    confReport->RWCycle = doi_false;
  }

  //free memory
  doi_freeMemCPP( & conflictingCDDS );
```

```
  return confReport;
}


/*--------------------------------------------------------------------------- FUNCTION

integrateLocalUpdate

DESCRIPTION

This function just adds a CDDS sequence at the end of the logfilter. This is only for local updates because those
sequences
can not be in conflict with existing ones.

SYNOPSIS INPUT

sequence:    the sequence which should be appended to the logfilter

OUTPUT RETURNS doi_Boolean             indicating if all is all right EXCEPTIONS ERRORHANDLER
POSSIBILITIES REMARKS
---------------------------------------------------------------------------*/
doi_Boolean drd_LogFilterHandler::integrateLocalUpdate( drd_CDDSseq * sequence )
{
  drd_CDDSseq * oldLast;

  if ( lf.first == 0 )
  { //no element in logfilter
    lf.first = sequence;
    lf.last = sequence;
  }
  else
  { //one element in logfilter
    if ( lf.first == lf.last )
    {
      lf.last = sequence;
      lf.first->next = lf.last;
    }
    else
    { //more than one elements in logfilter
      oldLast = lf.last;
      lf.last = sequence;
      oldLast->next = lf.last;
    }
  }

  lf.last->next = 0; //the last entry points to 0

  //increase the number of sequences in the logfilter
  lf.noSequences++;
  return doi_true; //doi_boolean verwenden
}



/*--------------------------------------------------------------------------- FUNCTION

insertSequence

DESCRIPTION
```

This function is used within the Log Filter. It inserts a sequence to the Log Filter.

SYNOPSIS INPUT

drd_CDDSseq *insertSeqAfter the sequence after which the new one should be integrated
drd_CDDSseq *seqToInsert the
new sequence which shall be integrated in the Log Filter

OUTPUT RETURNS EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS
--*/

```
void drd_LogFilterHandler::insertSequence( drd_CDDSseq * insertSeqAfter, drd_CDDSseq * seqToInsert )
{
  drd_CDDSseq * tempNext;

  if ( insertSeqAfter == 0 ) // the sequence hast to be added at the beginning
  {
    drd_CDDSseq * tempSeq;
    tempSeq = lf.first;
    lf.first = seqToInsert;
    seqToInsert->next = tempSeq;
    return;
  }

  tempNext = insertSeqAfter->next;

  if ( tempNext == 0 ) //insert sequence at the end
  {
    insertSeqAfter->next = seqToInsert;
    seqToInsert->next = 0;
    lf.last = seqToInsert;
  }
  else
  { //insert sequence in the middle
    insertSeqAfter->next = seqToInsert;
    seqToInsert->next = tempNext;
  }
}

//checks if two char are the same
```

/*-- FUNCTION

integrateLocalUpdate

DESCRIPTION

checks if two char are the sameing ones.

SYNOPSIS INPUT

char *string1 first char* char *string2 second char*

OUTPUT RETURNS doi_Boolean indicating if both are equal (true). EXCEPTIONS
ERRORHANDLER POSSIBILITIES REMARKS
--*/

```
doi_Boolean drd_LogFilterHandler::areStringsEqual( char * string1, char * string2 )
{
  if ( strcmp( string1, string2 ) == 0 )
    return doi_true;
  return doi_false;
}
```

Average Resolver (drd_Resolver.cc)

```
/*----------------------------------------------------------------------------- FUNCTION

resolveWWConflict

DESCRIPTION

This function resolves write/write conflicts by calculating the average Value of the values which are in conflict.

SYNOPSIS INPUT drd_CDDS *cddsVector                         the CDDS of the participating entries
Datum *values                                   the values of the
participating entries int  numberOfValues                   the number of values in the above vectors
(always have to be the same for both
vectors) OUTPUT drd_CDDS *mergedCDDS                        the merged CDDS
        Datum *mergedValue                          the merged value RETURNS EXCEPTIONS
ERRORHANDLER POSSIBILITIES REMARKS

