Recovery in Distributed Real-Time Database Systems

HS-IDA-MD-99-009

Ægir Örn Leifsson

Submitted by Ægir Örn Leifsson 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 1999

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

Ægir Örn Leifsson

Abstract

Recovery is a fundamental service in database systems. In this work, we present a new mechanism for diskless real-time recovery in fully replicated distributed real-time database systems. Traditionally, recovery has relied on disk-resident redundant data. Unfortunately, disks cannot always be used in real-time systems since these systems are sometimes used in environments which do not allow the use of disks. Also, minimizing the amount of hardware can save money, especially in mass-produced products. Instead of loading the database from disk, our recovery mechanism enables a restarted node to retrieve a copy of the database from an arbitrary remote node. The recovery mechanism does not violate timeliness during normal processing and, during recovery, all nodes except for the recovering node can guarantee the timeliness of critical transactions. The mechanism uses fuzzy checkpointing to copy the database to the recovering node. Fuzzy checkpointing has been chosen since it copies the database without regard to concurrency control and, thus, does not increase data contention in the database. We conclude that the suggested recovery mechanism is a feasible option for fully replicated distributed real-time database systems.

Acknowledgments

I would like to thank my supervisor Jonas Mellin for all his good advice during this project. Without his help I surely would not have succeeded. Furthermore, I want to thank my examiner Prof. Sten F. Andler for the many discussions we have had during the course of the project. The idea for the project was his and he has provided invaluable input to this work and led me in new directions.

I would also like to thank Jörgen Hansson for always being available for discussion when his expertise was needed and Ragnar Birgisson for his good advice during the initial stages of the project. Also, C. Mohan helped by giving ideas on which direction the project should take.

I want to thank my classmates from the MSc class of 1999 for the year we spent together at the University of Skövde and for interesting discussions and exchange of ideas during this year. In particular, my good friend Ragnar Steinsen has provided useful comments during the course of the project.

Finally, to my family, thank you for your support and patience during my years abroad.

The best of luck to you all in your future endeavours.

Contents

1 Introduction **1**

1.1 Recovery in Distributed Real-Time Database Systems 2

1.2 Overview of the Dissertation 4

2 Background **6**

2.1 Distributed Real-Time Database Systems 7

2.1.1 Database Systems 7

2.1.2 Real-Time Issues 12

2.1.3 Main-Memory Database Systems 15

2.1.4 Distribution Issues 16

2.2 Main-Memory Database Recovery 18

2.2.1 General Recovery 19

2.2.2 Inadequacy of Existing Approaches from Disk-Based Systems 21

2.2.3 Improved Recovery Approaches 23

3 The Distributed Real-Time Recovery Problem **26**

3.1 Motivation for Distributed Recovery 26

3.2 Distributed Real-Time Database Management System Model 29

3.3 Problems in Distributed Recovery 33

3.4 Building a Consistent Database View 34
3.5 Guaranteeing Durability for Locally Committed Transactions 35

4 Our Approach to Recovery in Distributed Real-Time Database Systems 37
4.1 Overview 38
4.2 Building a Consistent Database View 41
4.2.1 Checkpointing 48
4.2.2 Logging 51
4.2.3 Replication Forwarding 53
4.3 Guaranteeing Durability for Locally Committed Transactions 57

5 Evaluation of the Recovery Mechanism 63
5.1 Overview 63
5.2 Timeliness 64
5.3 Resource Requirements 66
5.4 Feasibility of Implementation 68
5.5 Related Work 70

6 Conclusions 74
6.1 Summary 75
6.2 Contributions 78
6.3 Future Work 79

Bibliography 83

List of Figures 88

Chapter 1

Introduction

Database systems, including distributed real-time database systems, have to be able to recover from failures. *Recovery* is the process of restoring a database to a correct state after failure [CBS98]. For transactions to display atomicity and durability (two of the ACID properties for transactions), recovery must be provided by the system. Atomicity means that updates made by partially completed transactions are never visible in the database. Durability, on the other hand, means that updates made by successfully completed transactions are always visible and permanent in the database (until they are overwritten by another transaction which also completes successfully).

Database recovery has been the subject of research for several decades. A well known early implementation of a recovery mechanism can be found in the IBM IMS/360 system from 1969 [Obe98]. In spite of the long history of recovery mechanisms, we have been unable to find any research on real-time recovery in distributed real-time database systems, the focus of this work.

In the following section, we give an overview of this work. The problem is introduced and motivations given for why it is of interest to solve it. We then discuss the

proposed solution and evaluation results.

In section 1.2, the organization of this dissertation is presented and the contents of the remaining chapters described.

1.1 Recovery in Distributed Real-Time Database Systems

As already stated, this work deals with real-time recovery in distributed real-time database systems. That is, the system should continue executing critical transactions during recovery. We assume a distributed real-time database management system model in which the database is *fully replicated*, i.e. each node holds a complete copy of the database. Also, each node stores the database entirely in volatile main-memory. Finally, each node can commit transactions locally without consulting remote nodes.

The motivation for this work is the desire to provide a recovery mechanism which does not require that each node in the system is equipped with a disk. Avoiding disks can be beneficial both for environmental and financial reasons. For example, real-time systems are sometimes used in environments which do not tolerate disks, e.g. due to vibrations. Also, minimizing the need for hardware can save money, especially when a product is mass-produced.

We address two fundamental problems in diskless distributed recovery. Firstly, how can a node return to normal processing after a crash when it has lost its database copy and, secondly, how can durability be guaranteed for locally committed transactions. Since we are dealing with real-time systems, timeliness must be considered an important issue when approaching both of these problems.

In our approach, a restarted node, called the *recovery target*, obtains a copy of the

database from a healthy node, the *recovery source*. The recovery source first makes one sweep through the entire database, copies it page-by-page, and sends the copy to the recovery target. This is done with minimal disturbance to transaction processing at the recovery source, which can continue guaranteeing transaction timeliness while it copies the database. Since the recovery source may be altering the database as it is being copied, all updates are logged and sent to the recovery target. The recovery target then applies this log to the database copy it has received and, thus, obtains a consistent database copy.

In order to guarantee durability for locally committed transactions, we have suggested two possible approaches. The first approach is to have every node inform one other node of all updates being made by local transactions before committing. This approach is based on the assumption that if one of the nodes crashes, the other one will have time to replicate the updates before it also crashes. Under this assumption it is sufficient for two nodes to know about an update in order to guarantee its durability. The second approach uses non-volatile memory to hold database updates for use by the recovery process in the case of node failure.

Our evaluation indicates that the suggested recovery mechanism is a feasible choice for distributed real-time database systems. In other words, it is possible for a system to retain timeliness while using the mechanism. The copying of the database does not increase data contention and can itself be executed as a non-real-time task. Both approaches described for guaranteeing durability can be made predictable and, thus, they are applicable to real-time processing.

The copying of the database does not require any dedicated hardware. On the other hand, guaranteeing durability either requires dedicated network resources to ensure timeliness or non-volatile memory. Moreover, the system needs to be designed

in such a way that it can tolerate the increased load when recovery is in progress.

To summarize, we have suggested a real-time recovery mechanism which we believe is a viable option for distributed real-time database systems. It allows timeliness and can guarantee that transaction durability is not violated.

1.2 Overview of the Dissertation

Chapter 2 covers material which is necessary for the understanding of the rest of the dissertation. We start by discussing distributed real-time database systems and then present the basics of main-memory database recovery.

