On recovery and consistency preservation in distributed real-time database systems
HS-IDA-MD-00-015

Sanny Gustavsson

Submitted by Sanny Gustavsson to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the Department of Computer Science.

September 2000

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

____

Sanny Gustavsson

Abstract

In this dissertation, we consider the problem of recovering a crashed node in a distributed database. We especially focus on real-time recovery in eventually consistent databases, where the consistency of replicated data is traded off for increased predictability, availability and performance. To achieve this focus, we consider consistency preservation techniques as well as recovery mechanisms.

Our approach is to perform a thorough literature survey of these two fields. The literature survey considers not only recovery in real-time, distributed, eventually consistent databases, but also related techniques, such as recovery in main-memory resident or immediately consistent databases. We also examine different techniques for consistency preservation.

Based on this literature survey, we present a taxonomy and state-of-the-art report on recovery mechanisms and consistency preservation techniques. We contrast different recovery mechanisms, and highlight properties and aspects of these that make them more or less suitable for use in an eventually consistent database. We also identify unexplored areas and uninvestigated problems within the fields of database recovery and consistency preservation. We find that research on real-time recovery in distributed databases is lacking, and we also propose further investigation of how the choice of consistency preservation technique affects (or should affect) the design of a recovery mechanism for the system.

Contents

1 Introduction **1**

1.1 Dissertation overview 3

2 Background **4**

2.1 The transaction concept 4

2.1.1 Atomicity 5

2.1.2 Consistency 6

2.1.3 Isolation 6

2.1.4 Durability 8

2.1.5 Transaction processing systems 8

2.2 Distributed systems 8

2.2.1 Distributed databases 9

2.3 Real-time systems 12

2.3.1 Real-time databases 13

2.3.2 Main-memory databases 14

2.4 The database recovery concept 14

2.4.1 Transaction recovery 16
2.4.2 Crash recovery 16
2.4.3 Media recovery 17
2.4.4 Impact of concurrency control on recovery 17
2.5 The DeeDS prototype 20

3 Problem description 22

3.1 Aims 22
3.2 Objectives 23
3.2.1 Perspectives 23
3.3 Literature survey 26
3.4 Properties of recovery mechanisms 26

4 Method and approach 28

4.1 Classification of recovery techniques 28
4.2 Presentation of results 30

5 Database recovery techniques 32

5.1 Logging 33
5.1.1 Write-ahead logging 33
5.1.2 Force-log-at-commit 33
5.1.3 Log entries 34
5.2 Checkpointing 35
5.2.1 Fuzzy checkpointing 36

5.3 Shadow paging 39
5.4 Main-memory recovery 40
5.4.1 Fuzzy checkpointing approaches 41
5.4.2 Incremental main-memory recovery 43
5.5 Distributed recovery 44
5.5.1 Distributed checkpointing 45
5.5.2 Distributed multi-level recovery 46
5.6 Diskless recovery 48
5.7 Analysis 51
5.7.1 Applicability in DeeDS 52

6 Replication management and consistency preservation 54

6.1 Immediate consistency 56
6.1.1 Two-phase commit 56
6.1.2 Non-blocking commit protocols 58
6.1.3 Other pessimistic approaches 61
6.1.4 Recovery in immediately consistent databases 63
6.2 Eventual consistency 66
6.2.1 Conflict detection 67
6.2.2 Conflict resolution 70
6.2.3 Other optimistic approaches 73
6.2.4 Recovery in eventually consistent databases 75
6.3 Analysis 76

6.3.1 A comparison to concurrency control 77
6.3.2 Recovery and consistency preservation 78
6.3.3 Applicability in DeeDS 82

7 Conclusions 83

7.1 Omissions and uninvestigated problems 83
7.2 Related work 84
7.3 Research directions and future work 85
7.3.1 General problems in distributed database recovery 86
7.3.2 Eventual consistency problems 87
7.3.3 Diskless recovery extensions 87
7.4 Contributions 88

8 Acknowledgments 90

Chapter 1

Introduction

The distributed real-time database field of research is interesting for several reasons. Not only is the demand for timely and dependable database systems high, the inherent complexity of distributed and concurrent systems coupled with the need for predictability in real-time systems also creates unique and interesting problems.

The project described herein addresses the problem of recovering the contents of a crashed node in a distributed database system. This project builds upon results from several projects previously conducted within the DeeDS project (Andler, Hansson, Eriksson, Mellin, Berndtsson & Eftring 1996) at University of Skövde, primarily Leifsson's work on distributed and diskless main-memory recovery (Leifsson 1999) and Lundström's work on bounded-delay replication and consistency preservation (Lundström 1997). We investigate the major areas which are covered by Leifsson's and Lundström's

work, contrasting their work with other, similar techniques.

We are particularly interested in recovery in eventually consistent databases, i.e., databases where availability, predictability and performance are improved by allowing replicas to deviate (see section 2.2.1 for a more formal definition of eventual consistency). Such databases are interesting in that they must be *convergent*, which means that if all transactions in the system are quiesced, all replicas must eventually become consistent. Eventually consistent databases also put restrictions on the applications that use the database, since they must be resilient to possibly stale data. We do, however, also cover other recovery mechanisms, e.g, recovery in immediately consistent databases, since it provides us with a useful frame of reference when we discuss recovery in eventually consistent systems.

The major effort in this work is put on literature surveys of consistency preservation techniques and recovery mechanisms. The surveys are used as a basis for a taxonomy and a state-of-the-art report of the fields. We complement the presentation of the surveys with analyses of how current recovery techniques can be applied to immediately and eventually consistent systems, respectively. We also discuss how different techniques for both consistency preservation and recovery can be applied within the DeeDS project.

In performing these surveys, we find that there are several related fields where the amount of research performed is very sparse. For example, there is very little work done on timely recovery of distributed databases. Similarly, consistency preservation in general and eventual consistency in particular is

rarely considered by distributed recovery mechanisms.

1.1 Dissertation overview

The rest of this document is partitioned as follows. Chapter 2 describes the necessary background and briefly introduces the relevant research areas, while chapter 3 defines the specific problem investigated in this dissertation. Chapter 4 outlines the approach taken to the literature survey and analysis described in chapters 5 and 6. Finally, chapter 7 contains our conclusions, including contributions, related work, and future work.

Chapter 2

Background

Recovery of distributed and main-memory resident databases encompasses several different research areas, especially when put in a real-time context. This chapter describes the fields of research touched upon in this document, and defines the most important terms within those fields.

2.1 The transaction concept

A common abstraction in database systems is the *transaction* concept. A transaction is a series of data object[1] manipulations in the form of read operations (which only sample the current value of a data object) and write operations (which update the value). A transaction consisting of only read operations is often referred to as a *query transaction*. If a transaction contains

[1] In this dissertation, we use the term *data object* in a very general sense. We do not concern ourselves with whether the data object in question is a row in a table, a physical page, or an object in an object-oriented database.

one or more write operations, it is called an *update transaction.*

By definition, a transaction should have four properties, commonly referred to as the *ACID properties.* In order, the ACID letters denote the need for transactions to be *atomic*, *consistent*, *isolated*, and *durable.* Each of these concepts is expanded below.

2.1.1 Atomicity

To the system, transactions should appear *atomic.* That is, a transaction should either complete all of its updates successfully, or none at all. No effects of partially executed transactions should ever be visible in the system. When a transaction has run successfully to completion, it *commits*, thereby making its updates permanent in the system. A transaction which cannot complete its execution is said to *abort.* An abort can be either voluntary (i.e., the transaction itself initiates the abort), or the result of a failure. To the rest of the system, it should appear as if an aborted transaction had never executed.

To support the atomicity property, the system must contain a transaction recovery mechanism which undoes updates made by an aborted transaction (see section 2.4.1).

2.1.2 Consistency

A database state is said to be consistent if it conforms to all the restrictions in the database specification (such as business rules, value ranges etc.). For a transaction to satisfy the consistency requirement, it must take the database from one consistent state to another. This requires careful programming of the transaction.

Note that the database may be in an inconsistent state at any time *during* the execution of a transaction. Once the transaction commits, however, any such inconsistencies must have been resolved.

2.1.3 Isolation

Transactions should run in *isolation*. This means that even though there may be several active transactions in the system at any given time, no transaction should be affected in any way by concurrently running transactions. To an individual transaction, it should appear that it is the sole transaction in the system. The example below illustrates the importance of isolation.

Assume that two transactions, T_1 and T_2, execute in parallel. T_1's task is to read the current balance of two bank accounts, A_1 and A_2, and return the total balance. T_2, on the other hand, performs a simple money transaction, moving $1000 from A_1 to A_2. Using o_n^m to denote operation m performed by transaction n, T_1's operations are

o_1^1: read balance(A_1)

o_1^2: read balance(A_2)

o_1^3: (compute and return total balance)

while T_2's operations are

o_2^1: read balance(A_1)

o_2^2: (check that A_1 holds more than \$1000, otherwise abort)

o_2^3: set balance(A_1) to its old value - \$1000

o_2^4: read balance(A_2)

o_2^5: set balance(A_2) to its old value + \$1000

If these transactions are executed serially (i.e., T_1 executes to completion before T_2 begins its execution, or the other way round), no problems arise. However, if the transactions are executed in parallel without any form of concurrency control, the operations may be performed in any, possibly interleaved, order. The execution below depicts an erroneous operation ordering:

$o_2^1\ o_2^2\ o_2^3\ o_1^1\ o_1^2\ o_1^3\ o_2^4\ o_2^5$ commit(T_1) commit(T_2)

Given this execution order, T_1 would return a total balance which is \$1000 off its mark. Since the problem exists only because T_1 is allowed to read T_2's uncommitted updates, the source of this problem is clearly lack of isolation.

Transaction isolation thus requires a mechanism for *concurrency control*, which prevents transactions from reading data objects updated by concur-

rent, uncommitted transactions. See section 2.4.4 for a more elaborate discussion on this.

2.1.4 Durability

Updates done by a committed transaction must be *durable*. That is, even if the system fails or crashes at any time succeeding the transaction's commit time, the transaction's updates should not be lost. Guaranteeing durability of transaction updates most often means writing logs of the updates to a stable medium, such as a hard drive or nonvolatile memory, and thus ensuring permanence of the updates.

2.1.5 Transaction processing systems

A *transaction processing system* (TPS) is a system which handles several of the issues associated with transactions. For example, the TPS should handle transaction synchronization, scheduling, and recovery. Of the ACID properties, a TPS should guarantee AID (atomicity, isolation and durability), while the responsibility of consistency preservation is often put on the programmer of the application that executes the transaction (Weihl 1994).

2.2 Distributed systems

A *distributed system* can be defined as

> "a system of multiple autonomous processing elements, cooperating in a common purpose or to achieve a common goal" (Burns & Wellings 1997).

The autonomous processing elements mentioned in this definition are often referred to as *nodes*, and are usually physically separated from one another.

The main reasons for distributing a system are increased availability and fault-tolerance through redundancy. Also, many embedded systems are distributed by nature, and it is possible that physical requirements (such as a restriction on the amount of cabling that can be used in a system) demand a distributed architecture.

2.2.1 Distributed databases

A *distributed database* is a database whose data is located at several of the nodes in a distributed system. It is often used to efficiently share data between nodes. Data in distributed databases may be *replicated*, which means that what the database user sees as a single, *logical* database object (El Abbadi, Skeen & Cristian 1985) may be stored in the database as several *physical* objects, usually located at different nodes. A distributed database where every data object is replicated on all nodes in the system is called a *fully replicated database*. As with all distributed systems, distributed databases have a high level of fault-tolerance (due to replication of data and/or processing elements) and also provide the option to disperse data and computing

resources to the locations where they are used (which leads to decreased communication overhead and increased availability).

Data replication

One of the advantages of having a distributed database is the high fault-tolerance and availability made possible by having replicated data, i.e., data stored on several of the nodes in the system. If data is replicated, it does not become unavailable when a node in the system crashes, since it can be retrieved from any of the nodes containing replicated copies of that data. If nodes can be assumed to be *fail-silent* (i.e., if a failing node does not transmit any data at all), and if data is replicated on t nodes, a distributed database can tolerate up to $t - 1$ node failures.

A distributed database which supports replication of data needs a *replication mechanism*, which ensures that updates of a data object on a node will be propagated to all other nodes where that data object is replicated. In certain real-time systems it may be necessary to be able to specify a bound on the time needed for replication of data. Such a *bounded-delay* replication mechanism is described in Lundström (1997).

Different replication mechanisms are often referred to as being either active or passive (see for example Wiesmann, Pedone & Schiper (1999)). In a system using *active* replication, all transactions that update a certain data object are sent to and executed on all nodes where that data object is replicated. Under the *passive* replication paradigm, update transactions are sent

to a particular node (called a *primary node*) which performs the update and, after the update has been performed, requests the update transaction to be executed on all other nodes that replicate the updated data. Different transactions may use different nodes as their primary node. Such a scheme is called *multi-primary replication*. A system where a single node serves as the primary copy for all concurrenctly executing transactions is said to use *primary-backup replication* (Wiesmann et al. 1999).

Database consistency

Of particular interest for this dissertation is the notion of consistency in distributed databases. As stated in section 2.1.2, a database state is said to be *consistent* if it does not violate any of the constraints in the database specification. In replicated databases, this implies that to uphold consistency, all views of a replicated (logical) database object must be identical, regardless of on which node the object is accessed.

We consider two types of consistency, immediate and eventual. Under *immediate consistency*, changes made by a transaction to a (logical) database object are not made visible to other transactions until all replicas of that object have been updated to reflect the changes. Having immediate consistency in a database guarantees that replica consistency is maintained at all times, but introduces overhead in the form of locking and synchronization protocols.