-----------------------------------------------------------------------------*/
doi_Boolean resolveWWConflict( drd_CDDS * cddsVector, Datum * values, int numberOfValues,
   drd_CDDS * mergedCDDS, Datum * mergedValue )
   {
    long tempValue = 0;
    int resultCddsSize = 0;
    char * currentCDDS;
    char * newCDDS;
    char * resultCDDS;

    if ( numberOfValues > 1 )
    {
     // calculate the average value
     for ( int i = 0; i < numberOfValues; i++ )
     {
      tempValue += * ( long * ) ( values[i].dptr );
     }
     tempValue = tempValue / numberOfValues;
     mergedValue->dptr = ( char * ) doi_allocMem( sizeof( long ) );
     memcpy( mergedValue->dptr, & tempValue, sizeof( long ) );

     // merge the CDDS
     currentCDDS = cddsVector[0].cddata;
     for ( int k = 0; k < numberOfValues - 1; k++ )
     {
      newCDDS = cddsVector[k + 1].cddata;
      drd_CDDSHandler::mergeCDDS( currentCDDS, newCDDS, & resultCDDS, & resultCddsSize );
      currentCDDS = resultCDDS;
     }
    }
    else
    {
     return doi_false;
    }
    mergedCDDS->cddata = ( char * ) doi_allocMem( resultCddsSize );
    memcpy( mergedCDDS->cddata, currentCDDS, resultCddsSize );
    mergedCDDS->cddataSize = resultCddsSize;

    //delete the cddsVector and the values because they are merged and so no longer needed
    // the results are in the mergedCDDS and mergedValue variables.
    for ( int k = 0; k < numberOfValues - 1; k++ )
```

```
        {
          //TODO:  delete the referenced objects in the vector cddsVector and values
          //like the key etc.
          //destroy_drd_CDDS( (drd_CDDS*)&cddsVector[k] );
          //destroy_Datum( &values[k]);
        }
        doi_freeMemCPP( & cddsVector );
        doi_freeMemCPP( & values );


        if ( currentCDDS != 0 )
          doi_freeMemCPP( & currentCDDS );



        return doi_true;
}
```

Conflict Detection Data Structure Maintainance (drd_CDDS.cc)

```
// -- Destructors of the data structures ---
// Many data structures are not simple and so the destruction is
// more complex. To hide it from the rest of the code these functions
// are defined here.
void destroy_drd_CDDS( drd_CDDS * object )
{
  //delete the object
  if ( object->key != 0 )
    doi_freeMemCPP( & object->key );
  if ( object->cddata != 0 )
    doi_freeMemCPP( & object->cddata );
  delete object;
}

void destroy_drd_CDDSseq( drd_CDDSseq * object )
{
  //delete the object
  drd_CDDS nextCDDS;
  drd_CDDS tempCDDS = * object->head;

  if ( object->head == 0 )
  {
    delete object;
    return;
  }

  while ( true )
  {
    if ( tempCDDS.next != 0 )
    {
      nextCDDS = * tempCDDS.next;
      destroy_drd_CDDS( & tempCDDS );
      tempCDDS = nextCDDS;
    }
    else
    {
      destroy_drd_CDDS( & tempCDDS );
      delete object;
      return;
    }
  }
}

void destroy_drd_LogFilter( drd_LogFilter * object )
{
  //delete the object
  drd_CDDSseq tempSeq = * object->first;
  drd_CDDSseq nextSeq;

  if ( object->first == 0 )
  {
    //delete object;   -- the logfilter is part of the logfilterHandler and so automatically destroyed
    return;
  }

  while ( true )
  {
    if ( tempSeq.next != 0 )
    {
      nextSeq = * tempSeq.next;
      destroy_drd_CDDSseq( & tempSeq );
      tempSeq = nextSeq;
    }
    else
    {
      destroy_drd_CDDSseq( & tempSeq );
      //delete object;
      return;
    }
  }
}

void destroy_drd_ConflictReport( drd_ConflictReport * object )
{
  //delete the object
  if ( object->conflictingSeq != 0 )
    doi_freeMemCPP( & object->conflictingSeq );
  if ( object->conflictingCDDS != 0 )
    doi_freeMemCPP( & object->conflictingCDDS );
  if ( object->currentCDDS != 0 )
    doi_freeMemCPP( & object->currentCDDS );
}

void destroy_Datum( Datum * object )
{
  if ( object->dptr != 0 )
    doi_freeMemCPP( & object->dptr );
  delete object;
  return;
}