In chapter 3, we present the problem that is tackled in this work, i.e., real-time recovery in distributed real-time database systems. We motivate why we want to do distributed recovery and present a distributed real-time database management system model. After that, we give an overview of the problems in distributed recovery and identify two problems which must be solved. The chapter then ends with a more detailed description of these two problems.

Chapter 4 contains the proposed solution to the problem from chapter 3. The chapter starts with an overview. After this overview we take a closer look at each of the two problems and present the proposed solutions to these.

In chapter 5, we present an evaluation of the solution from chapter 4. The chapter opens with an overview, which is followed by a more detailed evaluation of the solutions to each of the two problems. In particular, we consider how our solutions relate to timeliness in the system, which requirements our algorithms put on the system, and whether an implementation of our solution is feasible. Chapter 5 ends with a discussion about related work in the field of database recovery.

Chapter 6 begins with a summary of this work. We then highlight the contributions of this work to the real-time database community and, finally, possible future research directions are discussed.

Chapter 2

Background

Database systems have been used for decades to handle large amounts of data. A relatively new use for database systems is in real-time applications. In 1988 a SIGMOD Record special issue on real-time database systems [RTD88] was published and, examples of early workshops on real-time databases are ARTDB-95 [BH95] and RTDB-96 [BLS97]. Traditional database systems are designed to minimize average response time. In contrast, real-time databases must be able to guarantee a response within a certain time. Due to the environments in which real-time systems are often used (e.g. process control), a distributed model is often the most natural one.

Since database systems are trusted with large amounts of data, it is important that data is not lost. This is where recovery fits in, in other words, a database should not loose any data even when it crashes.

In this chapter, we start by discussing various aspects of distributed real-time database systems in section 2.1. We consider database systems and recovery in general in section 2.1.1, and take a look at real-time issues in section 2.1.2. The real-time issues then lead us to a discussion about main-memory databases in section 2.1.3.

We conclude section 2.1 by considering distribution in real-time database systems.

In section 2.2, recovery mechanisms in main-memory databases are discussed. In section 2.2.2, we argue that traditional recovery mechanisms from disk-based database systems do not work well for main-memory databases and, in section 2.2.3, we present better ways of handling recovery in main-memory database systems. One of these approaches is fuzzy checkpointing, which is examined closer in chapter 4.

2.1 Distributed Real-Time Database Systems

Andler et al. [AHE+96] state that complex real-time systems "...often require distribution and sophisticated sharing of extensive amounts of data, with full or partial replication of the database." For example, distributed real-time database systems can be used in integrated vehicle systems control and automated manufacturing.

2.1.1 Database Systems

In this section, we start by defining the term database system and then discuss why database systems are a useful. We conclude this section by presenting the notion of transactions and recovery.

Elmasri and Navathe [EN94, pp. 2-3] define a *database system* as consisting of a database and a database management system. A *database*, is defined as "...a collection of related data." Further, Elmasri and Navathe state that:

> ...a database has some source from which data are derived, some degree of interaction with events in the real world, and an audience that is actively interested in the contents of the database.

The software used to create and maintain a database is called the *database management system (DBMS)* [EN94, pp 2]. Figure 1 shows how a database and a DBMS form a database system.

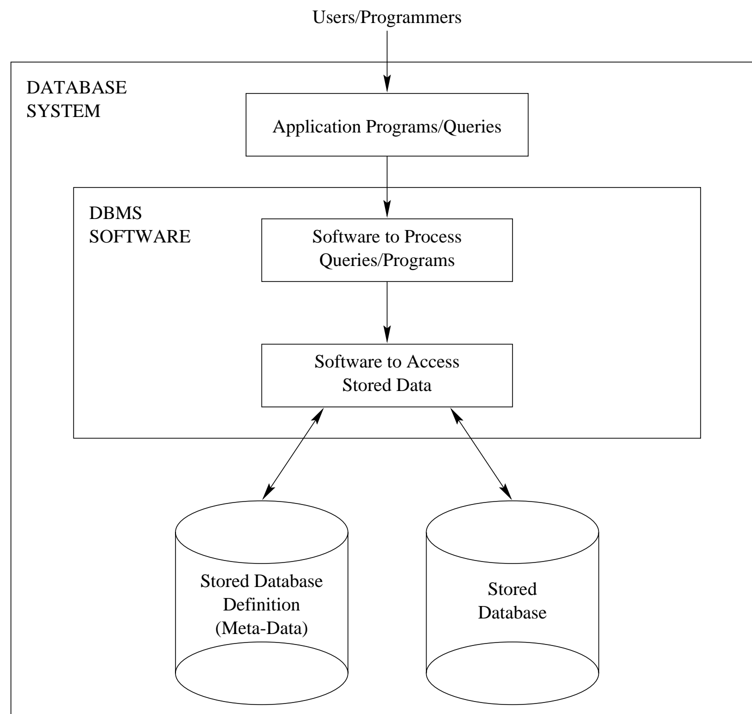

Figure 1: A simplified database system environment [EN94, pp 3].

As described by Elmasri and Navathe [EN94], a database system presents a conceptual representation of data to the database users, which can be other computer programs. This means that the programs can be isolated from the data, i.e. in order to use the database it is not necessary to know how data is stored and manipulated by the database management system. Therefore, it is not necessary to modify all programs using a database if the internal data representation in the database changes. Also, when a database is used, it is possible to provide different users with different

perspectives or views of the data.

Multiuser database systems allow multiple users (or applications) to access a database at the same time [EN94]. Hence, users must access and update the database in a controlled manner. This is enforced by a concurrency control mechanism. Without concurrency control, multiple users might try to access and update the same data at the same time, leaving it in an incorrect, or *inconsistent*, state. When multiple users use the same database, it may be the case that some users are only authorized to access certain parts of the database. A database system should, therefore, provide means of protecting the database from unauthorized access: a security and authorization subsystem.

A single database state change may involve several operations. For example, transferring a sum of money from one bank account to another involves subtracting the sum from one account and adding it to another. Conceptually, a state change like this is seen as a single operation. It is therefore desirable to define a construct that encapsulates database operations in a larger unit. This construct is the *transaction*, which forms the basis for fault tolerance and recoverability, and is an important concept in concurrency control in database systems.

Consider again the example of a sum of money being transferred from one bank account to another. It is possible to identify certain properties we want such a transaction to display. Firstly, we want the transaction to execute completely or not at all, i.e. we do not want the database to end up in a state where money has been withdrawn from one account but not deposited to the other one. Secondly, we do not want a transaction to see an intermediate state caused by a concurrently executing transaction, e.g. no transaction should see a state where money has been withdrawn

from one account but not deposited to the other. Thirdly, we want every transaction which causes a state change in the database to leave it in a correct state, i.e. a transaction should never cause the database to end up in an incorrect state. Finally, the state changes made by a transaction should not be lost, even after a software or hardware failure.

These transaction properties are often called the *ACID properties* [GR93, EN94]. Gray and Reuter define the ACID properties as follows [GR93, pp 6]:

Atomicity: A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

Consistency: A transaction is a correct transformation of the state. The actions taken as a group do no violate any of the integrity constraints associated with the state. This requires that the transaction is a correct program.

Isolation: Even though transactions execute concurrently, it appears to each transaction, T, that others executed before T or after T, but not both.

Durability: Once a transaction completes successfully (commits), its changes to the state survive failures.