Under *eventual consistency*, updates to a physical object are made vis-

ible to local transactions (i.e., transactions running at the node where the object is located) as soon as the updating transaction commits, which it can do without contacting the other nodes in the system. The updates are then propagated to other nodes as soon as possible. If a conflicting update has been done to the object at a remote node, this is handled by a *conflict management protocol*. Such a protocol consists of a *conflict detection mechanism* and a *conflict resolution algorithm* that resolves conflicts according to a *conflict resolution policy*.

2.3 Real-time systems

Certain systems, particularly embedded and safety-critical systems, are often constrained not only in the functional domain, but also in the time domain. In such systems, the time at which an output is produced may be as important as, or even more important than, the correctness of the output. A system whose operations are subject to time constraints is referred to as a *real-time system*. There are many definitions of real-time systems, most of which emphasize the tendency of real-time systems to interact with their physical environment in a time-constrained manner. An example is the definition by Young (1982), who views a real-time system as:

> "any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period."

An important property of a real-time system is thus its *timeliness*. In order for a system to be timely, it must be predictable and sufficiently efficient. A system is *predictable* if there is an upper bound on its resource requirements, and *sufficiently efficient* if its peak load is *schedulable*, i.e., if all tasks in the system can be guaranteed to meet their deadlines at all times (Andler, Leifsson & Mellin 1999).

The timeliness requirements on a real-time system can be used to categorize the system as either hard or soft (see for example (Burns & Wellings 1997)). A *hard* real-time system is a system where missed deadlines cannot be accepted. Classic examples of such systems are nuclear power plants and airplane control systems, where failures may lead to human casualties and/or the destruction of valuable equipment. In a *soft* real-time system, on the other hand, a missed deadline only causes a decrease in the system's quality of service. As an illustration, unexpected delays when using an ATM might be acceptable, even though they may be annoying to the ATM user. If this is the case, the ATM is a soft real-time system.

2.3.1 Real-time databases

A *real-time database* is a database which is used in a real-time system. Thus, in order for it to be possible to guarantee deadline conformance in the system, the predictability requirements normally associated with such systems must be applied to the database as well. This means that all queries and updates on the database must be predictable and timely.

2.3.2 Main-memory databases

The choice of storage medium for a database is an important issue, especially in a time-constrained system. The most common database storage medium, ordinary disks, may not be ideal for real-time databases, since the long access times and the inherent unpredicability of many disk drivers may be too detrimental.

If all data is stored in main memory instead of on disks, the data access time is improved by several magnitudes. More importantly (from a real-time systems viewpoint) the access times in such a database can more easily be made predictable. A database which stores its data (either partly or entirely) in main memory is called a *main-memory database* (MMDB). If the entire database is kept in memory at all times (barring failures) it is referred to as a *main-memory resident database*. When discussing main-memory databases in this dissertation, we will consider only main-memory resident databases.

2.4 The database recovery concept

A database would be of little use if failures (either in the database subsystem, or in the system as a whole) would cause it to irrecoverably lose its contents, or make its data permanently inconsistent. Thus, any respectable database system must contain a *recovery mechanism* that allows it to recover from all or most failures that may occur. From a recovery viewpoint, failures can be separated into three categories.

Transaction failures occur whenever a transaction does not commit properly. This may be due to an *abort* operation being issued on behalf of the transaction, or because an error, such as a transaction deadlock, has occurred.

System failures or *crashes* result in the loss of all data currently resident in main memory. Thus, all buffered updates to a database – which have not yet been written to stable storage – are lost. In the case of fully main-memory resident databases, a system failure usually implies a media failure as well (see below). Also, a system failure leads to transaction failures for all uncommited transactions in the failing system.

A *media failure* marks the loss of all or part of the data on a stable storage device (such as a hard drive).

Failures of the different classes are not equally common. While transaction failures can be expected to occur several times per minute in a busy system, system failures often do not occur more than once or twice a week, and media failures only a few times every year (Haerder & Reuter 1996).

Although these failure classes are not independent, each of them requires a slightly different approach to recovery. Recovery from the different types of failures is discussed in the sections below.

2.4.1 Transaction recovery

A transaction which aborts may have updated any number of data objects during its execution preceding the abort. If its updates are left in the database, such a partial execution clearly violates the atomicity and isolation properties of the transaction. It may also endanger the consistency of the database, since database consistency is only guaranteed at the beginning and end of the transaction (see section 2.4.4 for examples of how database consistency can be affected by aborting transactions). Thus, all updates done by an aborted transaction must be undone during a transaction recovery process. Commonly, such undo processing is supported by logging all updates done by a transaction. Transaction logs can be constructed in a number of ways, a topic which is detailed in section 5.1.

2.4.2 Crash recovery

If the contents of the volatile memory are lost as an effect of a crash, transaction recovery must be performed for any transactions that were uncommitted at the time of crash. In addition, *redo* processing must be performed for committed transactions whose updates had not yet been written to stable storage at the time of crash.

As noted above, system failures are particularly severe in fully resident main-memory databases, since losing the main-memory contents means that the entire database is erased.

2.4.3 Media recovery

In the (hopefully) rare occasion of a media failure, the database is irrecoverably lost unless the system contains some form of redundancy, for example in the form of logs or archive copies of the database. Restoring a database from an archive copy is referred to as *archive recovery* by Haerder & Reuter (1996).

If a media failure occurs in a replicated database (e.g., if a node crashes in a main-memory resident, distributed database), the inherent redundancy in the system can be used for recovery purposes. This is somewhat similar to archive recovery. If a database replica is lost, it can be restored by copying the contents of another replica. Ensuring the consistency of the recovering replica while allowing transactions to run on the other replicas in the system (or even the recovering replica itself!) is an interesting research topic, and the problems associated with this are discussed in the section on distributed recovery (5.5).

2.4.4 Impact of concurrency control on recovery

In any database allowing parallel execution of transactions, *concurrency control* must be performed in order to maintain transaction isolation. If concurrent transactions are allowed to run unchecked, several problems may occur, as can be seen in section 2.1.3.

Other problems occur when transactions abort. Two famous such exam-

ples are the dirty read problem and the lost update problem (Hsu & Kumar 1996). The *dirty read problem* occurs when a transaction reads data which has been updated by another transaction, which subsequently aborts. If the first transaction is not somehow notified of the abortion, it will operate on dirty data, most likely resulting in a failure or in inconsistencies in the database.

When two transactions – one of which later aborts – update the same value, we may experience a *lost update problem*. As an illustration, assume that transaction T_1 consists of the following operations

o_1^1: read value of object a

o_1^2: set value of object a to its old value + 100

and that transaction T_2's operations are

o_2^1: read value of object a

o_2^2: set value of object a to its old value + 500

A possible execution scenario is (assuming that a is initially 0):

o_1^1	(a=0)
o_1^2	(a=100)
o_2^1	(a=100)
o_2^2	(a=600)
commit(T_2)	(a=600)
abort(T_1)	(a=?)

When T_1 aborts, its operations should be rolled back. Depending on the *transaction recovery strategy* of the database system, a could either be set to the value that T_1 read before making its update (0), or a *compensating* operation could be performed, decreasing a's value by 100 and leaving it at 500. If the former approach is used, executions such as the one above must be disallowed by the concurrency control mechanism, since setting a's value to 0 would mean that T_2's update is lost. On the other hand, the latter strategy is more complex, since it requires each transaction to have a compensating transaction that can be executed in case of an abort.

This example clearly illustrates how the degree of concurrency control in a system must influence the choice of transaction recovery strategy, and vice versa. Allowing a high amount of concurrency generally means that a more sophisticated (and probably more resource-demanding) transaction recovery mechanism must be used, while having a simple recovery strategy places constraints on the level of concurrency that can be allowed. Ultimately, this trade-off should be made based on the system as a whole. For example, if transaction aborts are expected to be common, the transaction recovery process should be as efficient as possible. If, on the other hand, the system experiences a high transaction arrival rate and demands high transaction throughput, a complex recovery process can be justified by the high amount of concurrency this allows during normal processing.

2.5 The DeeDS prototype

DeeDS (Andler et al. 1996) is a distributed, active, real-time database system prototype currently under development by the distributed real-time systems research group at the computer science department, university of Skövde. Since DeeDS is our testbed, and since applicability to the DeeDS prototype is a major assessment criterion for our results in this and future projects, DeeDS's design determines the focus of our work, and a brief description of its design philosophy is in order.

One of the main goals of DeeDS is to provide a distributed database suitable for systems with hard real-time constraints. Care has thus been taken to eliminate, or at least restrict the impact of, the sources of unpredictability normally associated with database systems. Because of this, DeeDS uses a diskless architecture, instead placing the entire database in main memory. Furthermore, the critical system services, such as event monitoring and scheduling, run on a separate, dedicated processor to ensure that these do not interfere with processing of application transactions. To reduce communication overhead during transaction processing, and to increase availability of data, the database is also fully replicated at each node.

To provide data replication without endangering the timeliness of the system, and to eliminate the risk for blocking associated with protocols such as the two-phase commit, DeeDS provides only eventual consistency guarantees (see section 2.2.1). To support hard real-time constraints, the time

between an update transaction committing at one node and its updates being reflected at all nodes in the database must be predictable. Thus, DeeDS uses bounded-delay replication (see section 2.2.1).

Chapter 3

Problem description

This chapter contains the project aims, and the objectives that have been identified as necessary to fulfill these aims. The objects are detailed with respect to a number of perspectives, which are described in section 3.2.1.

3.1 Aims

The primary aim of the project is to analyze, and provide a thorough view of, the state of the art in distributed database recovery, especially for eventually consistent databases. Our second aim is to contrast different recovery techniques in order to find advantages and disadvantages of these techniques, and our third to identify aspects of the recovery problem that have not yet been explored. Finally, our fourth aim is to identify whether or not "traditional" approaches to database recovery may be applied to (or modified to

fit) recovery in diskless main-memory database systems under an eventual consistency scheme. In short, we hope to identify problems within the field which are interesting enough to warrant future research.

3.2 Objectives

In order for our view of the state of the art in distributed main-memory database recovery to be useful, and thus to fulfil our primary aim, we need to cover the topic from several perspectives. The perspectives that we have identified are listed in section 3.2.1. Our second and third aims require an extensive literature survey within the database recovery field (see section 3.3). To fulfil our fourth aim, we need to find properties of recovery mechanisms that make them more or less suitable for use in a distributed database system under the eventual consistency assumption. This objective is discussed in section 3.4.

3.2.1 Perspectives

Since we are interested in distributed and main-memory resident databases, it is natural to cover existing work on distributed database recovery, as well as recovery in main-memory databases. Since we are particularly interested in real-time (and thus time-constrained) systems, we also want to consider the timeliness of different recovery protocols, and their effect on the timeliness of the rest of the system. Finally, we argue that the replication policy

in a distributed database, which can either support immediate or eventual (delayed) consistency, can have an impact on recovery in the database.

Distributed database recovery

Recovery in a distributed database is made more complex by the fact that the global database state must still be consistent after a node has crashed and recovered. During node recovery, the other (functional) nodes in the system are often required to provide a similar level of service as they do in a fully operational scenario. Updates done at these nodes may affect the state of the recovering node. The recovery process must not cause the recovering node to miss any information about such updates.

Another issue is how checkpoints and logs are stored. Either the storage is centralized (i.e., logs and checkpoints are sent to and maintained at a single node) or each node performs its own logging and checkpointing. Hybrids of these two techniques are also possible. Another technique is for a node to log its actions by sending them to another node in the system (e.g., the *buddy system* (Leifsson 1999).

Main-memory database recovery

If a main-memory resident database crashes (and the main memory is volatile) the entire database is lost and must be reloaded into main memory from some recovery source. This is analogous to a media failure in conventional databases. Important issues in main-memory recovery include logging mecha-

nisms, checkpointing policies and whether or not the approach is incremental (i.e., whether or not the node can resume partial service before the recovery process has been entirely completed). Usually, some kind of stable storage is used as recovery source. A distributed system may use other nodes in the system for this purpose, provided that the database is replicated, and that all nodes do not crash simultaneously. If non-recovering nodes must continue their operation during such recovery, care must also be taken to ensure that database consistency is preserved throughout the recovery process (see section on consistency preservation below).

Timeliness of recovery

If the database is part of, or used by, a real-time system, timeliness of database operations and recovery processing is paramount. Thus, for a recovery mechanism to be viable in such a system it must be predictable and sufficiently efficient, both on a local (recovering node) and global non-recovering nodes) level. We use timeliness as an orthogonal perspective – i.e., for a given recovery method, we discuss whether or not it is inherently timely, and if not, if it can be made timely in a simple way.

Effect of consistency preservation on recovery

As mentioned in section 2.2.1, the replication of data in a distributed database adds complexity to transaction synchronization in the database. We need to ensure that all (physical) replicas of an object are seen as a single, logical

object in the database. This means that all nodes should have a consistent view of the database object. Different ways of ensuring distributed database consistency and how they interact with recovery is described in chapter 6.

3.3 Literature survey

The results of our literature survey are presented in chapters 5 and 6. The purpose of the survey is to obtain a view of the state of the art, and to determine which of the investigated recovery and replication methods are suitable for use (with or without modifications)in real-time recovery of distributed main-memory databases. We are also interested in the relationship between consistency preservation and database recovery. We have tried to identify omissions within the relevant fields, such as open research directions and uninvestigated problems. We present these along with recommendations for future work and possible directions for further research.