// The next part contains init functions which are used to initialize
// the data structures after their creation


void init_drd_CDDS( drd_CDDS * object )
{
  //init  the object
  object->key = 0;
  object->cddata = 0;
  object->next = 0;
}

void init_drd_CDDSseq( drd_CDDSseq * object )
{
  //init  the object
  object->head = 0;
}

void init_drd_LogFilter( drd_LogFilter * object )
{
  //init  the object
  object->first = 0;
  object->last = 0;
}
```

```
void init_drd_ConflictReport( drd_ConflictReport * object )
{
  //init the object
  object->conflictingSeq = 0;
  object->conflictingCDDS = 0;
  object->currentCDDS = 0;
}
```

Data Structure Definitions (drd_CDDS.h)

```
/* Types definitions */

/* This is a single CDDS (Conflict Detection Data Structure) * which contains additional information
like the key it belongs to */
typedef struct drd_CDDS
{
  char * key;
  int keySize;
  /* confilct detection datastructure */
  char * cddata;
  int cddataSize;
  struct drd_CDDS * next;
}
drd_CDDS;



/*This defines a CDDS (Conflict Detection Data Structure) sequence
* which holds all the CDDS of one transaction -- you can say that such
* a data structure represents a transaction during the conflict detection */
typedef struct drd_CDDSseq
{
  /* the pathname of the dbm file associated to the seq / transaction */
  char pathname[MAX_PATH_LENGTH];
  drd_CDDS * head;
  /* this contains a link to the next sequence in the Log Filter */
  struct drd_CDDSseq * next;
}
drd_CDDSseq;



/* This data structure the Log Filter is working on. It contains all the
* CDDS sequences of all the transaction which were integrated. */
typedef struct drd_LogFilter
{
  /* Indicate the number of sequences in the log filter. */
  int noSequences;
  drd_CDDSseq * first;
  drd_CDDSseq * last;
}
drd_LogFilter;



/* This data structure is used to give back the status of the conflict detection * to the Conflict Handler.
* It contains detailed information about the type of conflict which occured. */

typedef struct drd_ConflictReport
{
  doi_Boolean WWConflict;
  doi_Boolean RWConflict;
  doi_Boolean RWCycle;
  drd_CDDSseq * conflictingSeq;
  drd_CDDS * conflictingCDDS;
  /* if a WWconflict occures the conflicting CDDS has to be saved */
  drd_CDDS * currentCDDS;
  /* pointer to the conflicting CDDS in the sequence we want to integrate */

}
drd_ConflictReport;



/* This is the data structure which is used within the conflict detection
* to indicate in which relation two CDDS or CDDS sequences are.
* It is not passed outside the conflict detection. This is done via the * drd_ConflictReport. */
typedef enum drd_Rc
{
  DRD_INDEPENDENT, DRD_DOMINATES, DRD_DOMINATED, DRD_WW_CONFLICT,
DRD_RW_CONFLICT
}
drd_Rc;


/* This data structure is still available but not really used at the moment because
* the replication type is always set to ASAP */
typedef enum drd_replicationType
{
  ASAP, BOUNDED
}
drd_replicationType;

/* This enumeration indicates if the operation type is read or write */
typedef enum
{
  drd_Read, drd_Write
}
drd_OperationType;



/* This data structure represents a header which is saved for every read
* or written value in the update message. It contains information like
* the type of access (read or write) and the different sizes. */
typedef struct drd_LogHeader
{
  drd_OperationType opType;
  char pathname[MAX_PATH_LENGTH];
  EntryDesc keyDesc;
  int keySize;
  EntryDesc valueDesc;
  int valueSize;
  /* the size of the conflict detection datastructure within the value */
  long cddataSize;
}
drd_LogHeader;



/* Is used during the update of a VV. It gives back the position at which * the value has to be increased and
additionally
indicates whether an entry * already exists there or a new one has to be created. */
typedef struct drd_searchResult
{
  long index;
  /* at which point the new entry have to be insert -- or which entry has to be increased */
  doi_Boolean newEntry;
}
drd_searchResult;



/* destructors */
void destroy_drd_CDDS( drd_CDDS * object );
void destroy_drd_CDDSseq( drd_CDDSseq * object );
void destroy_drd_LogFilter( drd_LogFilter * object );
void destroy_drd_ConflictReport( drd_ConflictReport * object );
void destroy_Datum( Datum * object );