As stated in the definition for durability, a transaction is said to *commit* when it completes successfully. If a transaction does not commit, it *aborts*. Figure 2 shows the states a transaction can visit during its execution. This figure is based on figure 17.4 from Elmasri and Navathe [EN94, pp 535].

As previously stated, transactions are important to fault tolerance in database systems. In fact, Connolly et al. [CBS98] state that transactions are "...the basic

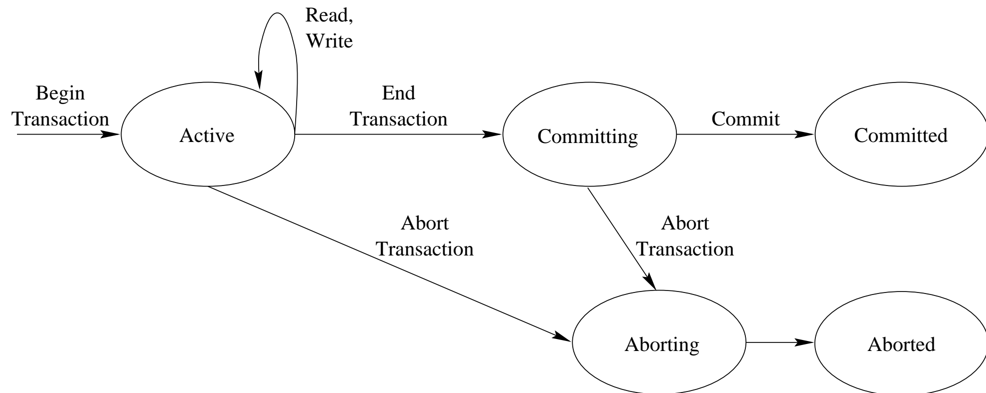

Figure 2: State transition diagram for transaction execution.

unit of recovery in a database system." A *recovery mechanism* ensures that transaction atomicity and durability hold in the presence of failures. That is, the recovery mechanism must make sure that the database is in a consistent state at all times. Hsu and Kumar [HK98] define a *consistent database state* as "... a database state in which all changes made by committed transactions are installed while none of the changes made by uncommitted transactions are installed." Given these definitions of recovery mechanism and consistent database state we can now define the term *database recovery* as the work carried out by the recovery mechanism to restore the database to a consistent state in the event of failure [CBS98].

To make sure that database recovery is always possible, the recovery mechanism must ensure that the database is in a resilient state at all times. Hsu and Kumar [HK98] define a *resilient database state* as a "... database state from which a consistent database state can be constructed." A database can be kept in a resilient state by recording information about transactions which allows the recovery mechanism to undo all changes made by uncommitted transactions and redo changes made by committed transactions.

The extent of database recovery depends on the failure which has occurred. When a single transaction fails before commit, it is sufficient to undo updates made by that transaction. When the entire database system or a node in a distributed database crashes multiple transactions may have to be undone or redone. In this work, we focus on the case when a node in a distributed database crashes.

All the things that have been discussed in this section so far, help in reducing application development time. Once a database system is operational, developing an application which uses the database is a much quicker process than if the application had directly used files for storing data. In fact Elmasri and Navathe state:

> Development time using a DBMS is estimated to be one-sixth to one-fourth of that for a traditional file system. [EN94, pp 16]

2.1.2 Real-Time Issues

In this section, we define the terms real-time system and real-time database system. We also discuss why non-real-time databases, even fast ones, are not suitable for real-time processing.

Many definitions of the term real-time system exist. Most of them, however, have two things in common. They state that a *real-time system* interacts with the environment and that the correctness of a real-time system depends not only on the logical correctness of an output but also on the time of output. Burns and Wellings state that:

> ...the correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced. [BW96, pp 2]

Burns and Wellings also quote the following definition from the Predictably Dependable Computer Systems (PDCS) project:

> A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment. [RLKL95]

The previous discussion and definitions also apply to real-time database systems. Non-real-time database systems are usually designed to minimize the average response time of transactions. As shown by Stankovic et al. [SSH99], this is not sufficient for real-time processing, i.e., high execution speeds do not make a database system suitable for real-time processing. Stankovic et al. state that:

> ... real-time databases aim to meet the timing constraints and data-validity requirements of individual transactions and also keep the database current via proper update rates. [SSH99]

This means that time-cognizant protocols are needed in real-time database systems. Furthermore, it is argued that modifying existing non-real-time databases in order to squeeze in real-time capabilities is not a good approach, that is, real-time database systems should be designed and implemented from the ground up with real-time processing in mind.

Timeliness is an important concept in real-time systems. *Timeliness* implies that a system meets all required deadlines [Mel98]. This requires that each task is predictable and sufficiently efficient. *Predictability* implies that there is an upper bound on the resource requirements, in particular processor time requirements. *Sufficient efficiency* implies that the complete task load is schedulable.

An important problem in real-time databases is the unpredictable read and write times of disks. This must be eliminated if a real-time database is to be able to guarantee transaction timeliness. There are two ways in which this can be done, by making the use of disks predictable, or eliminating disks from the system. The first approach, mentioned by Stankovic et al. [SSH99], requires time-cognizant disk scheduling algorithms. In this work, we assume the second approach, taken by Andler et al. [AHE+96] in the DeeDS distributed real-time database system. When disks are eliminated from the system, the entire database must be placed in main-memory. The implications of this are discussed in section 2.1.3.

Song et al. [SKR+99] have suggested a recovery mechanism for real-time main-memory database systems. This mechanism utilizes non-volatile memory to store database updates before they are written to disk. It is claimed that this mechanism is more suitable to real-time database systems than previous main-memory database recovery mechanisms since it guarantees high-performance and low interference to normal transaction processing.

An example of a real-time database is one which reads values from several sensors on a production line and makes adjustments to the line according to these values. Each sensor is equipped with a computer which gathers data and shares it with other computers on the production line. The system may need to monitor the arrival rate of raw materials for the production and the state of the equipment on the production line. Given all this data, the real-time database system has to control the production speed and notify operators whenever the arrival rate of raw materials drops below some point or if some equipment fails. This example illustrates yet another typical property of real-time systems and real-time database systems, i.e. they are often distributed by nature [AHE+96]. The implications of this are discussed further in

section 2.1.4.

2.1.3 Main-Memory Database Systems

Gruenwald et al. [GHD+96] state that in a *main-memory database system* the entire database or a large portion of it is placed in main memory. When the entire database resides in main-memory it is sometimes referred to as a *main-memory resident database*. Throughout this work, we use the two terms interchangeably. Unless explicitly stated, we assume that the entire database resides in main-memory when we refer to main-memory database systems or main-memory databases.

Main-memory database systems were originally aimed at applications requiring high throughput and fast response times [GHD+96]. This was because accessing data in main-memory is orders of magnitude faster than accessing data on disk (no disk I/O is required when accessing data in a main-memory database). But, as stated by Stankovic et al. [SSH99], real-time computing is not the same as fast computing, so why implement a real-time database as a main-memory database? As mentioned in section 2.1.2, it is necessary to eliminate the unpredictability caused by disks in real-time database systems. One way to do this, without requiring special time cognizant disk scheduling algorithms, is to avoid disks, i.e. by placing the database in main-memory. Another reason to store a real-time database in main-memory is to achieve sufficient efficiency, i.e., the real-time database system must be capable of executing transactions fast enough to meet deadlines.