3.4 Properties of recovery mechanisms

Since we want to analyze how database recovery interacts with consistency preservation in general, and with eventual consistency in particular, we must identify the properties of a recovery mechanism that affect its interaction with the consistency preservation mechanism, and that thus determine its suitability for eventually consistent databases. Such properties are listed in

the recovery technique classification presented in section 4.1 and are discussed in chapters 5 and 6.

Chapter 4

Method and approach

In this chapter, we present our approach to a classification of the papers and methods that have been covered by our literature survey. We also outline how the rest of this dissertation is structured, and how our perspectives are covered by the results presented in chapters 5 and 6.

4.1 Classification of recovery techniques

As should be apparent to the observant reader by now, database recovery is a complex area, where many different issues interact. We have tried to identify those attributes of recovery mechanisms (and the database systems in which they operate) that have a major influence on how recovery is performed. A classification of recovery techniques can then be obtained by assigning values to these attributes for any given recovery technique. For some values of the

attributes, further attributes can be identified, for which a value must also be assigned. Figure 4.1 on page 31 shows the hierarchy of attributes and corresponding values that we have identified. In the figure, attribute names are shown in bold face, while attribute values are in plain text.

As the figure illustrates, we believe that there exist four major influences on a recovery technique. The first, of course, is the recovery algorithm itself, which is our primary concern. The other three (the concurrency control scheme, the replication management and consistency preservation mechanism, and the database system properties) concern the system model in which the algorithm is assumed to operate. Of the three system model attributes, we are primarily interested in the replication and consistency aspect.

As we have already discussed (see section 2.4.4), the concurrency control scheme used in a system has a major impact on recovery. This relationship, however, has already been thoroughly analysed by for example Weihl (1989). Thus, we focus instead on the relationship between consistency preservation and database recovery.

Finally, the database system itself has several properties that dictate what recovery techniques may be used in the system. The timeliness requirements on the database, the choice of storage medium, and the level of distribution all influence the choice of recovery mechanism.

4.2 Presentation of results

The following two chapters contain the results of our survey of recovery techniques. Chapter 5 presents the recovery techniques themselves, defining important recovery concepts and describing past and contemporary research tracks within the field. Chapter 6 contains a discussion on different approaches to replication management and consistency preservation in a distributed database, and how they affect recovery. We are particularly interested in the interdependencies between eventual consistency and recovery.

Two of our perspectives (as listed in section 3.2.1) – main-memory recovery and distributed recovery – are covered in chapter 5, while a third – the effect of consistency preservation on recovery – is discussed thoroughly in chapter 6. This leaves the fourth perspective – timeliness. The timeliness of recovery and replication methods is not discussed separately – we have instead chosen to consider the timeliness of each method individually as they are presented in chapters 5 and 6.

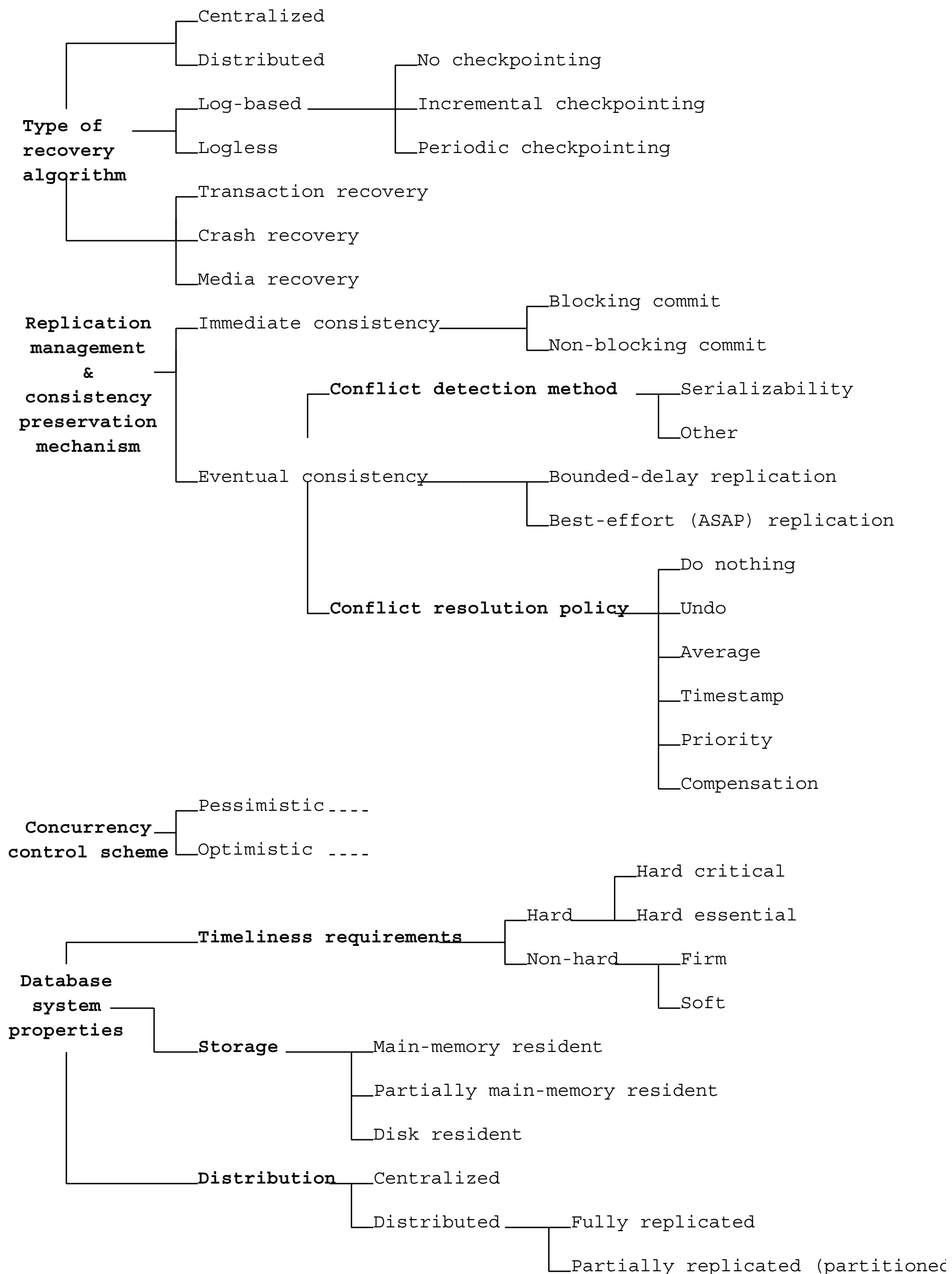

Figure 4.1: Classification of recovery techniques by attributes

Chapter 5

Database recovery techniques

In section 2.4 we briefly looked at how recovery from different types of failure (transaction-, crash-, and media failure) is done on a conceptual level. In this chapter, we do a more thorough presentation of existing recovery techniques, detailing those that we have found to be the most influential to the field in general, and those that we have judged most interesting for the DeeDS project. Sections 5.4 through 5.6 present these techniques, and section 5.7 contains our analysis. We begin, however, with a discussion of logging and checkpointing, since these are important concepts in many of the recovery techniques presented in this chapter.

5.1 Logging

A common approach for supporting all types of recovery is to keep a *log* of all updates done in the database. In order to support recovery from crash- and media failures, the log must be kept on stable storage, and needs to be unaffected by media failures (i.e., it should not be stored on the same medium as the database). How the log is constructed, and how log maintenance is performed varies. Below, some important aspects of logging are discussed.

5.1.1 Write-ahead logging

The purpose of the *write-ahead log* principle (WAL) (Gray & Reuter 1993) is to support transaction recovery by allowing transaction undo to be performed. Following the WAL principle means forcing updates to be logged before they are installed in the database. This way, undo data for all updates is guaranteed to be present should the transaction later abort.

5.1.2 Force-log-at-commit

Similar to the way that the WAL principle ensures atomicity of an update transaction by allowing all updates to be undone if an abort occurs, the *force-log-at-commit* rule (Gray & Reuter 1993) is concerned with upholding transaction durability. This is done by prohibiting a transaction from committing before all log entries concerning that transaction's updates have been flushed to stable storage.

If a transaction were allowed to commit without ensuring stable log entries for all its updates, a system failure would mean that all updates to main-memory database objects without corresponding stable log records would be irrecoverably lost. Thus, the force-log-at-commit rule must be followed to ensure durability of update transactions in a log-based TPS.

5.1.3 Log entries

An update record in a log is known as a *log entry*, and can take several forms. A common approach is to store a *before image* (BFIM) and/or an *after image* (AFIM) of the affected data object in each log entry (Hsu & Kumar 1996). The BFIM is a copy of the unupdated data object which can be reinstalled in the database during the transaction recovery process should the updating transaction abort. Similarly, the AFIM of an object can be used to redo a buffered update of a committed transaction that was lost in, for example, a crash. BFIM/AFIM logging is often referred to as *physical logging*. The counterpart to physical logging is *logical logging*. A logical log entry consists of a high-level description of the update, rather than physical before- and/or after images. Logical logging is particularly space-efficient for redo processing, since the space required for storing an AFIM in most cases exceeds the space needed for a logical log entry. For undo processing, logical logging can theoretically be exploited to allow a higher amount of concurrency than is possible with physical logging (Jagadish, Silberschatz & Sudarshan 1993). This is possible since logical log entries can be undone

by performing a compensating update. The only way to undo an operation whose log entry consists of a BFIM is to overwrite the updated data object with the BFIM. Since this implies overwriting updates performed on that object by other transactions, a stricter concurrency control policy must be used with physical logging.

A disadvantage of logical logging is that logical operations are normally not idempotent. For example, if the (logical) operation "add $500 to account A" is performed twice during recovery (as part of redo processing), an excess $500 is added to the account, putting the database in an inconsistent state. This must be considered in designing the recovery protocol. Physical logging avoids this problem, since multiple installations of an AFIM does not lead to inconsistencies.

Regardless of whether physical or logical logging is used, each log entry must contain a *transaction identifier* (TID). The identifier is used to decide whether the update was done by a committed or aborted transaction. Log entries must also be made for *begin transaction*, *transaction commit*, and *transaction abort*.

5.2 Checkpointing

To decrease the time needed for log processing during recovery, a log is often supplemented by periodic *checkpoints* of the entire database on stable storage. Whenever the database needs to recover its contents the previous checkpoint

is read, and the updates in the log which are not reflected in that checkpoint are applied to the checkpoint data. Checkpointing can thus be seen as a way of optimizing the log, reducing the amount of processing and storage needed for recovery. Checkpoints are recorded in the log by *begin checkpoint* (BC) and possibly *end checkpoint* (EC) entries. Obviously, if a new checkpoint is started as soon as the previous checkpoint is finished, only BC entries are needed.

5.2.1 Fuzzy checkpointing

Traditional checkpointing normally requires the system to be in a quiescent state (i.e., there must not be any update transactions running) while taking a checkpoint, since allowing updates during the checkpointing process might render the checkpoint inconsistent. If the database is large, such an approach to checkpointing may lead to extended periods of transaction inactivity. In many database systems, particularly in real-time database systems, such inactivity periods are unacceptable.

An interesting alternative to traditional checkpointing is called *fuzzy checkpointing*. While different fuzzy checkpointing approaches vary in their implementation, they are all similar in that they allow transactions to run while the checkpoint is being taken. Thus, transaction throughput is enhanced at the price of getting possibly inconsistent checkpoints (something which must be compensated for in the recovery process). If the database is large, and if continuous operation is required, this is, however, a necessary

trade-off.

Fuzzy checkpointing for main-memory databases is discussed in section 5.4.

Logging after writing

As noted by Li & Eich (1993), using fuzzy checkpointing in conjunction with physical logging of AFIMs (see section 5.1.3) may lead to problems if restrictions are not put on when the AFIM may be written to the log. For example, in the scenario depicted in figure 5.1, the AFIM for update u is written to the log at a time prior to the first begin checkpoint-entry (BC1), while the actual update to the database is done *after* the checkpointing procedure has begun. After the checkpoint is completed (marked by BC2 in the figure – note that BC2 implies a end checkpoint-entry for checkpoint 1), the system crashes.

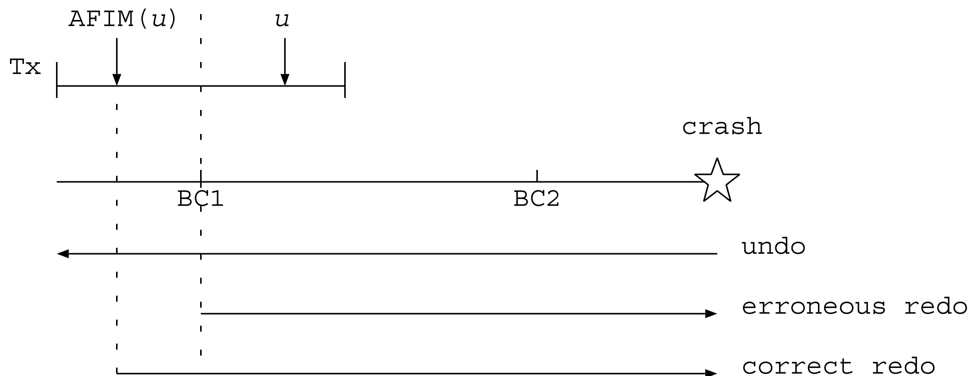

Figure 5.1: Redo without LAW (Li & Eich 1993)

Now, intuition tells us that to perform crash recovery, we should read the last completed checkpoint – in this case, checkpoint 1 – undo all operations of transactions which were uncommitted at the time of crash, and then redo all updates of committed transactions by applying all relevant log entries which *postdate* the BC entry for that checkpoint. However, in this scenario, this is not enough, since this would mean that u would be lost! Thus, we must process a certain amount of log entries that precede the BC.