/* initializers */
void init_drd_CDDS( drd_CDDS * object );
void init_drd_CDDSseq( drd_CDDSseq * object );
void init_drd_LogFilter( drd_LogFilter * object );
void init_drd_ConflictReport( drd_ConflictReport * object );

#endif /* DRD_CDDS_H */
```

Logger (drd_Logger.cc) – only parts changed

```
/* This number indicates which entry in the version vector * corresponds to the actual node. */
long localNodeNumber;


/*---------------------------------------------------------------------------- FUNCTION

drd_normalTrans

DESCRIPTION

This functions returns true if the argument is a replicated transaction.

SYNOPSIS INPUT OUTPUT RETURNS EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS
----------------------------------------------------------------------------*/
doi_Boolean drd_normalTrans( Tid * tid )
{
  return ( tid->drd_logInfo.transType == drd_ReplicatedTrans ? doi_true : doi_false );
}

/*---------------------------------------------------------------------------- FUNCTION

drd_checkIfReadOnlyTrans

DESCRIPTION

Returns true if the argument is a read-only transaction.

SYNOPSIS

if (drd_checkIfReadOnlyTrans(tid)) { ... }

INPUT

Tid *tid The transaction ID.

OUTPUT RETURNS

doi_true  - if the readOnly flag is true doi_false - if the readOnly flag is false

EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS -----------------------------------------------------
---------------------*/
doi_Boolean drd_readOnlyTrans( Tid * tid )
{
  return tid->drd_logInfo.readOnlyTrans;
}

/*---------------------------------------------------------------------------- FUNCTION

drd_setWriteFlag

DESCRIPTION

Set the readOnly flag to false.

SYNOPSIS
```

```
drd_setWriteFlag(tid)

INPUT

Tid *tid The transaction ID of the transaction.

OUTPUT RETURNS EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS
---------------------------------------------------------------------------*/
void drd_setWriteFlag( Tid * tid )
{
  tid->drd_logInfo.readOnlyTrans = doi_false;
}

/*--------------------------------------------------------------------------- FUNCTION

drd_setNodeNumber

DESCRIPTION

Allows to set the node number from outside.

SYNOPSIS INPUT OUTPUT RETURNS EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS
---------------------------------------------------------------------------*/
void drd_setNodeNumber( long nodeNumberInit )
{
  localNodeNumber = nodeNumberInit; //nodeNumberInit - 1 changed to nodeNumberInit
}


long drd_getNodeNumber()
{
  return localNodeNumber;
}


/*--------------------------------------------------------------------------- FUNCTION

drd_createLog

DESCRIPTION

Creates a volatile log file and starts a transaction which is used to store entries in the log.

SYNOPSIS

DbmRc drd_createLog(Tid *tid, drd_TransType transType)

INPUT

Tid *tid The transaction ID of the transaction that shall be logged.

drd_TransType transType The replication type (asap, bounded).

OUTPUT RETURNS EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS
---------------------------------------------------------------------------*/
DbmRc drd_createLog( Tid * tid, drd_TransType transType )
{
  DbmRc rc;
  drd_LogInfo & logInfo = tid->drd_logInfo;
  /* Initializing the logInfo strucure in the main transaction. */
  logInfo.transLog = NULL;
  logInfo.logTid = NULL;
  logInfo.transType = drd_ReplicatedTrans;
  logInfo.readOnlyTrans = doi_true;
  logInfo.noObjects = 0;
  logInfo.totalLogSize = 0;


  /* Open a log file, begin log transaction, and initialize the * logInfo structure for the log transaction. */
  if ( ( rc = dtd_open( NULL, DBM_VOLATILE, NULL, & logInfo.transLog, doi_false, NULL ) ) != DBM_OK
)
  {
    return rc;
  }

  /* Begin the logging transaction. This should be a local * transaction, since we don't want it to be replicated.
We use the
  * internal interface of TDBM, since the global lock will always be * locked when we use this transaction. */
  if ( ( rc = dtd_begin( NULL, & logInfo.logTid, drd_LocalTrans ) ) != DBM_OK )
  {
    return rc;
  }

  /* Initialize the drd_LogInfo structure for the logging transaction. */
  logInfo.logTid->drd_logInfo.transType = drd_LocalTrans;
  logInfo.logTid->drd_logInfo.readOnlyTrans = doi_true;
  logInfo.logTid->drd_logInfo.noObjects = 0;
  logInfo.logTid->drd_logInfo.totalLogSize = 0;

  return DBM_OK;
}

/*-------------------------------------------------------------------------- FUNCTION

drd_updateCDDS

DESCRIPTION

Update the version vector for an object in the database by calling the updateCDDS-function in the
CDDSHandler...