Recovery mechanisms from disk-based database systems are not suitable for main-memory database systems since they involve too much overhead during normal operations and therefore impede the overall performance of the system [JSS98]. For this reason, different recovery techniques have been implemented for main-memory

database systems. These are discussed in section 2.2.

2.1.4 Distribution Issues

This section starts with a definition of the term distributed system. After that, we discuss distributed databases, what is meant by the term distributed database, and the reasons for distributing databases. We end this section with a discussion about distributed real-time databases by looking at which additional problems distribution brings to real-time processing and why it is interesting to look at distribution in real-time database systems.

Schroeder [Sch93, pp 1] defines a *distributed system* as "...several computers doing something together." From this, quite general, definition Schroeder identifies three primary characteristics of distributed systems.

Multiple computers: A distributed system consists of multiple physical computers, each with their own processor, memory, I/O channels etc.

Interconnections: The individual computers in a distributed system are interconnected.

Shared state: The computers in a distributed system are cooperating towards a common goal and have a shared state which describes the entire distributed system.

As indicated by Garcia-Molina and Hsu [GMH95], the driving force behind the development of distributed non-real-time databases has been the desire to bring together data from multiple sources. In contrast, real-time database systems can be distributed due to environmental requirements [AHE+96].

The distributed real-time database system assumed in this work is a homogeneous system which is distributed because of requirements from the environment. We are not considering the type of distributed database system which is implemented on top of existing, more or less autonomous, databases in order to access their data as if it existed in a single database.

Andler et al. [AHE+96] have suggested a distributed real-time database management system architecture named DeeDS (Distributed Active Real-Time Database System). This is the architecture assumed in this work, a choice which is explained in the following chapters. A distributed real-time database system is still a real-time system, and should be able to guarantee timeliness. In order to achieve timeliness, unpredictable network delays must be eliminated from real-time processing. The approach taken in DeeDS is to make sure that real-time transactions do not access remote nodes. Another possible approach is to use a real-time network. An advantage of the approach take in the DeeDS system is that it works for common non-real-time networks.

Two assumptions must be made if a transaction is to run without accessing a remote node. Firstly, all data required by the transaction must be present locally and, secondly, the transaction must be able to commit locally. As described by Andler et al. [AHE+96], the first issue can be solved by assuming a *fully replicated database*, i.e. a system where every node holds a complete copy of the database. In order to allow transactions to commit locally, Andler et al. replicate updates made by a transaction after the transaction commits. This requires *conflict detection and resolution algorithms* [Lun97] which detect conflicting updates coming from different nodes and resolve these. Both ASAP (as-soon-as-possible) replication and bounded delay replication can be used to replicate transaction updates after commit. *ASAP*

replication replicates the updates at the first opportunity, but does not give any guarantees about when replication happens. *Bounded delay replication* gives an upper bound on the time it takes to replicate transaction updates.

2.2 Main-Memory Database Recovery

The ability to recover from failures is extremely important in database systems. In this section, we start by describing which properties of main-memory database systems are important when recovery is being considered. Then we discuss why recovery is an important issue and present the most important recovery terminology. We briefly discuss how recovery is done in disk-based databases and then describe why recovery mechanisms from disk-based database systems are not good for main-memory database systems. Finally, we discuss recovery methods better suited to main-memory database systems.

As stated in section 2.1.3, the primary copy of a main-memory database resides in volatile main-memory. This means that when the database system crashes, the primary copy is lost and has to be reconstructed. In contrast, a disk-based database is usually not lost in a crash but can be inconsistent and out-of-date after restart. Also, as stated by Gruenwald et al. [GHD$^+$96], main-memory database systems are often targeted for high-throughput applications. Recovery is the only part of these systems which can require disk I/O. Therefore, recovery must be designed in such a way that it does not become a bottleneck in the system.

2.2.1 General Recovery

A primary task of a database recovery mechanism is to ensure that atomicity and durability hold when failures occur. The last of the ACID properties, durability, states that updates made by committed transactions should survive failures, including a database system crash. This is ensured by the log. Gray and Reuter [GR93] describe the *log* as a sequence of log records, where each *log record* describes an update to the database.[1] Using the log, every database update can be both redone and undone. After restart, all committed transactions which are not reflected in the database can be replayed from the log, thus ensuring that durability is maintained.

Similarly, atomicity is ensured using the log. If the effects of an uncommitted transaction are reflected in the database after restart, these need to be undone. This is done with help from the log. Undoing uncommitted transactions is necessary since transaction execution must be atomic, i.e. transactions should execute completely or not at all.

For undo and redo to be possible after restart, the database system must follow the following rules [GR93, sec. 10.3.7]:

Write-ahead log (WAL): Before a database page is updated on disk, all log records concerning that page must be flushed to stable storage. This ensures that uncommitted updates can be undone after restart.

Force-log-at-commit: Before a transaction commits, its log pages must be written to stable storage. This ensures that committed transactions can be redone after restart.

[1] Actually, this is a simplification, there are log records which describe other things than updates to the database, e.g. records used to guide recovery.

In theory, the log is all we need in order to do recovery. The log contains records of every update that has ever been made to the database. By replaying the entire log the most recent consistent database state can be recreated [GR93]. The problem is that the log can get very large, a single database update can cause several log records to be written and in a system with a high transaction-rate or a system that has been running for a long time the log can be huge. Therefore, it would take too long to process the entire log after every restart and in many cases it is not realistic to maintain all log records indefinitely. This is why databases are checkpointed. The term checkpoint can mean different things for different systems, but the basic idea is always the same. A *checkpoint* is used to reduce the amount of log records which need to be processed during restart. Some disk-based database systems do this by writing a log record stating which committed transactions need to be redone during restart. Main-memory database systems need to do more work than this. They need to record the entire database to stable memory, or at least the parts of it which have changed since the last checkpoint. A checkpoint in a main-memory database system could be a complete copy of the database written to disk.

Logging can be done is different ways. The most important of these are logical logging and physical logging [GHD+96]. In *logical logging*, operations carried out on the database are logged. This has the benefits of producing a relatively small log, but as described by Gruenwald et al. and Gray and Reuter [GR93], logical logging is more complex to deal with than physical logging. In *physical logging*, database states are recorded in the log, as opposed to logical logging which records state changes. When a database page is updated, its state before and after the update are recorded in the physical log.

2.2.2 Inadequacy of Existing Approaches from Disk-Based Systems

When a recovery mechanism is designed, the types of failures that must be anticipated are transaction, system, and media failures [Eic87]. Table 1 shows what needs to be done in database systems in order to recover from these failures.

Failure Type	**Recovery Operations Required**	
	Traditional DBMS	**MMDB**
Transaction Failure	Transaction UNDO	Transaction UNDO
System Failure	Global UNDO Partial REDO	Global REDO
Media Failure	Global REDO	Global REDO Partial REDO

Table 1: Database recovery operations [Eic87].

Eich [Eic87] describes a *transaction failure* as occurring when a transaction is unable to commit. This is the most common of the three failure types and requires that any updates made by the failing transaction are undone. This is done in much the same way in disk-resident and main-memory database systems, the only difference is that it is very important in main-memory database systems that the log records required for transaction undo are in main-memory. Furthermore, Eich states that undoing a transaction should take roughly as long as it would have taken the transaction to complete successfully. Hence, disk I/O must be avoided during transaction undo in a main-memory database system.