The problem is: how much of the log history do we need to consider? To avoid this problem, Li & Eich (1993) suggest using the *Logging After Writing* (LAW) protocol. Under LAW, AFIMs of updates may only be written to the log after the update has been installed in the database (see figure 5.2). This protocol for correct redo complements the WAL protocol (described in section 5.1.1) that most systems use for writing BFIMs to support undo.

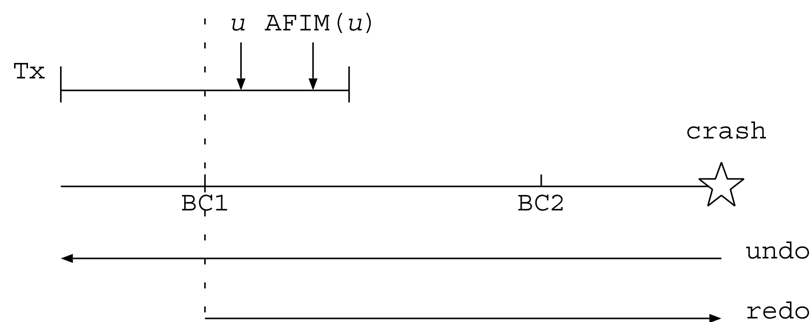

Figure 5.2: Redo with LAW (Li & Eich 1993)

5.3 Shadow paging

Shadow paging is an update technique that can be used as an alternative to (or in combination with) logging (Ylönen 1995). Using shadow paging, the update (and lock) granularity is database pages. Every data object in the database is associated with one or more *logical* database pages. A *page table* is used to map logical pages to *physical* pages.

Whenever a transaction wishes to update a data object, copies of all pages associated with that data object are created, and the transaction receives write locks on those pages. All updates are then performed on the page copies (the *shadow pages*). When and if the transaction commits, the page table is modified to map the relevant logical pages to the freshly updated shadow pages, which thus cease to be shadow pages and become *current pages.*

Note that since there is no log, the updated pages must be flushed to stable storage at commit in order to guarantee durability of the updates (see section 5.1.2). Transaction undo, on the other hand, is very simple using shadow paging. If a transaction aborts, its write locks are released and the relevant shadow pages are discarded. No updates are made to the page table.

In a fully resident main-memory database, shadow paging could also be done the other way round, with transactions updating the current page rather than the shadow page. Using this method, the page table is only updated in case of an abort, rather than at commit (Hsu & Kumar 1996).

5.4 Main-memory recovery

Due to the volatile nature of memory, system failures in a main-memory database normally imply a media failure as well. Thus, media recovery can be assumed to be more common in an MMDB system than in a system where the database is kept on stable storage. For this reason, it is important that the media recovery process in an MMDB is as efficient as possible. Traditionally, even main-memory databases use disks or some other non-volatile storage for logging and checkpointing purposes (see sections 5.1 and 5.2). However, since having a fully resident main-memory database means that no I/O is required to access the database itself, the I/O associated with a disk-resident log may become a major bottleneck in the system, negating some of the advantages of having an MMDB architecture.

That the I/O associated with logging and checkpointing has severe implications for transaction processing in an MMDB system is noted by Eich (1987). In this paper, it is examined how four properties of MMDBs influence the average transaction recovery time and transaction throughput of the dabatase system. The properties are (*i*) availability of stable memory, (*ii*) use of specialized logging hardware, (*iii*) use of dedicated checkpointing hardware, and (*iv*) choice of commit policy (immediate commit or pre-commit/commit). Of these, the availability of stable memory is found to have the most impact on transaction throughput. Alongside the use or non-use of checkpointing hardware, it is also the property that affects transaction

response time the most. However, it is found that increasing the amount of stable storage beyond what is needed to store the log does not yield further improvement to response times.

5.4.1 Fuzzy checkpointing approaches

In a series of papers by Dunham (formerly Eich) et. al. (Li, Eich, Joseph, Gulzar, Corti, Nascimento & Peltier 1995), (Lin & Dunham 1995), (Lin & Dunham 1996), (Dunham, Lin & Li 1996), fuzzy checkpointing alternatives tailored for main-memory databases are discussed. In particular, two techniques for checkpointing *partitioned* or *segmented* databases are presented. These techniques are discussed in the sections below.

Dynamic segmented fuzzy checkpointing

The first technique, called *dynamic segmented fuzzy checkpointing* (DSFC) divides the main memory into n segments. Checkpoints of these segments (which are *dynamic* in the sense that the memory segmentation can be changed during run-time to adapt to changing conditions) are taken in a round-robin fashion. The checkpoint is seen as a sliding window, using the n last segment checkpoints to construct a checkpoint of the complete database. Figure 5.3 illustrates how the mean time between complete checkpoints is reduced by using n=4.

This technique improves upon traditional checkpointing in several ways. Since it is a fuzzy technique, transactions need not be quiesced before the

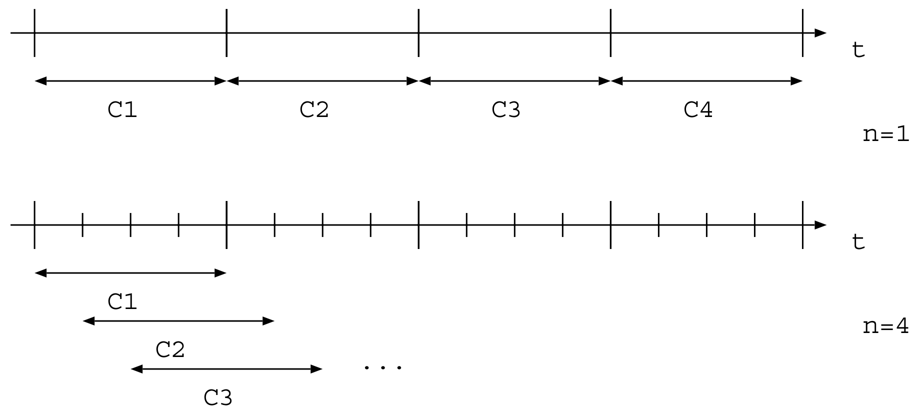

Figure 5.3: Dynamic segmented fuzzy checkpointing

checkpoint is taken, thus improving transaction throughput (see section 5.2.1). In addition, since checkpoints are updated incrementally, the average time needed for recovery is improved, since the average amount of log entries that need to be processed during recovery is reduced.

Partition checkpointing

A derivative of DSFC, *partition checkpointing* (PC) splits the database into n segments just like its sister technique. However, checkpoints of the segments are not taken sequentially, or even with an equal frequency. Instead, the checkpointing frequency of a segment is proportional to the update frequency of that particular segment. This means that commonly updated segments (often referred to as *hot spots*) are checkpointed more regularly than segments with a low update frequency. This implies that complete checkpoints are constructed more seldom than they would when using DSFC, but this is

compensated for by getting more accurate partition checkpoints. Thus, log processing at recovery can be further reduced since most log records should concern the more frequently checkpointed segments – which in turn should have a relatively 'fresh' image in the checkpoint.

DSFC and PC are both suited for systems where failures occur relatively often. Both techniques incur a slight overhead over normal transaction processing[1], but if recovery needs to be performed often, this is offset by the performance gain in the recovery process compared to recovery a system with a 'traditional' checkpointing scheme. If the database contains easily identified hot spots, the partition checkpointing technique should outperform the DSFC. However, if database accesses are more or less evenly distributed over database segments, the algorithms should perform equally well, suggesting that the more simple DSFC technique is used.

5.4.2 Incremental main-memory recovery

Gruenwald, Huang, Peltier, Lin & Dunham (1996) discuss *incremental* recovery of a main-memory database. In an incremental recovery scheme, transaction processing is allowed to start before the entire database has been reloaded into main memory. When the database hot spots have been reloaded, most transactions should be able to run to completion in spite of the incomplete state of the memory-resident database.

[1] This overhead can, however, be reduced significantly by having a separate processor perform the checkpointing and logging.

This, of course, requires that hot spots can be identified, and that the database can be partitioned in such a way that incremental recovery can be performed (cf. section 5.4.1). In Levy & Silberschatz (1992), an incremental recovery technique is presented that reloads database pages in an order depending on their access history. This means that the most frequently accessed pages are reloaded first during the recovery process.

While incremental recovery can be used in most systems to improve transaction throughput, its applicability in distributed (fully replicated) real-time databases is limited. In such a database, data objects can be accessed on any node in the system, meaning that we do not depend on the recovering node to provide crucial data. Since the real-time constraints on the system dictate that we allow for the worst case scenario when designing the system, we must ensure that the transaction deadlines can be met even if data needs to be accessed on a remote node. Thus, since the system is designed to function properly in the worst case, we gain little from having a node provide partial service.

5.5 Distributed recovery

Distributed recovery differs from centralized recovery in that there is no natural recovery coordinator. Another problem is that a distributed database has a *global state* as well as a *local state* at each node. These properties of a distributed database mean that recovery becomes more complicated. At the

same time, a distributed database often contains inherent redundancy in the form of replicated objects, a fact which can be exploited for recovery. In the following sections, we discuss some approaches to distributed recovery.

5.5.1 Distributed checkpointing

A distributed database offers several ways to perform checkpointing. For example, either checkpoints of the global database state could be taken, for example by some central checkpoint coordinator, or the nodes could take independent checkpoints of their local state. Lin (1997) presents a taxonomy for distributed checkpointing, and also briefly criticizes the current trend to take transaction-consistent global checkpoints in distributed databases.

The taxonomy is based on three dimensions; level of synchronization, level of interference and level of consistency. The level of synchronization measures the 'tightness' of the checkpointing, i.e., how much autonomy the nodes have in deciding when to take a checkpoint. The level of interference shows to what extent the normal processing of the system is affected by the checkpointing (e.g., a checkpointing approach which requires the system to be in a globally quiescent state has a high level of interference). Finally, the level of consistency determines the 'fuzziness' of the approach both on a global and local level.

The author's case against transaction-consistent global checkpoints is based on a number of observations. Among these are the facts that (*i*) global recovery is seldom needed, (*ii*) forcing a node to recover (to restore a global

checkpoint) violates node autonomy, and (*iii*) global checkpointing does not scale well to large systems.

Lin (1997) also presents a new recovery protocol, *Loosely Synchronized Local Fuzzy Checkpointing* (LSLFC), which, as the name implies, takes fuzzy checkpoints (see section 5.2.1) of local nodes' database states. During checkpointing, messages are exchanged between the checkpointing node and the checkpoint coordinator to determine which transactions should be included in the checkpoint. This improves recovery time (since only necessary transactions are included) and enables the construction of a transaction-consistent global state at recovery time.

It should be noted that the eventual consistency paradigm and the diskless recovery protocol in DeeDS (Leifsson 1999) (and, thus, DeeDS) aim for roughly the same goals as those described by Lin (1997). Both de-emphasize global properties of the system, while upholding strong node autonomy. Scalability is also supported well by both eventual consistency and diskless recovery.

5.5.2 Distributed multi-level recovery

An interesting approach to recovery is the multi-level recovery technique presented by Bohannon, Parker, Rastogi, Seshadri, Silberschatz & Sudarshan (1996). In their method, an operation hierarchy is used to allow commits and aborts to be done at different levels. Transactions are at the highest level (L_n), and individual database updates are at level L_0. An operation at level

L_i can consist of any number of operations at level L_{i-1}. Synchronization is enforced by locking, and when an operation at level L_i pre-commits, all locks held by the operations at level L_{i-1} are released. It is emphasized in the method that entire pages should not be locked. Instead, *regions* are used as the lock granularity unit. A region could be any database object, but several regions could be contained in a page, thus allowing concurrent updates of several regions within the same page.

Logging is performed using logical log entries (see section 5.1.3). Low-level physical entries are replaced by higher-level logical entries which contain undo information at the operation level. Also, *ping-pong checkpointing* is used to get consistent checkpoints without resorting to the WAL (see section 5.1.1) protocol (WAL requires locks to be held on checkpointed pages, something this protocol aims to avoid). Ping-pong checkpointing uses two checkpoints, which are overwritten in an alternating fashion. Since we do not use WAL, the checkpointed pages may contain updates whose log entries have not yet been flushed to disk. However, once the checkpoint is completed, sufficient undo and redo information is written out to make the checkpoint consistent. Should the system crash while we are taking the checkpoint, or while we are writing the log entries, we can always fall back on the other checkpoint (which is older, but consistent).

Two distributed implementations of this protocol are discussed by Bohannon et al. (1996). The first one assumes a client/server architecture, where checkpointing and recovery are performed by the server. The clients have

copies of the database, and have their own local lock managers. A global lock manager at the server keeps track of the cached locks at the clients.

The second version uses a shared disk architecture. A global lock manager (which may or may not be distributed) is used to maintain synchronization. Since there is no server to merge updates to a page performed by different nodes, the log entries for all updates must be broadcast. When a checkpoint is to be taken, one of the nodes is selected to be the *checkpointing coordinator*. The coordinator checkpoints its own database image along with each site's active transaction table.

5.6 Diskless recovery

Another approach to database recovery (in a distributed system) is a disk- and logless recovery mechanism, which instead uses replicated data at other nodes as the recovery source. Such an approach is described by Leifsson (1999). Certain systems operating in a hostile environments may benefit greatly if the need for disks can be eliminated. Cost, weight and space restrictions on a system may also make it preferable to use a diskless architecture. Leifsson's approach makes five assumptions about the main memory resident, fully replicated database system (Andler et al. 1999):

(*i*) *Immediate local consistency and eventual global consistency*: Transactions are allowed to commit on a local node without considering the state of any

replicas of the data objects updated by the transaction. The local updates are then propagated to the other nodes in the system to ensure eventual consistency (see section 6.2).