SYNOPSIS INPUT

Tid *tid; Dbm *dbm; Datum key This is database information and is used to determine wheter the
version vector shall be increased or not.

Datum *value This is the updated value received from the application

Datum *oldEntry This is the complete entry, that is, data plus version vector that
shall be updated through the value received from above.

OUTPUT

Datum *newEntry This is the result when the value and the old increased vv is merged. This
is what is finaly stored in the database. Must be deallocated by caller.

RETURNS EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS
--------------------------------------------------------------------------*/
```

```
void drd_updateCDDS( Tid * tid, Dbm * dbm, Datum key, Datum * value, Datum * oldEntry, Datum *
newEntry, long * cddsSize )
{
  // VV has to be updated
  Datum * newEntryTemp;
  newEntryTemp = drd_CDDSHandler::updateCDDS( value, oldEntry, localNodeNumber, tid, dbm, & key,
cddsSize );
  * newEntry = * newEntryTemp;
  //free the memory -- value has been copied
  delete newEntryTemp;
}
```

/*-- FUNCTION

drd_removeCDDS

DESCRIPTION

Removes the CDDS structure which is stored as an prefix to the normal value in the database. Before the value is given back to
the application the CDDS has to be removed. Otherwise the value would be wrong.

SYNOPSIS INPUT

Datum *value This is the value received from the application from which the CDDS shall be removed.

long *cddsSize This is the size of the CDDS. It is necessary to specify how many Bytes have to be removed from the value.

OUTPUT

Datum *value This variable is also an output parameter because after the execution of the function it includes the truncated
content. (the right value without CDDS)

RETURNS EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS
--*/

```
void drd_removeCDDS( Datum * value, long * cddsSize )
{
  // nodeNumber has to be added -- this is done here
  drd_CDDSHandler::removeCDDS( value, cddsSize );
}
```

/*-- FUNCTION

drd_storeLogEntry

DESCRIPTION

Stores a value in the database. There are some conditions that are evaluated
to determine wheter a value shall be stored or not. If a value is already
stored in the database it is overwritten if the new is a write operation on that value.

SYNOPSIS

drd_storeLogEntry(dbm, tid, key, value, drd_Write)

INPUT

Dbm dbm Used to store the name of the file in which the modified value is stored.

Tid *tid The transaction id. This contains information about the log file.

Datum key The key to the value that has been read or modified.

Datum value This contains the new value for a modfied object. If the object i only read, this i NULL.

drd_OperationType opType The type of operation made to the value. Can either be drd_Read or drd_Write.

OUTPUT RETURNS EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS

```
--------------------------------------------------------------------------*/
DbmRc drd_storeLogEntry( Dbm * dbm, Tid * tid, Datum key, Datum value, drd_OperationType opType, long
cddsSize )
{
  DbmRc rc;
  Datum logValue;
  Datum logValueOld;
  drd_LogHeader newHeader;
  bool newValue = false;

  /* Constants that define a log value Datum. */
  const EntryDesc desc = ALIGN2;

  /* Check to see if this object is already stored in the log database. */
  rc = dtd_fetch( tid->drd_logInfo.transLog, tid->drd_logInfo.logTid, key, & logValueOld, 0 );
  if ( rc == DBM_ENTRY_NOT_FOUND || rc == DBM_OK )
  {
    if ( rc == DBM_ENTRY_NOT_FOUND )
    {
      newValue = true;
    }
    else
    {
      /* If this is a read operation and the value has already been * read and/or written, we do not need to do
anything (the
      * version vector hasn't changed and the value has not been * updated). */
      if ( opType == drd_Read )
      {
        return DBM_OK;
      }
      newValue = false;
    }