In this work, we focus on single-node failures in a fully replicated, distributed main-memory database system. The *single-node failure* assumptions means that a crashed node has time to recover before any other node crashes. A node failure

resembles a *system failure* in a centralized system as described by Eich [Eic87]. That is, the failing node loses the entire database contents and a global redo must be performed, i.e. the entire database copy on the crashed node needs to be rebuilt (after a single-node crash the entire database may still exist in the rest of the system, while after a system crash in a centralized system the database only exists as a secondary copy). This is the major difference between recovery in main-memory databases and disk-based databases. As shown in table 1, a disk-based database system needs to do a global undo and a partial redo after a system failure, i.e. all uncommitted transactions need to be undone and committed transactions which are not reflected in the database need to be redone. In a main-memory database no undo needs to be done, since the effects of uncommitted transactions are lost when the entire database contents disappear.[2]

In main-memory databases, a global redo is performed by loading an archive copy of the database into memory and then processing the log as necessary to bring the database up-to-date [Eic87]. Note that the archive copy exists purely for recovery purposes and is never read during normal processing. In contrast, the working copy of a disk-based database resides on disk and needs to be brought up-to-date after restart.

A rule of thumb is that recovery after a system failure should take a comparable amount of time as it would have taken to successfully complete all transactions active at the time of failure [Eic87]. Since system recovery in a main-memory database requires disk I/O this is difficult. In order to make sure that restart is as fast as possible, the database archive copy needs to be updated frequently so that a minimal amount of the log needs to be processed once the archive copy has been loaded into

[2]This is not entirely true, when fuzzy checkpointing is used uncommitted transactions which need to be undone can be reflected in the checkpoint. Fuzzy checkpointing is described in section 2.2.3.

memory. For this not to be a bottleneck in the system, archiving or *checkpointing*, a main-memory database should interfere as little as possible with other transaction processing.

Eich [Eic87] presents the following wish list for main-memory database recovery:

1. No disk I/O required to accomplish transaction undo.
2. Frequent checkpoints performed with minimum impact on transaction processing.
3. Asynchronous processing of log I/O and transaction processing.

Since we are focusing on single node recovery in a fully replicated, distributed main-memory database system not all of these are relevant to our work. We identify the following wish list for our recovery mechanism:

1. Checkpointing should have a minimum impact on transaction processing.
2. After recovery, a restarted node should have a consistent, up-to-date copy of the database.

2.2.3 Improved Recovery Approaches

Several approaches have been proposed for checkpointing main-memory databases. Gruenwald et al. [GHD+96] classify these as fuzzy checkpointing, non-fuzzy checkpointing, and log-driven checkpointing.

In this work, we use fuzzy checkpointing as the basis for our approach. Gruenwald et al. [GHD+96] state that fuzzy checkpointing is the most popular checkpointing method for main-memory databases. This popularity stems from the fact that fuzzy

checkpointing interferes the least with other processing compared to other approaches. We describe fuzzy checkpointing in detail later on, but in short it works as follows. The entire database, or those database pages that have been updated since the last checkpoint, is written page-by-page to disk without any regard to locks. This means that the database is checkpointed without any transactions being blocked by locks held by the checkpointer. However, the checkpoint created can be inconsistent, it may reflect partially executed transactions, and contain partial updates. After the checkpoint is loaded into memory, the log must therefore be used to bring the checkpoint up-to-date and make it consistent. This means redoing and possibly undoing transactions.

Non-fuzzy checkpointing schemes take locks on the data objects being written to disk, thereby creating an action-consistent or transaction-consistent checkpoint. An *action-consistent* checkpoint contains no partially executed updates, but can reflect partially executed transactions. A *transaction-consistent* checkpoint reflects only committed transactions. The problem with the non-fuzzy approaches is that they increase data contention in the database and can incur considerable overhead [GHD+96].

Log-driven checkpointing assumes that a previous checkpoint of the database exists on disk. As described by Gruenwald et al. [GHD+96], the log is applied to the existing checkpoint to bring it up-to-date. In our work, the recovering node does not have any copy of the database and this approach is therefore not applicable to our work.

Fuzzy checkpointing is best implemented with physical logging since database pages in a fuzzy checkpoint can be partially updated and inconsistent. This is why Gruenwald et al. [GHD+96] state that physical logging is usually recommended for main-memory databases. This can be contrasted to disk-based database systems

where logical logging is usually recommended since it requires less space.

Fuzzy checkpointing has been researched for several years. Since Hagmann [Hag86] first suggested fuzzy checkpointing in 1986, it has been optimized to reduce restart time. Two interesting approaches by Dunham et al. [DLL98] divide the database into segments in order to speed up restart. The first of these approaches, dynamic segmenting fuzzy checkpointing (DSFC), dynamically divides the database into differently sized segments based on database access patterns. These segments are then checkpointed in a round-robin fashion. The second approach, partition checkpointing (PC), assumes that the database has been divided into sections in advance. The sections are then checkpointed with a frequency proportional to their update frequency. Both of these approaches aim to minimize the amount of log data which must be processed after the checkpoint has been loaded into memory.

Chapter 3

The Distributed Real-Time Recovery Problem

We are concerned with developing a recovery mechanism for distributed real-time database systems. We assume that the database is fully replicated and that only single-node failures can occur. The problems we focus on are how a restarted node can build a consistent database view, and how durability can be guaranteed for locally committed transactions.

The reasons for carrying out this work are outlined in section 3.1. In section 3.2 the assumed system model is presented. An overall view of problems in distributed recovery is given in section 3.3. In sections 3.4 and 3.5 these problems are discussed in more detail.

3.1 Motivation for Distributed Recovery

It is often desirable to reduce the number of hardware components in a real-time system, due to cost and environmental factors. Since real-time systems are frequently

integrated in mass-produced products, such as cars, cost can be an important issue. Huge savings can be attained by reducing hardware cost by a small amount per unit produced. Also, real-time systems are sometimes used in environments that do not allow certain types of hardware. For example, a missile guidance system can be exposed to heavy vibrations. It is desirable to eliminate disks from such a system since disks do not tolerate vibrations well. The environment can also limit the physical size of a computer system and, thus, the possible number of components.

Eliminating disks can also be beneficial to real-time processing, since by making a database main-memory resident, unpredictability and pessimistic worst case execution times caused by disks are avoided [AHE+96]. In addition, by enforcing full replication and eventual consistency, distributed real-time databases can guarantee local timeliness and avoid the need for real-time networks. By using the inherent data redundancy in fully replicated distributed databases, it may be possible to avoid the involvement of disks in recovery processing. If disks are neither needed for storing a database nor for recovery processing, then it should not be necessary to equip every node in a distributed real-time database system with a disk.

Our hypothesis is that by retrieving data from a remote node during recovery instead of from disk, it is possible to avoid disks in recovery processing in fully replicated distributed databases. The entire database can be retrieved from a healthy node, since every node holds a complete copy of the database [AHE+96]. If we assume that all the nodes cannot crash at the same time, then a complete copy of the database always exists in the system.

As opposed to our recovery approach (fig. 3b), traditional database recovery mechanisms rely on redundant data written to stable storage (fig. 3a). This principle has been described by, for example, Hsu and Kumar [HK98] and Haerder and Reuter

[HR98], and it is the same whether a disk-based or a main-memory database is used. All database updates are logged, and this log can be used to retrace all changes that have been made to the database. The database is also checkpointed in order to limit the amount of log information that needs to be processed during recovery.