(*ii*) *Nodes can detect and resolve conflicts in replicated updates*: For an eventual consistency scheme to work, nodes must be able to detect scenarios where several transactions have updated the same logical data object differently. There must also be a policy for resolving such consistency conflicts.

(*iii*) *Nodes are fail-silent*: When a node fails, it should cease all transmissions. This means that we do not need to consider byzantine failures.

(*iv*) *Shadow-paging is used for database updates*: Shadow pages are used instead of log records to ensure atomicity and durability of transactions. This way, pages can never be partially updated.

(*v*) *No other nodes fail while a node is recovering (single failure assumption)*: To ensure the durability of locally committed transactions, all updates are sent to another node in the system (the so-called *buddy node* of the local node) before the transaction commits. This way, the updates will not be lost even if the node where the updates were made subsequently crashes. However, since durability will be violated if both a node and its buddy go down at the same time, we must assume only single failures in the system.

Two of the nodes in the system are involved in the recovery process – the recovering node (also called the *recovery target*) and the *recovery source*, which is the node that supplies its current database state to the recovery target during the recovery process. The recovery source should be chosen in a somewhat intelligent way to prevent nodes with an already high workload to become recovery sources. For example, a voting scheme can be used.

The process can be divided into three phases (see figure 5.4). During the first phase, the recovery source sends its database state to the recovery target. This can be seen as the recovery target reading a fuzzy checkpoint of the database. During the second phase, the recovery source transmits a log of all updates that it has performed concurrently with sending the checkpoint to the recovery target. When the recovery target has applied this log to its freshly received checkpoint, it has a locally consistent copy of the database and can thus go online. During the second and third phases, the recovery source forwards all replicated updates that it receives to the recovery target. During the third phase, all such replicated updates are applied to the recovered data, and the phase ends when it can be guaranteed that the recovery target will receive all replicated updates in the system.

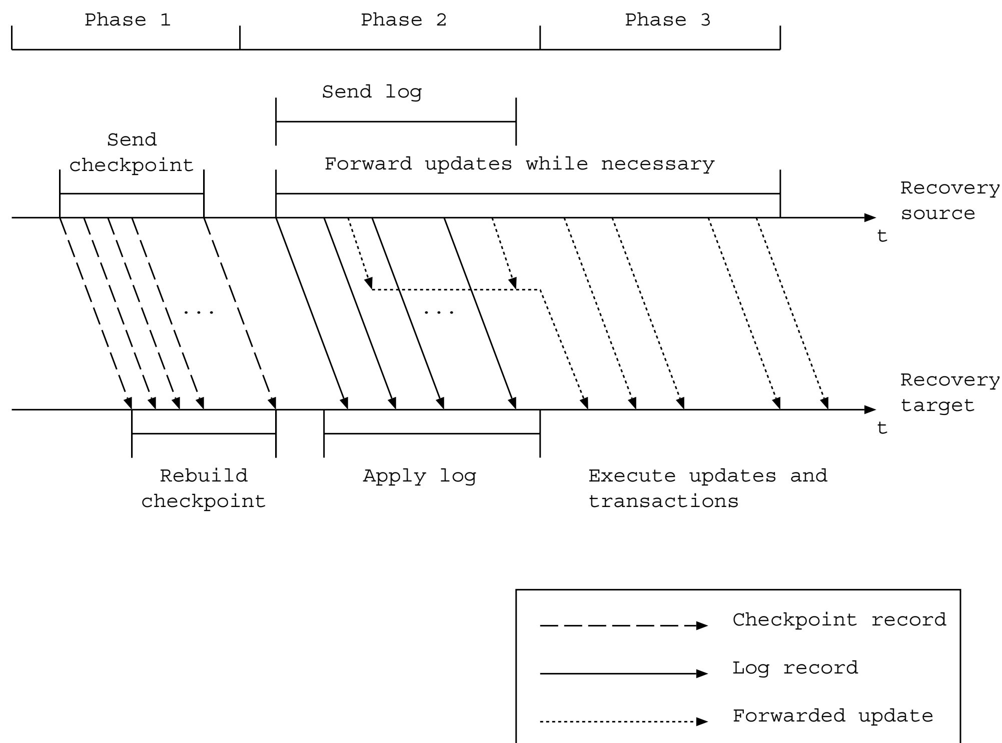

Figure 5.4: Phases in diskless recovery (Andler et al. 1999)

5.7 Analysis

Most of the recovery techniques discussed in this chapter are related to one another. Distributed checkpointing is just an implementation of regular logging/checkpointing in a distributed system – although the distributed nature of the system presents the system architect with interesting dilemmas (e.g., should the checkpoints be global or local?). Diskless recovery assumes a main-memory database, which means that it is really a main memory recovery technique. Furthermore, the sending of a recovery source's state to

a recovery target while transactions are allowed to continue updating the recovery source is analogous to reading a fuzzy checkpoint.

Multi-level recovery is interesting, as it seems to be a refinement of traditional main-memory recovery. However, both the ping-pong checkpointing scheme and the maintenance of different levels of locks suggest that it is more complex and resource-demanding than regular main-memory recovery schemes. It differs from the other techniques in that it does not require adherance to the WAL rule.

Shadow paging is probably the technique which has the least in common with the other techniques, since it does not require a log of any kind. With this said, diskless recovery does assume that shadow paging is used to ensure that no partially updated database pages are sent to a recovery target.

5.7.1 Applicability in DeeDS

DeeDS, being a real-time database, requires any recovery technique used in the system to be timely. With this in mind, regular (non-fuzzy) checkpointing is unsuitable, since it needs to quiesce the system to take a checkpoint – causing unpredictability by halting transaction executions. Fuzzy checkpointing improves transaction throughput, but still assumes that checkpoints and logs are stored on a stable medium, which makes it incompatible with DeeDS's diskless architecture. The two versions of multi-level recovery also require stable storage, and the client-server version in particular is unsuitable for a safety-critical system, since it introduces a single point of failure.

Diskless recovery, which also as discussed assumes shadow-paging, is tailor-made for the DeeDS architecture, and is thus naturally the most suitable of the techniques presented in this chapter. It is, however, not thoroughly examined how it interacts with the eventual consistency paradigm, which is another of the underlying assumptions in DeeDS. For further discussion on eventual consistency, see section 6.2.

Chapter 6

Replication management and consistency preservation

In this chapter, we discuss different methods for maintaining replica consistency in a distributed database, and their influence on recovery. To reiterate, *replica consistency* means that each physical replica of a logical database object should provide the same view of the logical object (i.e., should contain the same value).

There are two approaches to maintaining replica consistency. The methods that aim for *immediate consistency* restrain transactions in that they do not allow an update transaction to commit if the database cannot be made consistent at the time of commit (i.e., it should be possible to update enough physical objects for the view of the logical object to be consistent to

all users). *Eventual consistency* schemes sacrifice immediate consistency for increased availability, performace and/or increased predictability. An eventually consistent database allows inconsistencies to exist in the database, but requires that the system eventually becomes consistent if all transactions are quiesced.

For example, assume that a network of banks have a fully replicated database of all their customers' accounts. Using an immediate consistency scheme in such a database would most likely lead to serious delays for, e.g., a customer performing a transaction at an ATM, since all the replicas of his or her account would have to participate when the transaction commits. Using an eventual consistency scheme, the transaction would be allowed to commit on the local node only, and any conflicts that were later detected would resolved in accordance with some conflict resolution policy (such as billing customers for overtaxation of their accounts).

In accordance with our perspectives, we consider recovery in both immediately consistent and eventually consistent databases (sections 6.1.4 and 6.2.4). In our analysis, we contrast recovery in the two types of systems and identify issues that influence the design of a recovery mechanism in the different systems.

6.1 Immediate consistency

Immediate consistency schemes try to uphold one-copy serializability at all costs. A set of transactions on physical objects in the database is said to be *one-copy serializable* if they could be made serializable in a database with only a single copy of each data object (i.e., a non-distributed database where each logical database object is represented by only one physical object). One-copy serializability is a sufficient, but not necessary condition for consistency. Below, we present some immediate consistency schemes.

6.1.1 Two-phase commit

The two-phase commit protocol (Gray & Reuter 1993) is the simplest way to ensure atomicity of transactions that involve several nodes in a distributed system. In a replicated database, it can be used to guarantee that all physical replicas of a logical database object are updated atomically, thus ensuring one-copy serializability and upholding consistency. A two-phase commit protocol consists roughly of the following steps:

Prepare to commit. In this step, a node chosen as the *coordinator* sends *prepare-to-commit* messages to all nodes that have taken part in the transaction.

Decide. Each node votes either *yes* or *no*, depending on whether it is able to

commit the transaction updates. The votes are sent to the coordinator.

Commit. If everyone votes yes, the coordinator broadcasts a commit decision to all participants, which commit the transaction. If any participant votes no, an abort decision will be broadcast instead.

Complete. When all participants have completed their operations, the coordinator completes the commit by writing an ET entry to the log.

A problem with regular two-phase commit is that it is blocking. A protocol is *blocking* if it allows executions where correct participants cannot decide whether to commit or abort (Babaoglu & Toueg 1994). This may, for example, be the case in a scenario where all of the following occurs (see figure 6.1):

- all participants vote correctly
- the coordinator sends the decision to one or more of the participants, then crashes
- the nodes which have received the decision (only node A in the figure) all crash

In this scenario, the correct participants (nodes B and C in the figure) cannot

decide whether or not they should commit until either the coordinator or one of the crashed participants has recovered.

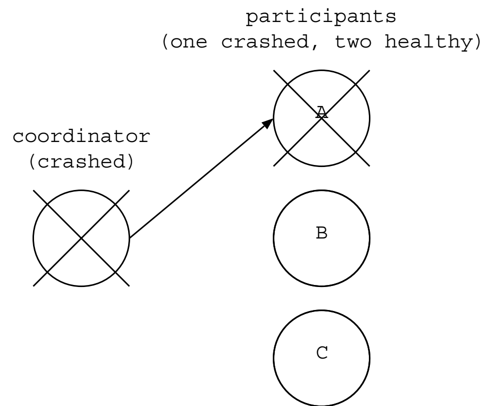

Figure 6.1: A blocking scenario

Blocking protocols are unsuitable in a real-time system, since they introduce unpredictability in the commit process, and since they are inefficient (blocked nodes may hold resources). It is therefore in our best interest to find *non-blocking* commit protocols.

6.1.2 Non-blocking commit protocols

The non-blocking commit problem is tackled by Skeen (1981). Using finite-state automata (FSA), Skeen formulates the *fundamental non-blocking theorem*, which states that the two (necessary and sufficient) conditions for a non-blocking commit protocol are that:

i) there exists no local state such that its concurrency set (see below) con-

tains both an abort and a commit state

(*ii*) there exists no **noncommitable** state (see below) whose concurrency set contains a commit state

In order to understand these conditions, we need to define a few terms. The *concurrency set* of a particular node's (local) state is the set of states that *may* be occupied by the other nodes in the system at the same time. Somewhat informally, a node in a *noncommittable* state does not know whether all the other sites have voted yes to commit. Condition (*i*) thus states that a node is blocked whenever it infers that two other nodes in the system have committed and aborted, respectively. Condition (*ii*) states that a node is blocked if it cannot commit due to lack of information about the other nodes, and if it cannot abort because another node may have committed and subsequently crashed.

Using the fundamental non-blocking theorem, Skeen (1981) presents two versions (one centralized and one decentralized) of a non-blocking commit protocol. Both versions add an additional phase to the commit protocol. This type of protocols is therefore often referred to as *three-phase commit* protocols.

There are, however, several problems with three-phase commit protocols. The most apparent is the extra overhead that is introduced in the third phase of the protocol. Three-phase commit algorithms also tend to be complex

both to understand and implement, often requiring elaborate communication between non-failed participants and the consideration of a large number of possible system states in order to reach the correct decision.

Noting this, Babaoglu & Toueg (1994) present a family of non-blocking protocols that are very similar to two-phase commit and its derivatives. The major difference is the broadcast primitive used to disseminate vote results to the participants. By using an implementation of *uniform timed reliable broadcast* (UTRB), it is possible to achieve *uniform agreement*, which means that if a participant (correct or incorrect) is able to act upon a vote result, all correct participants are eventually able to act upon that vote result.

Babaoglu & Toueg (1994) distinguish between a received message and a delivered message. When the communication subsystem at a node *receives* a message, it does not *deliver* it (i.e., make it available for local processing) until it has relayed that message to all other processes in the system.

Figure 6.2 shows an implementation of UTRB. In this figure, G is the set of all processes in the system, and m is the message to be broadcast.

The original paper presents extensions to the basic UTRB algorithm which improve its message- and time efficiency. In the most refined version of the algorithm, UTRB-3, the number of required messages is linearly proportional to the number of nodes, and the time required to deliver a message it more than halved from the original UTRB algorithm.

procedure broadcast(m, G)

%Broadcaster executes:
    send [DLV: m] **to all** processes in G
    deliver m

%Process p $\neq$ broadcaster in G executes:
    upon (first receipt of [DLV: m])
        send [DLV: m] **to all** processes in G
        deliver m

end

Figure 6.2: Uniform timely reliable broadcast (Babaoglu & Toueg 1994)

6.1.3 Other pessimistic approaches

Ceri, Houtsma, Keller & Samarati (1994) survey a number of replication protocols, many of which are pessimistic and guarantee one-copy serializability. It is found that most replication protocols can be seen as variants of a small number of core protocols, and Ceri et al. (1994) thus classify the protocols into *basic* and *derived* protocols. The basic protocols are (*i*) Read One, Write All (ROWA), (*ii*) Quorum Consensus (QC), (*iii*) As Soon As Possible (ASAP) and (*iv*) Differential File (DF). In the sections below, we will give a brief description of each of these techniques. For a presentation of the derived protocols, and how they relate to the basic protocols, the reader is advised to look up the original paper.