    /* Fill in header */
    newHeader.opType = opType;
    strcpy( newHeader.pathname, dbm->dbmfile->pathname );
    newHeader.keyDesc = key.desc;
    newHeader.keySize = key.dsize;
    newHeader.valueDesc = value.desc;
    newHeader.valueSize = value.dsize;
    newHeader.cddataSize = cddsSize;
    //drd_htonLogHeader(&newHeader);

    /* Set up the logvalue */
    logValue.desc = desc;

    /* Calculate the size of the new logvalue. If the operation is a
    * read-operation, then only the version vector is stored in the * log, not the value itself. */
    int valueSize;
    switch ( opType )
    {
      case drd_Read:
        /* This is a read -> only copy the version vector of the dptr */
        valueSize = cddsSize; //changed ... sizeof(drd_VerVec);
      break;
      case drd_Write:
        /* This is a write -> store the complete value (which includes * the version vector) */
        valueSize = value.dsize;
      break;
    }
    logValue.dsize = sizeof( drd_LogHeader ) + key.dsize + valueSize;
    logValue.dptr = ( char * ) doi_allocMem( logValue.dsize );

    /* Copy header, key, and value into the log value. */
    memmove( logValue.dptr, & newHeader, sizeof( drd_LogHeader ) );
    memmove( & logValue.dptr[sizeof( drd_LogHeader )], key.dptr, key.dsize );
    memmove( & logValue.dptr[sizeof( drd_LogHeader ) + key.dsize], value.dptr, valueSize );

    /* Convert version vector to network byte order. */
    //drd_htonVV((drd_VerVec*) &logValue.dptr[sizeof(drd_LogHeader) + key.dsize]);

    /* drd_verifyLog(&logValue); */

    /* Store value in the log database. */
    rc = dtd_store( tid->drd_logInfo.transLog, tid->drd_logInfo.logTid, key, logValue, DBM_REPLACE );
    if ( rc != DBM_OK )
    {
      fprintf( stderr, "Error while storing a logValue: %s\n", DbmErrorString( rc ) );
      doi_freeMem( ( void * * ) & logValue.dptr );
      return rc;
    }

    /* Increase the number of objects in the log and their total size * to the counters. */
    if ( newValue )
    {
      tid->drd_logInfo.noObjects++;
      tid->drd_logInfo.totalLogSize += logValue.dsize;
    }
    else
    {
      tid->drd_logInfo.totalLogSize += ( logValue.dsize - logValueOld.dsize );
    }

    doi_freeMem( ( void * * ) & logValue.dptr );
  }
  else
  {
    /* Some error */
    fprintf( stderr, "DRD_LOGGER ERROR: While reading logvalue: %s\n", DbmErrorString( rc ) );
    return rc;
  }

  return DBM_OK;
}

/*------------------------------------------------------------------------- FUNCTION

drd_fetchLog
```

DESCRIPTION

Fetches the log produced by a transaction and prepares it to be sent to the propagator.

SYNOPSIS

Datum mergedLog; drd_fetchLog(tid->drd_logInfo, &mergedLog);

INPUT

drd_LogInfo *drd_logInfo Loginfo about the transaction.

OUTPUT

Datum *mergedLog A merged log entry. mergedLog->dptr must be deallocated by the caller.

RETURNS EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS

```
---------------------------------------------------------------------------*/
DbmRc drd_fetchLog( const drd_LogInfo * drd_logInfo, Datum * mergedLog )
{
  DbmRc rc;
  Datum logValueKey;
  Datum logValue;
  int pos = 0;
  long noObjects;
  Datum * keyList;
  /* Plus a long for a object counter. */
  mergedLog->dsize = drd_logInfo->totalLogSize + sizeof( long );
  mergedLog->dptr = ( char * ) doi_allocMem( mergedLog->dsize );