We do not know of any previous attempts to perform distributed recovery by reading a database from an arbitrary node. In work by Treiber and Burkes [TB95], a leader/follower model is used, in which a restarted node retrieves the database from the current leader. Their work, however, assumes pairs of cooperating leader/follower nodes in which data is replicated for recovery purposes only. In our case, it is assumed that the node supplying the database is chosen during recovery, and that the database is fully replicated in order to facilitate real-time processing.

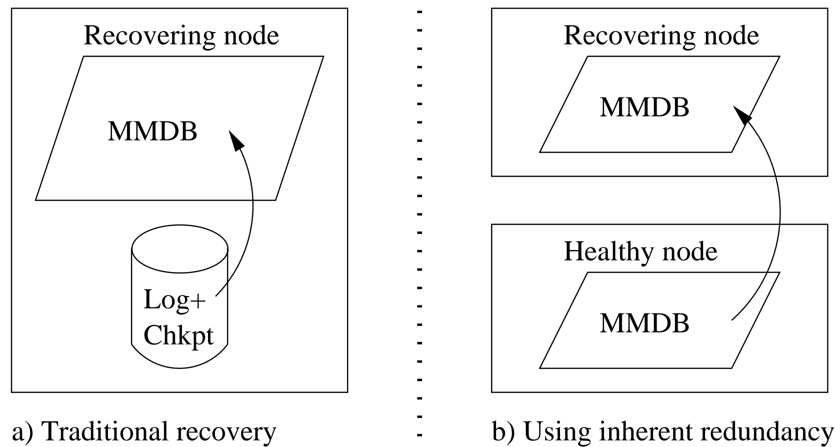

Figure 3: *a)* Traditionally, recovery processing uses redundant data on disk. *b)* In a fully replicated distributed database redundancy is inherent.

An example of a fully replicated distributed main-memory resident database management system is the Distributed Active Real-Time Database System (DeeDS) developed at the University of Skövde, Sweden [AHE+96]. DeeDS is a distributed real-time

database system which implements full replication to facilitate real-time processing. Also, DeeDS is implemented as a main-memory database in order to avoid the adverse effects disks can have on real-time processing.

To summarize, it may be possible to utilize the inherent data redundancy in fully replicated databases for recovery purposes. Then, it should be possible to avoid equipping every node in a distributed real-time database system with a disk, thus reducing the amount of hardware needed by the system. Eliminating disks from a real-time system is often beneficial since disks can be a source of unpredictability or pessimistic worst case execution times. Also, disks are not suitable for certain environments and eliminating hardware saves money.

3.2 Distributed Real-Time Database Management System Model

This section is a description of the assumed distributed real-time database management system model. We start with assumptions about the distributed system as a whole and then describe assumptions about individual nodes.

In a distributed database system, the database resides on more than one node. Since we are dealing with real-time requirements, the system has to be able to guarantee timely execution of transactions. In order to facilitate timely transaction execution, it is desirable to eliminate the need for an executing transaction to access the database at remote nodes [AHE+96]. Each node has to hold a complete copy of the database, or at least a copy of all the data it will ever need to access. A distributed database is said to be *fully replicated* when every node holds a complete copy of the database.

Assumption 1 *The database is fully replicated.*

Since our database model enables transactions to be executed without accessing data at remote nodes, it is a logical next step to allow transactions to commit locally [AHE+96]. Allowing transactions to execute and commit locally makes real-time processing possible in the absence of a real-time network. To allow this, we must make the following assumption:

Assumption 2 *Only eventual consistency is guaranteed.*

After a transaction commits, assumption 2 guarantees that the effects of that transaction will eventually be replicated to the entire system. In other words, a node informs other nodes of an update only after it has committed the transaction making the update and temporary inconsistencies may occur.

It is necessary to detect and resolve conflicting updates, since different nodes can simultaneously execute transactions that modify the same part of the database [AHE+96]. Therefore, assumption 3 is made.

Assumption 3 *Nodes can detect and resolve conflicts in replicated updates.*

Assumptions 4 and 5 deal with the way in which the system fails. Since the main focus of this work is on recovering individual nodes, the failure model of the system is limited to single-node failures. It is reasonable to make the following assumption since, as stated by Verissimo and Kopetz [VK93], multiple failures are highly unlikely to occur within a single recovery interval.

Assumption 4 *Only single-node failures can occur.*

The effect of assumption 4 is that only one node can fail at any given time. Also, after a node fails, it has time to restart and get a consistent database view before

any other node fails. In addition, we also assume that nodes do not start sending out incorrect data when they fail.

Assumption 5 *Nodes are fail-silent.*

The fact that nodes are fail-silent means that after a node fails it will not transmit any data. This eliminates the need to deal with complex failure scenarios such as Byzantine failures which do not fall within the focus of this work.

The remaining assumptions concern individual nodes. As indicated in section 3.1, it is desirable to avoid disks in real-time database systems. The following assumption, along with assumption 4, enables us to avoid disks.

Assumption 6 *The database is main-memory resident at each node.*

All access to a database should be handled by the database management system. In order to guarantee this the following assumption is made.

Assumption 7 *The database resides in its own address space which is accessed only by the database management system.*

As indicated by Gruenwald et al. [GHD+96], it is common for recovery mechanisms in main-memory database systems to operate at the memory-page level. The following assumption enables our recovery mechanism to work with memory pages rather then higher-level database objects.

Assumption 8 *Meta-data, e.g. indexes, are stored in the database.*

Assumptions 7 and 8 enable us to view the database as a collection of memory pages which reside in an isolated part of main-memory. During database recovery

it is sufficient for a recovering node to copy this part of the memory from a healthy node.

Two main approaches are used for updating databases [EN94, pp 579]. In the first one, *in-place updating* or *update-in-place*, a transaction modifies database pages before commit. This means that it must be possible to undo all modifications when transactions roll back. The other method of updating is called *shadowing* or *shadow paging*. When shadow paging is used, a copy is made of the page that is to be updated. This copy, or *shadow-page*, is then modified. Upon commit the shadow-page replaces the original database page. In this approach, it is sufficient to drop the appropriate shadow-pages when a transaction rolls back, i.e. it is not necessary to do any undo-logging, redo-logging is sufficient. Since shadow-paging on the average requires less logging than update-in-place we assume shadow-paging in our approach.

Assumption 9 *Shadow-paging is used for database updates.*

Since the database pages themselves are never changed, but only exchanged for shadow-pages at commit time, it should be noted that a database only changes when transactions commit. Furthermore, only updates from committed or committing transactions are ever present in a database and partially updated pages do not exist.

In this section, we have provided a model of a distributed real-time database management system. The database is fully replicated and main-memory resident. Single-node failures are assumed. This model serves as a foundation for further discussions about the problems that we focus on and the solutions to these problems.

3.3 Problems in Distributed Recovery

Two problems need to be solved in order to attain diskless distributed recovery. As illustrated in figure 4, it must be clear what happens to locally committed transactions when a crash occurs and it must also be possible for a restarted node to get a consistent database view.

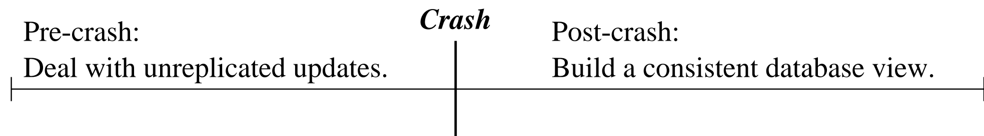

Figure 4: Pre- and post-crash activities in diskless recovery.