Read one, write all

Under the ROWA protocol, read operations are allowed to query any physical replica of the relevant logical object. Write operations, on the other hand, must obtain locks on all physical replicas of the objects that they wish to update. A somewhat less strict version of the ROWA protocol is Read One, Write All *Available* (ROWAA), where only the physical objects on those nodes available at the time of commit are updated with the changes. Non-available (crashed) nodes are updated as part of their recovery procedure, and transactions should be prevented from reading such nodes until they have been made consistent.

Quorum consensus

In a system using the QC protocol, each physical object is assigned a weight. Also, a *read threshold* (RT) and a *write threshold* (WT) are defined for each logical object, such that both 2*RT and RT+WT exceed the sum of all weights of the corresponding physical objects. A transaction is only allowed to read (or write) a logical object if it can access enough physical replicas (the *read quorum* or *write quorum*, respectively) of that object for their total weights to exceed RT*WT for the object in question. A read transaction is translated to a number of reads on the read quorum, which are guaranteed to return the most up-to-date value. This is made possible by timestamping objects each time they are updated. Similarly, a write operation is executed as a set of writes on a write quorum.

As soon as possible

Using ASAP replication, one of the physical objects is seen as the *primary copy* of a logical object. All updates are initially made to this primary copy (where concurrency control is conducted), and updates are sent (ASAP) to the other physical objects as independent transactions (sometimes called *ripple subtransactions* (Chundi, Rosenkrantz & Ravi 1996)). This is actually pessimistic only in that it avoids write-write conflicts. It is simliar to eventual consistency, and thus optimistic, in that it allows stale data to be read at non-primary nodes (see section 6.2).

Differential file

The DF protocol is similar to the ASAP replication protocol, but the changes made to the primary copy are stored in a differential file. This file is then used to update all other physical objects. The time at which the updates are made differ in the various implementation of this technique; either it can be done on access of an unupdated physical object, on user demand, or periodically. How and when updates are made dictates whether an implementation of a DF protocol is pessimistic or optimistic.

6.1.4 Recovery in immediately consistent databases

As discussed in section 5.5, recovery in a distributed and replicated database can use the inherent redundancy in the system to recover a crashed node.

For the discussion in this section and sections 6.2.4 and 6.3.2, we assume a recovery scheme where a node recovers its data by copying it from some node in the system where that data is replicated (such as in diskless recovery, see section 5.6). Remember that a logical data object is the abstract view of a data object that a database user sees. This logical object can be represented by several physical objects (replicas) on different nodes in the system (see section 2.2.1).

Conceptually, node recovery in an immediately consistent, distributed database does not differ significantly from recovery in a centralized database. In most immediately consistent systems, all physical replicas of a logical database object always have the same value on all healthy (non-crashed) nodes. For a recovering node to restore a physical object x, it only needs to read the corresponding logical object x'. How this is done is unimportant – it could, e.g., be read from a primary copy (ASAP replication), from a read quorum (QC), or from any site in the system (ROWA). Either way, since the logical object is always kept consistent by the consistency preservation mechanism, the recovered object is also consistent at the time it is copied. It may, however, become inconsistent at a later time during the recovery process unless the rest of the system is kept quiescent. This means that in a non-quiescent system, a log of all updates performed during recovery must be kept and applied to the recovering node after it has recovered its contents. A more thorough discussion on this can be found in section 6.3.2.

The different techniques for immediate consistency differ slightly, how-

ever, in the assumptions that must be made in order for recovery to work properly. 2PC and its non-blocking derivatives, along with the ROWA protocol, do not require any additional assumptions to ensure correct recovery. As long as there is at least one healthy node in the system, recovering nodes can copy data objects from any healthy node and be sure that the copied value is up to date. For quorum-based methods, enough healthy nodes must exist to form a read quorum. We thus need a t-fault assumption, where no more than t nodes may be down simultaneously for it to be possible to perform recovery. In primary copy-based methods (such as ASAP), correct recovery can only be guaranteed if the primary copy is healthy. Likewise, methods based on a differential file only guarantees correct recovery if the differential file can be made available to the recovering node.

From this discussion, we can conclude that a trade-off needs to be made. The more strict commit protocols, such as 2PC and ROWA, require that all nodes are available for writing at the time of commit. However, in systems which use these techniques, the assumptions required for correct recovery are very few and easily met (basically, the only assumption that must be fulfilled is that there exists one healthy node when recovery needs to be done). Techniques which do not put as harsh constraints on commit processing, such as QC and ASAP replication, require stronger assumptions on recovery.

6.2 Eventual consistency

In this disseratation, we have on several occasions touched upon the subject of eventual consistency (see section 2.2.1 for definition). In contrast to the *pessimistic* approaches to distributed database consistency and replication (such as the ones described in the previous section), the eventual consistency approach is *optimistic* in that it allows transactions to commit on a local node without worrying about the global state. It is assumed, when we replicate the locally committed updates to the other nodes in the system, that conflicts (due to concurrent access of the same logical database object by different nodes) will be rare, and that the conflicts that do occur will be relatively easy to resolve.

Having an optimistic approach to consistency preservation, such as eventual consistency, means that consistency is traded off for increased predictability, performance and availability. Performance is improved in a system with eventual consistency, since transactions can commit locally without communicating with the other nodes in the system, and without waiting for a voting protocol to be executed. Since internode communication is often done over an unpredictable network, predictability is also improved by eliminating such communication during commit processing. Finally, since resources (data objects) need to be locked on only the local node, and since locks are held for a small amount of time (due to high performance), the availability of an eventually consistent database is higher than that of an immediately

consistent database.

In order to support eventual consistency, we need first and foremost a *propagation protocol* that propagates updates made on a given node to the other nodes in the sytem. A propagation protocol can either be timely (in which case we refer to it as *bounded-delay replication*), or best-effort (*ASAP replication*). As noted in section 2.2.1 we also need a *conflict management method*. The crucial parts of such a method – conflict detection and conflict resolution – are discussed in the two following sections.

Furthermore, there are restrictions on applications that use an eventually consistent database. Since read transactions cannot be guaranteed to always return the most up-to-date value (it may have been updated at a different node, and the propagation of the update may not yet have reached every other node), the application must be resilient enough to not fail when using stale data. See section 6.2.2 for a discussion on this.

6.2.1 Conflict detection

An eventually consistent database needs to be able to detect when concurrent updates of a logical object conflict. Lundström (1997) discusses two major approaches to such conflict detection: *version vector algorithms* and *precedence graph protocols*. Since it was found that Lundström's modified version vector algorithm was most suitable for the DeeDS architecture (this because it works well with the DeeDS strategy of replicating only data, not transactions), we have chosen to focus only on this algorithm.

Version vector algorithms

The version vector algorithm was originally perceived by Parker & Ramos (1982) for detecting single file inconsistencies in LOCUS, a distributed operating system. Lundström (1997) extended the technique to make it more suitable for use in a distributed real-time database (DeeDS, see section 2.5). We will look at the technique from the database perspective. For this discussion, keep in mind that in a fully replicated database, a logical database object is stored as a number of physical objects, one on each node in the system.

Each physical object in the database is assigned a *version vector*. A version vector consists of a number of *entries* equal to the number of nodes[1] in the system. Each entry is on the form $N_i : V_i$, where N_i is a (unique) node identifier (each physical object thus has an entry in the version vectors of its corresponding logical object), and V_i is a version number. Whenever node N_i commits updates to a physical database object, it increases V_i by 1.

When the updates are replicated to the other nodes in the system, a conflict can be detected by observing that two version vectors for the same logical database object are incompatible. Two version vectors are said to be compatible if one dominates the other (see definition 6.1).

Definition 6.1: In a system with n nodes, a version vector W (with

[1] Lundström uses the term *site* instead of node. While both are used synonymously in the literature, we use the latter as it does not imply physical separation of the nodes.

entries $N_i : V_i$) *dominates* a vector W' (with entries $N_i : V_i'$) if $V_i \geq V_i'$ for $i = 1...n$.

As an example, study the execution illustrated in figure 6.3. The first step (a) shows the initial state of a database with two nodes (A and B), both containing a replica of the (logical) object x. Initially, the version vectors are identical, and the system is in a consistent state.

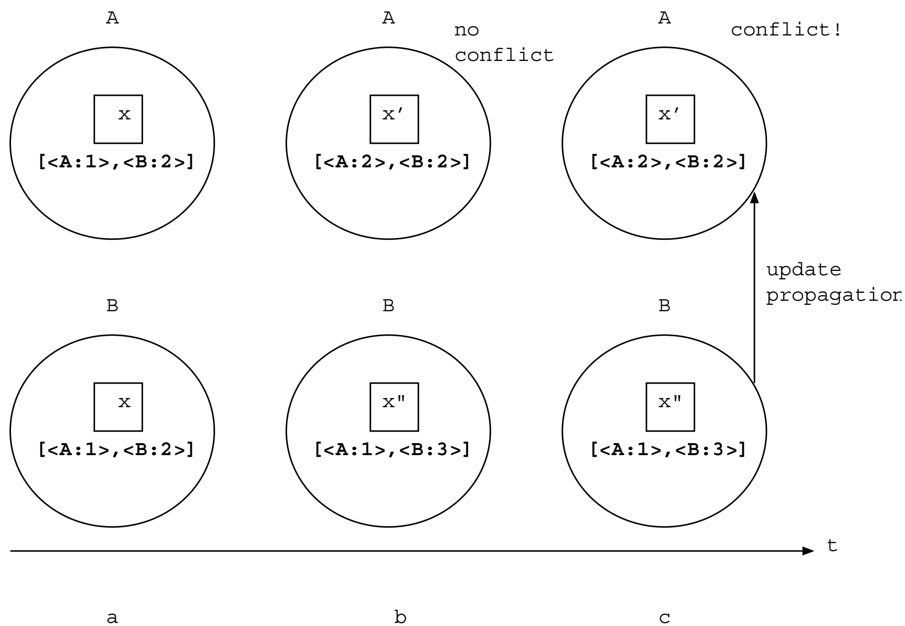

Figure 6.3: Version vector conflict detection

In the next step (b), both nodes simultaneously commit changes to object x, creating two versions: x' and x". The local versions vectors are updated to reflect the changes. The system is now in an inconsistent state, but the

conflict has yet to be detected.

In the last step shown in the figure (c), the conflict is detected. When B's updates are propagated to A, neither of the version vectors (A's local vector and the one propagated with x" from B) dominates the other ($2 \geq 1$ but $2 \leq 3$). Thus, we have a conflict.

In a way analogous to the original version vector algorithm's multiple file extension, the above technique can be extended to allow conflict detection for updates to multiple (logical) database objects by logging all version vectors associated with a transaction. For a thorough discussion on this, see Lundström (1997).

6.2.2 Conflict resolution

When a conflict has been detected (for example by using version vectors as described in the previous section), that conflict needs to be resolved. There are two basic approaches to performing conflict resolution: backward and forward. *Backward conflict resolution* is similar to backward transaction recovery. We *undo* the updates by either installing an older version of the affected database object, or by executing *compensating transactions*, which contain inversions of all updates made by the conflicting transactions. We then *redo* updates according to our conflict resolution policy – for example by applying the updates in some priority order. In a general real-time system this is not a very attractive strategy, since our worst-case assumptions (which dictate our resource requirements) need to consider the time needed

for backward conflict resolution and re-running of transactions.

Forward conflict resolution, on the other hand, aims to avoid the use of any type of undo. Instead, the conflicting updates are *merged* in some way that brings the database to a *new*, correct and consistent, state. Forward conflict resolution requires a certain amount of knowledge about the conflicting transactions' semantics. This type of conflict resolution can also be made by compensating transactions. To avoid confusion with the kind of compensating transactions that are used for undo, we call this type of transactions *convergence transactions.*

Another thing that must be considered is the *access patterns* of the database. Do we need to consider read-write conflicts only, or can write-write conflicts occur? In a *read-write conflict*, a transaction reads data which is simultaneously being updated (on another node) by a different transaction. The reading transaction thus reads stale data. A *write-write conflict* occurs when data is updated concurrently by transactions at different nodes. The example illustrated by figure 6.3 shows the detection of a write-write conflict.

Lundström (1997) identifies five conflict resolution policies:

- **Do nothing:** In the case of a read-write-conflict, it may be that old data can still be used to achieve an adequate result. For example, in a system controlling a furnace, it may not be absolutely necessary to get the very latest furnace temperature readings. There should, however, be a bound on the 'staleness' of the data.

- **Undo:** As discussed above, all updates done by conflicting transactions can be undone, either by reinstalling a BFIM (see section 5.1.3), or by executing a compensating transaction.

- **Average:** In a write-write conflict, if the updated object is numerical, an average value can be used. Note that if the updates are incremental in nature, care must be taken for this resolution strategy to work properly.

- **Time-stamp:** If the updates are time-stamped, the latest value can be used. This requires a global system clock.

- **Priority:** Similar to the technique using time-stamps, transactions (or nodes) can be assigned different priorities that are used to resolve a conflict. The update with the highest priority gets installed.

As can be seen in the descriptions above, which technique is the most appropriate varies from system to system. We need to consider not only access patterns and the type of basic resolution strategy we want (forward/backward), but also the nature of the updated data (numerical or not?) and properties of the system (do we have access to a global clock? Do we need to consider time constraints? Can we assign priorities to nodes or transactions in any relevant way?).