  //noObjects = htonl(drd_logInfo->noObjects);
  noObjects = drd_logInfo->noObjects;
  memmove( mergedLog->dptr, & noObjects, sizeof( long ) );
  pos = sizeof( long );

  keyList = ( Datum * ) doi_allocMem( noObjects * sizeof( Datum ) );
  int keyIndex = 0;
  /* Iterate through the database and create the keylist */
  rc = dtd_firstKey( drd_logInfo->transLog, drd_logInfo->logTid, & logValueKey );
  while ( rc == DBM_OK )
  {
    keyList[keyIndex] = logValueKey;
    rc = dtd_nextKey( drd_logInfo->transLog, drd_logInfo->logTid, & logValueKey );
    keyIndex++;
  }
  if ( !drd_CDDSHandler::orderKeyList( keyList, noObjects ) )
  {
    fprintf( stderr, "DRD_LOGGER ERROR: Error while ordering the list \n" );
    return rc;
  }
  /* Iterate through the keyList and fetch all entries. */
  for ( int k = 0; k < noObjects; k++ )
  {
    rc = dtd_fetch( drd_logInfo->transLog, drd_logInfo->logTid, keyList[k], & logValue, 0 );
    if ( rc == DBM_OK )
    {
      memmove( & mergedLog->dptr[pos], logValue.dptr, logValue.dsize );
      pos += logValue.dsize;
    }
    else
    {
      fprintf( stderr, "DRD_LOGGER ERROR: Error while fetching values from the log.\n" );
      return rc;
    }
  }


  //lock logfilter
  //update logfilter -- transaction type is fixed to ASAP at the moment
  informAboutLocalTrans( ASAP, mergedLog->dptr );
  //release lock

  if ( rc == DBM_ENTRY_NOT_FOUND )
  {
    return DBM_OK;
  }

  /* drd_verifyLog(mergedLog); */
  /* debug */

  return rc;
}




/*-------------------------------------------------------------------------- FUNCTION

drd_destroyLog

DESCRIPTION SYNOPSIS INPUT OUTPUT RETURNS EXCEPTIONS ERRORHANDLER
POSSIBILITIES REMARKS
---------------------------------------------------------------------------*/

/* Destroyes the log file. */
DbmRc drd_destroyLog( const drd_LogInfo * drd_logInfo )
{
  DbmRc rc = DBM_OK;
  if ( drd_logInfo->transType == drd_ReplicatedTrans )
  {
    if ( ( rc = dtd_commit( drd_logInfo->logTid ) ) != DBM_OK )
    {
      return rc;
    }

    if ( ( rc = dtd_close( drd_logInfo->transLog, doi_true ) ) != DBM_OK )
    {
      return rc;
    }
  }
  return rc;
}

/*-------------------------------------------------------------------------- FUNCTION

drd_finishLogging

DESCRIPTION
```

Send the data which has been updated to the propagator. The logging transaction is commited and the log file is closed.

SYNOPSIS

void drd_finishLogging(drd_LogInfo drd_logInfo)

INPUT

drd_LogInfo drd_logInfo Contains all the information about the log transaction such as Tid and Dbm.

OUTPUT RETURNS EXCEPTIONS ERRORHANDLER POSSIBILITIES REMARKS

```
--------------------------------------------------------------------------*/
DbmRc drd_finishLogging( const drd_LogInfo * drd_logInfo, Datum * mergedLog )
{
  DbmRc rc = DBM_OK;
  //fetches the log out of the LogFile -- the mergedLog variable is a ByteStream
  if ( drd_logInfo->transType == drd_ReplicatedTrans && !drd_logInfo->readOnlyTrans )
  {
    if ( ( rc = drd_fetchLog( drd_logInfo, mergedLog ) ) != DBM_OK )
    {
      return rc;
    }
    //send message to the Propagator
    drd_sendUpdate( mergedLog );

    /* Free memory allocated in drd_fetchLog() */
    doi_freeMem( ( void * * ) & mergedLog->dptr );
  }

  drd_destroyLog( drd_logInfo );
  return ( rc );
}


//Are not used at the moment and should be adapted if there is a need
//for them

/* Convert log header from host byte order to network byte order. */
void drd_htonLogHeader( drd_LogHeader * lh )
{
  lh->opType = ( drd_OperationType )htonl( lh->opType );
  lh->keySize = htonl( lh->keySize );
  lh->valueSize = htonl( lh->valueSize );
}

/* Convert log header from network byte order to host byte order. */
void drd_ntohLogHeader( drd_LogHeader * lh )
{
  lh->opType = ( drd_OperationType )ntohl( lh->opType );
  lh->keySize = ntohl( lh->keySize );
  lh->valueSize = ntohl( lh->valueSize );
}
```