Recovery related processing carried out before a crash is aimed at guaranteeing transaction durability and atomicity. It should be possible to redo committed transactions and undo partially executed ones. As we assume that the entire main-memory resident database is lost in a crash, it is not necessary to undo any transactions after restart and guaranteeing durability is our only pre-crash concern.

After a node restarts, its database copy needs to be restored, which implies that a main-memory database needs to be reloaded or rebuilt. Being able to get a consistent database view after restart is the most important goal of our work and is considered before the durability problem. If the problem of restoring a consistent database view is not solved, then it is of no use to have solved the durability problem. On the other hand, it is possible to follow the example of Treiber and Burkes [TB95] and not guarantee durability for locally committed, unreplicated transactions and still build a consistent view of the database after restart.

As described by numerous authors, e.g. [Hag86, Eic86, HK98, HR98, JSS98,

DLL98], these problems have been solved for systems that use disks in recovery processing. Since we want to avoid disks, the traditional disk-based approaches cannot be used, at least not without modifications. The following sections describe the problems presented in this section in more detail.

3.4 Building a Consistent Database View

We want a method of obtaining a consistent database view at a restarted node by loading the database from a remote node. After a crashed node restarts, the database at that node is empty. In order to resume database processing, it is necessary for the node to get a consistent view of the database. Traditionally, this is done by reading a checkpoint from disk and processing log information registered before the crash. As stated earlier, our method should use the inherent data redundancy in the system, rather than reading checkpoints and log information from disk.

Getting a consistent copy of a database from a remote node is not a problem as long as the entire system is quiescent. However, quiescing an entire system interrupts transaction processing. This can have an adverse effect on the ability to guarantee timeliness. Therefore, it is desirable to avoid quiescing a system while a node recovers.

The main problem when retrieving a copy of a database by reading it from a non-quiescent system is that the database is constantly changing. Furthermore, temporal inconsistencies can occur since transactions commit locally (assumption 2).

In short, we want to make a copy of a database in a non-quiescent system while disturbing other database processing as little as possible. This may be possible by using a fuzzy checkpointing algorithm. As described by for example Lin and Dunham [LD96], fuzzy checkpointing can be used to make a copy of a database without locking

any part of it. Thus, a copy can be made with minimal disturbance to other database processing. Since the copy provides an incorrect representation of the database, log information must be used to obtain a consistent database copy. During recovery, a fuzzy checkpointing algorithm should be executed by a healthy node, the *recovery source*. The recovery source should send the checkpoint, plus a log of changes that occurred during the time when the checkpoint was created, to the restarted node, denoted *recovery target*. The recovery target should then use the data from the recovery source to build a consistent database view.

3.5 Guaranteeing Durability for Locally Committed Transactions

We want to avoid disks in distributed real-time database systems. We also want transactions to commit locally before they are replicated to all the nodes in the system. An update that has been committed but not replicated will be lost in a node crash since we assume a main-memory database. This makes it difficult to guarantee durability for committed transactions before they are replicated.

The effects of transactions should be durable [GR93]. Since we want to avoid disks, we cannot guarantee durability by writing data to disk. To guarantee durability, it is necessary to either make sure that each node retains unreplicated changes in spite of a crash, or that changes by committed transactions have been replicated to at least one remote node. Under the assumption of single node failure, it is sufficient for two nodes to be aware of updates to make them durable. If a node is supposed to retain updates in spite of a crash, it is necessary to equip the node with non-volatile memory which is assumed to survive the crash intact. There must be an upper bound on the

size requirements of the buffer.

Chapter 4

Our Approach to Recovery in Distributed Real-Time Database Systems

We have designed a recovery mechanism for distributed real-time database systems. Our mechanism uses fuzzy checkpointing to take a snapshot of a database at a healthy node and transfer it to a recovering node. This is done without locking and, thus, minimizes disturbance to transaction processing at the healthy node. A buddy system or a non-volatile memory buffer are suggested for guaranteeing durability for locally committed transactions.

A brief overview of our mechanism is given in section 4.1. In section 4.2, we give a detailed description of how a recovering node obtains a copy of the database, and in section 4.3, we describe how durability is guaranteed for locally committed transactions.

4.1 Overview

As described in chapter 3, we have divided the distributed recovery problem in two parts. First, we consider how a recovering node obtains a consistent database copy. Second, we consider how to guarantee durability for locally committed transactions.

In our approach, the recovery target starts by selecting a healthy node to act as the recovery source. The recovery target does this by initiating a bid, asking healthy nodes to bid for the role of recovery source. The recovery target then chooses a recovery source based on the answers it receives and notifies the healthy nodes of its decision. The choice of recovery source can be based on, for example, how heavily loaded the nodes are.

Once a recovery source has been chosen, it starts a fuzzy checkpointing algorithm which copies every database memory page to the recovery target which starts rebuilding its own database. The recovery source creates the checkpoint without any locking and, parallel to checkpointing, transaction execution continues. The fuzzy checkpoint received by the recovery target may be inconsistent since the database can be updated during checkpointing.

Logging is used to enable the recovery target to bring the fuzzy checkpoint to a consistent state before it starts executing transactions. While the recovery source runs the fuzzy checkpointing algorithm, it must log all changes to the database and then transmit the log to the recovery target. The recovery target applies the log to its, possibly inconsistent, copy of the fuzzy checkpoint. The recovery target has a locally consistent database when it has received the entire checkpoint and applied the log to it. When a database is *locally consistent*, it is consistent from the local node's point of view. However, eventual consistency may not be guaranteed, since the recovery target may have missed some replicated updates.

Traditionally, checkpointing a database is strictly a pre-crash activity, where the checkpoint is written to disk and all database updates are logged to disk [Hag86]. In contrast, checkpointing is an on-demand post-crash activity in our approach. Also, the checkpoint and the log are sent over a network instead of writing them to disk.

In order to make sure that the recovery target does not miss any replicated updates, the recovery source must forward the replicated updates it receives to the recovery target. This has to be done until it is certain that the recovery target will not miss any replicated updates.

The recovery target starts executing transactions once it has built a locally consistent database, while the recovery source is still forwarding updates. This is possible since eventual consistency is assumed (assumption 2, section 3.2). Figure 5 shows how a checkpoint and a log are sent from the recovery source to the recovery target. The figure also shows how replicated updates must be forwarded from the recovery source to the recovery target.

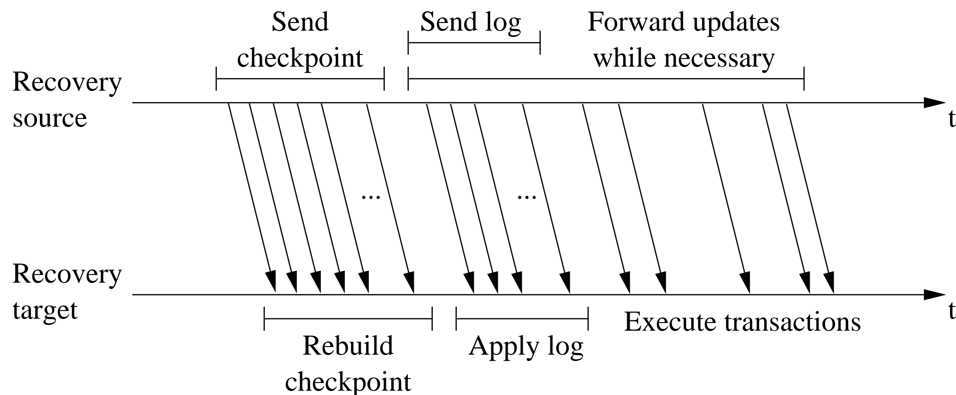

Figure 5: Data sent from the recovery source to the recovery target.