Besides the work by (Lundström 1997), there has been little work on conflict resolution policies that are suitable for eventually consistent databases. For databases where replicating transactions (as opposed to updates) is an option, Tewari (1993) suggests using a refinement of the *log transformation* technique (Blaustein & Kaufman 1985). In short, log transformation is a technique for finding pairs of transactions in an undo-redo log of transactions, which can be merged to a single transaction, or eliminated entirely. For example, two transactions that overwrite the same value can be replaced by only the overwrite transaction with the latest timestamp, and transactions that are inverses of each other (i.e., a transaction and its compensating transaction) can be eliminated from the log. Tewari (1993) extends this technique by further exploiting transaction semantics. For example, properties of arithmetic transactions (such as associativity and distributedness) can be used to merge transactions and thus reduce the amount of log that needs to be processed.

6.2.3 Other optimistic approaches

While we have chosen to focus on the eventual consistency approach to replication management and consistency preservation (because of its use in the DeeDS prototype), there are, of course, other optimistic techniques. For example, Carey & Livny (1991) contrasts two techniques (Distributed Certification (OPT) and Distributed Optimistic 2PL (O2PL)) where transactions are allowed to proceed locally without considering other replicas of the logical

objects that they update. These, like most optimistic techniques, differ from eventual consistency in their level of optimism. While an eventually consistent database allows transactions to commit locally, both OPT and O2PL require the updates to be validated against the other replicas before allowing the updating transaction to commit. We call this approach *semi-optimistic*. Using a semi-optimistic technique makes transaction recovery simpler (transactions which cannot commit are simply rolled back), while adding overhead in the form of network communication and validation procedures at each transaction commit.

Breitbart & Korth (1997) are some of the spokesmen for *lazy replication*, which is similar to eventual consistency. Breitbart and Korth's technique, however, requires each logical data object to have a *primary* copy on a particular node in the system. All updates on a logical data object are directed to the object's primary copy, where concurrency control is carried out. Local transactions are allowed to commit only if they have updated only (local) primary copies.

Using primary copy-based replication means that nodes must be discriminated, and it also means that a single point of failure is introduced for each database object. In the worst case, if the node with the primary copy for a logical database object crashes, that object cannot be updated until the node containing the primary copy has recovered. This may be solved by electing a new primary copy when the crash is detected, but this introduces overhead in the form of an election protocol, causing transactions to wait for a new

primary to appear. This limits the availability of the approach. Furthermore, the contention for the primary copy, as well as the communication needed to access the primary, restricts its timeliness.

6.2.4 Recovery in eventually consistent databases

We argue that recovery in an eventually consistent database is more complex than in an immediately consistent database. While in an immediately consistent database there is always a uniform view of a logical database object, in an eventually consistent system there may be several *different* physical representations (replicas) of a logical object. Thus, if we copy a database object from an arbitrary healthy node, we cannot be sure that we read an up-to-date copy. It may even be that the physical object that we read conflicts with another physical replica of the same object. In that case, any conflict resolution that is performed on the recovery source's object must be applied to the recovered object as well.

Leifsson (Leifsson 1999) presents a recovery protocol for use in an eventually consistent database (see section 5.6). The recovery protocol guarantees eventual consistency by having the recovery source propagate updates to the recovery target until it can be guaranteed that the recovery target will receive all update propagations from all nodes in the system. However, Leifsson does not discuss conflict detection and conflict resolution. As we have just discussed, a recovered object may be in conflict with other replicas of that object in the system. This conflict must be detected, and the conflict resolution

protocol must also allow newly recovered nodes to become consistent.

The other optimistic (or semi-optimistic) consistency preservation techniques discussed in this section do not have this problem. The commit protocols of O2PL and OPT do not allow replicas to deviate, which means that conflicts between physical object replicas can never occur. Thus, any recovered database object becomes immediately consistent. The same holds true for lazy replication, as long as only primary copies of objects are used for recovery. Note that lazy replication has problems similar to ASAP replication (see section 6.1.3), in that recovery can only be performed if all nodes with primary copies of the database objects to be recovered are healthy.

6.3 Analysis

The major property that distinguishes replication protocols is their degree of optimism. While the pessimistic approach is simple and attractive in that it guarantees consistency of replicated objects at all times, it is unsuitable for many systems. In small systems (with few nodes and thus few replicas), we may not gain much from an optimistic protocol, but in larger systems, where availability, timeliness, and/or performance is paramount, increased optimism is really the only option. The user of an ATM, for example, will not tolerate a long delay just because the update he or she wishes to perform on an account must be replicated (with immediate consistency) to all nodes (banks or ATMs) in the system. As with optimistic and pessimistic concur-

rency control, the general consensus seems to be that for large and highly available systems, or for systems where performance and/or timeliness is required, we need to move from pessimistic approaches to more optimistic as argued by Breitbart & Korth (1997).

6.3.1 A comparison to concurrency control

Concurrency control and consistency preservation in a replicated database are very similar and, as a result, easily confused. Both concurrency control and consistency preservation techniques can be divided into pessimistic and optimistic approaches (where pessimistic approaches strive to avoid conflicts at all costs, and optimistic approaches assume that conflicts will be rare, but that they can be detected and resolved when they do happen).

A concept often mentioned in the context of consistency preservation is one-copy serializability (see section 6.1 for definition). One-copy serializability is often used as a correctness criterion for a consistency preservation technique in a replicated system. It is our view that in a pessimistic system, consistency preservation is analogous to providing concurrency control for the logical objects in the database (the one-copies). As we increase the optimism in the system, this ceases to be the case. In an eventually consistent system, different physical representations of a logical database object can be updated independently, and concurrency control is only performed on the local nodes. Thus, the one-copy serializability criterion may be violated.

Since concurrency control and recovery are tightly connected (see section

2.4.4), and since consistency preservation is related to concurrency control as described above, our hypothesis is that the choice of consistency preservation mechanism affects recovery in a way similar to the choice of concurrency control mechanism (figure 6.4). Since concurrency control and consistency preservation are usually not treated separately (in many systems they are inseparable), this relation is as yet unexplored, and would be of interest to investigate further.

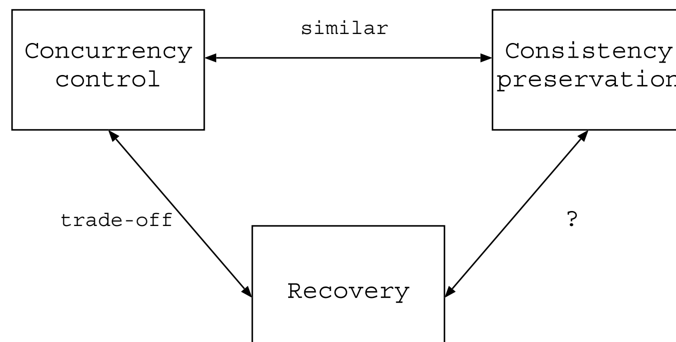

Figure 6.4: Interdependencies

6.3.2 Recovery and consistency preservation

As we have discussed in this chapter, the choice of consistency preservation mechanism affects recovery processing in several ways. We argue that an immediately consistent system must adhere to the failure assumptions that follow from the choice of consistency preservation mechanism, in order for recovery to work properly (see section 6.1.4). In an eventually consistent system, conflict detection and resolution must also be supported by the recovery

mechanism. How this is done is an as yet unexplored area and an interesting research subject.

The simplest form of distributed recovery, illustrated in figure 6.5, assumes a system where all non-recovery operations are quiesced during recovery. This means that the database on a recovered node will be immediately consistent with the rest of the system. Since no database updates are made on the non-recovering nodes during recovery, we do not need to send a log of such updates to the recovery target after it has restored its state (taken its "checkpoint").

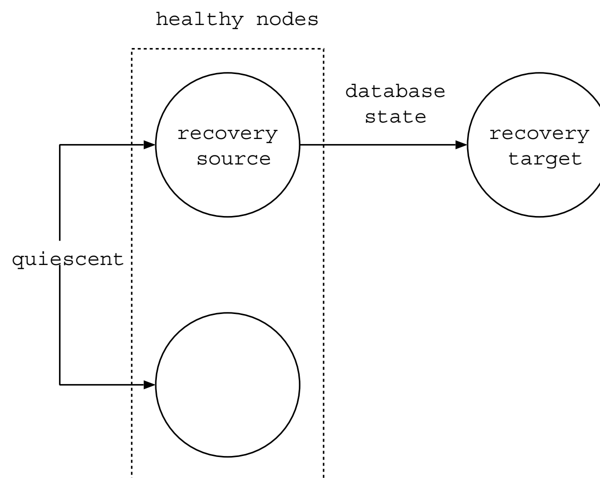

Figure 6.5: Distributed recovery in a quiescent system

If the system cannot be quiesced due to, e.g., time constraints or other system requirements, the recovery mechanism must be able to tolerate that database updates on healthy nodes are performed concurrently with the recovery process (see figure 6.6) Such updates can be committed in a way

that keeps the system immediately consistent (using, for example, any of the methods described in section 6.1). Alternatively, the database could be kept only eventually consistent. This means that conflict detection and conflict resolution must be considered.

As noted by Leifsson (1999), copying the contents of a recovery source to the recovery target in a non-quiescent system is analogous to installing a fuzzy checkpoint on the recovery target. This means that the recovery target does not immediately become (even locally) consistent. To compensate for this, the recovery source must log all updates that are done to it during the recovery process. After the "checkpointing" is completed, the log must be sent and applied to the recovery target.

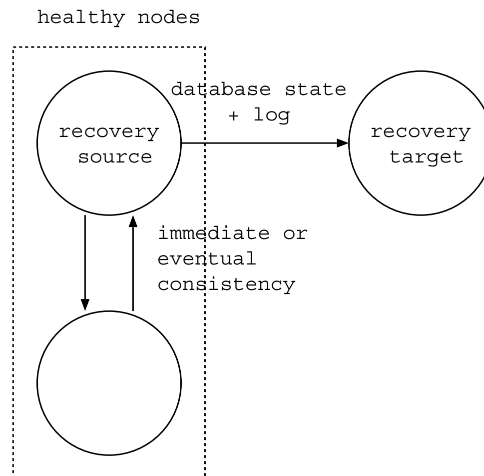

Figure 6.6: Distributed recovery in a non-quiescent system

In addition to consistency, durability for updates made on the recovering node *before* the crash must be guaranteed. This can be achieved by logging

such updates on stable storage, or by ensuring that the updates have been replicated to a buddy node (see figure 6.7 and section 5.6). For details about the buddy system, see Leifsson (1999).

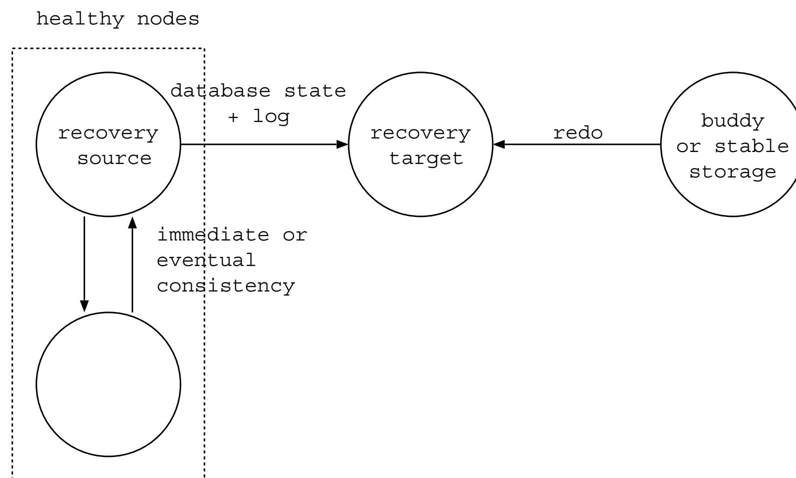

Figure 6.7: Distributed recovery and redo

Leifsson (1999) shows informally that in the absence of conflicts, the diskless recovery mechanism preserves eventual consistency, and also supports durability of local updates. Leifsson's approach is the only distributed recovery mechanism that we have found that considers both eventual consistency and durability. It is therefore an interesting research direction to formally show that diskless recovery works correctly in an eventually consistent system. It would also be useful to evaluate a real-world implementation of diskless recovery, and investigate what constraints an eventually consistent system with diskless recovery puts on the database applications that are run in the system.

6.3.3 Applicability in DeeDS

One of the goals of the DeeDS project is to build a database system that supports high-performance, embedded real-time systems with hard time constraints. In the light of this, eventual consistency is a much more natural choice of consistency preservation paradigm than immediate consistency. Seeing that timeliness and performance are paramount, we want to avoid the overhead associated with commit protocols that require inter-node communication. Since we have already established that diskless recovery is a preferable recovery algorithm for DeeDS (see section 5.7.1), we again stress the importance of investigating the relationship between diskless recovery and eventual consistency.

Chapter 7

Conclusions

This chapter contains the conclusions that we have drawn from the work presented in this dissertation. Initially, we try to identify "research tracks" and problems that we have found to be omitted or sparsely covered in the literature. Section 7.2 contains a discussion on related work, while section 7.3 lists some suggestions for future work within the field. We conclude the chapter with a summary of our contributions.

7.1 Omissions and uninvestigated problems

Our biggest concern with the state of the art in distributed database recovery techniques is the unexplored relationship between recovery and consistency preservation (see figure 6.4). While consistency during recovery in pessimistic and semi-optimistic (see section 6.2.3) can be guaranteed by the concurrency

control mechanism, recovery in an eventually consistent database has only been touched upon very briefly by Leifsson (1999).

Another neglected area is *real-time recovery* in distributed systems. While recovery schemes have been suggested for real-time databases (see for example Burkes & Treiber (1990) and Song, Kim, Ryu, Choi, Kim, Jin, Han & Choi (1999)), few consider distributed systems. In our survey, the only technique which was specifically modelled for real-time recovery in a distributed database is the diskless recovery approach described by Leifsson (1999).

We have also been unable to find any extensive investigation of the requirements on an application that operates in an eventually consistent database. The need for an application to cope with stale data and conflicts detected *after* a transaction commit means that the eventual consistency approach is applicable only to systems of certain properties. Typical for such systems is a focus on high availability or guaranteed timeliness at the expense of increased effort to ensure convergence of the system in case of temporary inconsistencies. Identifying the exact application requirements and restrictions in order to cope with eventual consistency is a task that has yet to be done.

7.2 Related work

A survey similar to the one in this project has been performed by Ceri et al. (1994). While their paper aims primarily to provide a taxonomy of replica-

tion algorithms, they also consider both normal and *abnormal* behaviors of the algorithms, where abnormal behavior is how the algorithm copes with network partitions. Since recovering a node in DeeDS is very much analogous to it reentering the system after a partition, much of the reasoning of abnormal behavior done by Ceri et al. (1994) can be applied to our discussion of recovery as well.

Of particular interest to our work is the *independent update* algorithm which is presented very briefly by Ceri et al. (1994). Unfortunately, very little is said about the implications that using the independent update algorithm has on the database system and its applications. The only things mentioned by the authors are that the reconciliation (conflict resolution) algorithm should ensure one-copy serializability (which is not necessarily the case for eventual consistency), and that the applications must be capable of handling stale data.

The focus of Ceri et al. (1994) is also different from ours. Their taxonomy is meant primarily as an aid for system developers, and their major contribution is the discovery that most replication algorithms are derivations of four basic algorithms (see section 6.1.3).

7.3 Research directions and future work

We have chosen to present the future work that we have identified in three categories. The first category concerns distributed database recovery prob-

lems that are general in that they are not necessarily specific to any particular recovery- or consistency preservation mechanism. The second category contains issues which are specific to eventually consistent systems. Finally, in the third category we list extensions that can be made to Leifsson's diskless recovery mechanism (Leifsson 1999).

7.3.1 General problems in distributed database recovery

Distributed database recovery is a relatively well-covered area, but we have identified two areas where we think that further research effort is needed. The first is the apparent lack of timely recovery mechanisms for distributed databases (see section 7.1). The only discussion on timely, distributed recovery that we have found is done by Leifsson (1999), who briefly discusses the timeliness of his diskless recovery mechanism. We believe, however, that a more formal and less method-specific look at how distribution affects the timeliness of recovery is needed.

For replicated databases, we would also like to see the relationship between consistency preservation and recovery explored, as discussed in section 7.1.

7.3.2 Eventual consistency problems

As mentioned in the previous section, we are interested in the interaction between concurrency preservation and recovery mechanisms. More specifically, it would be interesting to investigate how the conflict detection and resolution protocol affects recovery in an eventually consistent system.

Another interesting research topic is how applications are affected by operating in a system with an eventually consistent database. The applications must be built to tolerate possible inconsistencies in the data on which they operate. To support application development for an eventually consistent system, application requirements must be identified.

7.3.3 Diskless recovery extensions

Leifsson (1999) describes a diskless recovery mechanism for distributed main-memory databases. The mechanism could be further refined by relaxing some of its assumptions. For example, Leifsson assumes only single-node failures. It would be interesting to investigate how the mechanism needs to be modified to be able to handle several simultaneous node failures. Further, Leifsson suggests that incremental recovery techniques be looked into, in order to allow nodes to resume partial service before the recovery process is completed. However, incremental recovery may not be required for our system model (DeeDS) for reasons given in section 5.4.2.

No empirical investigation or implementation has been made of Leifsson's

algorithm. It would be very interesting to integrate diskless recovery in DeeDS and measure its performance. This would also allow us to investigate its relation to DeeDS's eventual consistency scheme. In addition, formally proving that the diskless recovery mechanism correctly preserves eventual consistency in a replicated database would be useful.

7.4 Contributions

We beleive that this work contains several contributions;

- A state of the art report for recovery and consistency control in distributed databases
- A taxonomy of existing recovery- and consistency preservation methods
- Identification of possible research directions
  - timeliness for distributed recovery
  - interdependencies between consistency preservation and recovery
  - application constraints for eventually consistent databases

Our main contribution is a view of the current state of the art in two major areas – database recovery and consistency preservation in distributed databases. We focus particularly on their applicability in fully replicated, real-time, main-memory database systems.

We present a taxonomy for techniques within the given fields, and also

identify important properties of recovery- and consistency preservation mechanisms which dictate their appropriateness for a given system.

Most of our contributions are identifications of areas where we believe that more research effort is needed. We point out that the intricate relationship between consistency preservation and recovery has not been investigated in depth. It is our hypothesis that in an eventually consistent database, where one-copy serializability may be a too strong correctness criterion for a conflict resolution scheme, concurrency control and consistency preservation must be treated separately in spite of their apparent similarity. Since concurrency control greatly affects recovery, we believe that consistency preservation must be considered as well when designing a recovery mechanism for an eventually consistent database.

We also note that timeliness is not considered in most approaches to recovery in a distributed system. Since many real-time systems are also distributed, and since most also employ some kind of database, real-time distributed recovery is a research topic which needs increased attention.

For eventually consistent databases to become usable in real-world systems, requirements for applications that operate on the database must be identified.

Chapter 8

Acknowledgments

> *"While I'm still confused and uncertain, it's on a much higher plane, d'you see, and at least I know I'm bewildered about the really fundamental and important facts of the universe." Treatle nodded. "I hadn't looked at it like that," he said, "but you're absolutely right. He's really pushed back the boundaries of ignorance."*
>
> *They both savoured the strange warm glow of being much more ignorant than ordinary people, who were only ignorant of ordinary things.*
>
> – *Terry Pratchett, Equal Rites*

My life seems to be full of wonderful people, and I would like to thank the particular subset of them that have helped me reach the end of this project. First of all, I'd like to thank my supervisor, Sten F. Andler, for sacrificing so

much of his time to give me his full support even in times when the “brain damage” on my behalf must have been overwhelming. Thanks also to the rest of the distributed real-time systems group, in particular my partners in crime – Robert Nilsson, Birgitta Lindström, and Mats Grindal – for providing great company and being a fantastic group in which to work. Further thanks go to Bengt Eftring for reading and commenting on this report, and to Jonas Mellin for useful feedback on my presentations.

I would also like to express my appreciation of the work previously performed by Egir Örn Leifsson, Johan Lundström, and Per Gustavsson here at the department. These are the giants on whose shoulders I'm trying to keep my balance.

Completing a project of this size could never be done without a loving family and dependable friends, and I'm fortunate enough to have been blessed with both. To the folks back home, and to Simon, Johanna, Johan, Robert, Hanna, and Maud – thanks for being there. You guys are the best.

Bibliography

Andler, S., Hansson, J., Eriksson, J., Mellin, J., Berndtsson, M. & Eftring, B. (1996), 'DeeDS towards a distributed and active real-time database system', *SIGMOD Record* **25**(1), 38–40.

Andler, S., Leifsson, E. & Mellin, J. (1999), Diskless real-time database recovery in distributed systems, in 'Work-In-Progress of 20th IEEE Real-Time Systems Symposium'.

Babaoglu, O. & Toueg, S. (1994), Non-blocking atomic commitment, in S. Mullender, ed., 'Distributed Systems', Addison-Wesley, chapter 6.

Blaustein, B. & Kaufman, C. (1985), Updating replicated data during communication failures, in 'Proceedings of the 11th International Conference on Very Large Data Bases', pp. 49–58.

Bohannon, P., Parker, J., Rastogi, R., Seshadri, S., Silberschatz, A. & Sudarshan, S. (1996), Distributed multi-level recovery in main-memory databases, in 'Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques'.

Breitbart, Y. & Korth, H. (1997), Replication and consistency: Being lazy helps sometimes, in 'Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, Tucson, Arizona'.

Burkes, D. & Treiber, R. (1990), Design approaches for real-time transaction processing remote site recovery, in 'Proceedings of the IEEE Spring CompCon Conference', pp. 568–572.

Burns, A. & Wellings, A. (1997), *Real-Time Systems and Programming Languages*, Addison-Wesley.

Carey, M. & Livny, M. (1991), 'Conflict detection tradeoffs for replicated data', *ACM Transactions on Database Systems* **16**(4), 703–746.

Ceri, S., Houtsma, M., Keller, A. & Samarati, P. (1994), A classification of update methods for replicated databases, Technical Report CS-TR-91-1392, Stanford University, Computer Science Department.

Chundi, P., Rosenkrantz, D. & Ravi, S. (1996), Deferred updates and data placement in distributed databases, in 'Proceedings of the Twelfth International Conference on Data Engineering, New Orleans, Louisiana', pp. 469–476.

Dunham, M., Lin, J.-L. & Li, X. (1996), Fuzzy checkpointing alternatives for main memory databases, in V. Kumar & M. Hsu, eds, 'Recovery Mechanisms in Database Systems', Prentice Hall, chapter 21.

Eich, M. (1987), A classification and comparison of main memory database recovery techniques, in 'Proceedings of the Third International Conference on Data Engineering, February 3-5, Los Angeles, California, USA', IEEE Computer Society, pp. 332–339.

El Abbadi, A., Skeen, D. & Cristian, F. (1985), An efficient, fault-tolerant protocol for replicated data management, in 'Proceedings of the 4th ACM Symposium on Principles of Database Systems', pp. 215–229.

Gray, J. & Reuter, A. (1993), *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, chapter 10.

Gruenwald, L., Huang, J., Peltier, A., Lin, J.-L. & Dunham, M. (1996), 'Survey of recovery in main memory databases', *International Journal of Engineering Intelligent Systems, A Special Issue on Databases and Telecommunications* **4**(3), 57–63.

Haerder, T. & Reuter, A. (1996), Principles of transaction-oriented database recovery, in V. Kumar & M. Hsu, eds, 'Recovery Mechanisms in Database Systems', Prentice Hall, chapter 3.

Hsu, M. & Kumar, V. (1996), Introduction to database recovery, in V. Kumar & M. Hsu, eds, 'Recovery Mechanisms in Database Systems', Prentice Hall, chapter 2.

Jagadish, H., Silberschatz, A. & Sudarshan, S. (1993), Recovering from main-memory lapses, in '19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings'.

Leifsson, E. (1999), Recovery in distributed real-time database systems (HS-IDA-MD-99-009), Master's thesis, University of Skövde.

Levy, E. & Silberschatz, A. (1992), 'Incremental recovery in main memory database systems', *IEEE Transactions on Knowledge and Data Engineering* pp. 529–540.

Li, X. & Eich, M. (1993), Post-crash log processing for fuzzy checkpointing main memory databases, in 'Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria'.

Li, X., Eich, M., Joseph, V., Gulzar, Z., Corti, C., Nascimento, M. & Peltier, A. (1995), Checkpointing and recovery in partitioned main memory databases, in 'Proceedings of the International Conference on Intelligent Information Management Systems, Washington, DC, June 7-9', pp. 59–63.

Lin, J.-L. (1997), 'A survey of distributed database checkpointing', *International Journal of Distributed and Parallel Databases.*

Lin, J.-L. & Dunham, M. (1995), Low-cost checkpointing approaches for distributed databases, Technical Report 95-CSE-xx, Southern Methodist University.

Lin, J.-L. & Dunham, M. (1996), Segmented fuzzy checkpointing for main memory databases, in 'Proceedings of the ACM Symposium on Applied Computing, Pennsylvania, USA, February 17-19', pp. 158–165.

Lundström, J. (1997), A conflict detection and resolution mechanism for bounded-delay replication (HS-IDA-MD-97-10), Master's thesis, University of Skövde.

Parker, D. & Ramos, R. (1982), A distributed file system architecture supporting high availability, in 'Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks', pp. 161–183.

Skeen, D. (1981), Nonblocking commit protocols, in M. Stonebraker, ed., 'readings in Database Systems', Morgan Kaufmann, chapter 3, pp. 249–258.

Song, E.-M., Kim, Y.-K., Ryu, C., Choi, M., Kim, Y.-K., Jin, S.-I., Han, M.-K. & Choi, W. (1999), No-log recovery mechanism using stable memory for real-time main memory database systems, in 'Proceeding of the Sixth International Conference on Real-Time Computing Systems and Applications'.

Tewari, R. (1993), Efficient parallel recovery in replicated databases, in N. Adam & B. Bhargava, eds, 'Advanced Database Systems', Lecture Notes in Computer Science, Springer-Verlag, pp. 277–287.

Weihl, W. (1989), The impact of recovery on concurrency control (extended abstract), in 'Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, Philadelphia, Pennsylvania, USA', pp. 259–269.

Weihl, W. (1994), Transaction-processing techniques, in S. Mullender, ed., 'Distributed Systems', Addison-Wesley, chapter 13.

Wiesmann, M., Pedone, F. & Schiper, A. (1999), 'A systematic classification of replicated database protocols based on atomic broadcast', web page as of july 20, 2000: http://lsewww.epfl.ch/Documents/html/WPS99.html.

Ylönen, T. (1995), Shadow paging is feasible, Technical Report tkk-tko-tr-B123-1995, Department of Computer Science, Helsinki University of Technology.

Young, S. (1982), *Real Time Languages: Design and Development*, Ellis Horwood.