One way of guaranteeing durability for locally committed transactions is by making sure that updates made by a transaction are replicated to one other node during

the transaction's commit phase. This can be compared to the force-log-at-commit rule as described by Gray and Reuter [GR93, pp 557]. Under the force-log-at-commit rule, a transaction's log records are written to disk during commit. Instead of doing this, we send all changes made by a committing transaction to one remote node, the buddy. A *buddy* of node X is a node Y ($X \neq Y$) which is designated to receive updates from X's transactions during their commit phase. If X crashes before replicating some updates to the entire system, Y must replicate the updates instead.

After committing a transaction, a node replicates the updates to all other nodes in the system, including the buddy. When this is complete the node informs the buddy that replication is complete and the buddy knows that no further involvement is required.

When a buddy system is used it is necessary to dedicate network resources to communication between buddies. For transaction execution to be predictable, the time it takes to communicate with the buddy during commit must be predictable.

Another way to guarantee durability for locally committed transactions is to equip every node with a non-volatile memory buffer. Instead of sending updates to a buddy during commit, the updates are written to a non-volatile buffer. When a crashed node restarts, it starts by replicating all updates present in its buffer. A similar approach is often used in main-memory database systems, where a log-tail is stored in a non-volatile buffer [Eic87]. This approach does not require any network resources and can potentially result in shorter worst case execution times than the buddy system approach. The problem with the non-volatile buffer approach is that it does not work for bounded replication, unless the time it takes for a crashed node to restart can be bounded, since the update requests present in the buffer at crash time will not be replicated until the crashed node restarts.

4.2 Building a Consistent Database View

As described in section 4.1, a recovery source is chosen by the recovery target after a bid. Once a recovery source has been chosen it runs a fuzzy checkpointing algorithm to make a copy of the database. This copy is sent to the recovery target along with a log of all changes that occurred during checkpointing. In this section, we discuss the bidding process. We also describe fuzzy checkpointing and motivate why it has been chosen as the basis for our approach. We also give a complete description of our mechanism. In subsection 4.2.1, we show that the mechanism works as intended. In subsection 4.2.2, we discuss which transactions need to be logged in order for the recovery target to get a consistent copy of the database. Finally, in subsection 4.2.3, we consider which replicated updates need to be forwarded from the recovery source to the recovery target in order to make sure that the recovery target does not miss any updates.

Once the recovery target has restarted it requests bids from all other nodes in the system. Through this bid, each node should express its capacity to act as a recovery source. Ramamritham et al. [RSZ89] have used a similar bidding mechanism in distributed task scheduling. The bidding process is illustrated in figure 6.

The recovery target chooses a recovery source based on the replies it gets to the bid. Each node sends a parameter which represents its capacity to act as a recovery source. This parameter can represent, e.g., current processor load, estimated processor load during recovery, or update rates in the database. Another possibility is for the recovery target to ignore this parameter and randomly choose one of the nodes which have answered to the bid as a recovery source.

Fuzzy checkpointing makes a page-by-page copy of the database. It is possible to take a higher-level view of recovery. Instead of viewing the database as a set

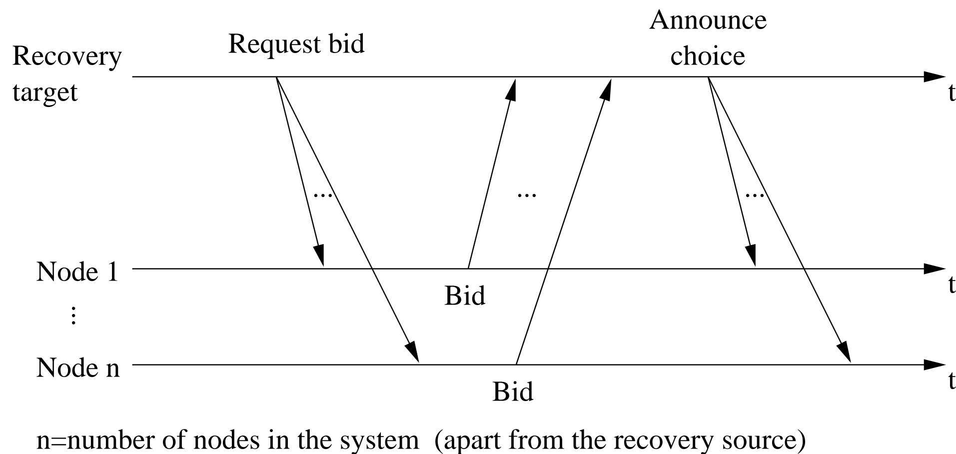

Figure 6: A bidding process is used to select a recovery source.

of memory-pages, it can be viewed as a collection of database objects (objects or relations, indexes etc). In this approach, each database object can be copied independently. This in turn makes it possible to prioritize certain parts of the database during recovery and enable incremental restart, i.e. letting the recovery target start running transactions before it has received the entire database. Also, a log would not have to be explicitly transmitted to the recovery target, since all update requests would be replicated via the eventual consistency mechanism (this assuming that each object is transaction consistent when copied from the recovery source to the recovery target).

The problem with this type of higher-level approach to recovery is locking. Database objects copied from the recovery source have to be identifiable at the recovery target. This requires at least action-consistent copies of the objects (at the very least the object identifier must be intact). If we want to avoid explicitly transmitting a log, we must either assume transaction-consistent copies of database objects, or that the eventual consistency mechanism replicates updates by value and not by operation.

Even if we assume that only action-consistency is required, this still means that locks must be obtained by the recovery process. A higher-level approach similar to the one described here would therefore suffer from the same problems as a non-fuzzy low-level checkpointer, i.e. that data contention in the database is increased which can lead to high overheads and have a severe impact on system performance.

A basic *fuzzy checkpointing* approach writes the database to disk page-by-page, without any regard to concurrency control. As stated earlier, this approach interferes little with normal transactions. On the other hand, the checkpoint created can be inconsistent, i.e. the effects of partially executed transactions and partial updates can occur in the checkpoint. After loading the checkpoint, it is therefore necessary to process the log in order to get a consistent view of the database. Gruenwald et al. [GHD+96] state that fuzzy checkpointing is best implemented with physical logging. In *physical logging*, state changes in the database are logged by keeping before *(BFIM)* and after images *(AFIM)* of changed memory pages. Applying the BFIMs and AFIMs to the database is idempotent since they reflect database states. Note that when shadow-paging is used, as in this work, only AFIMs need to be stored in the log since uncommitted transactions are not represented in a checkpoint.

```
fuzzyCheckpoint()
begin
   mark checkpoint start in log
   for every database page do
        write page to disk
   od
   mark checkpoint end in log
end
```

Algorithm 1: A fuzzy checkpointing algorithm which writes the entire database to disk without regard to locks.

```
loadFuzzyCheckpoint()
begin
   load most recent complete fuzzy checkpoint from disk
   undo uncommitted transactions by applying BFIMs from log
   redo committed transactions as needed by applying AFIMs from log
end
```

Algorithm 2: An algorithm that loads a fuzzy checkpoint and makes it consistent by processing the log.

The logging-after-write (*LAW*) protocol states that the redo log records of an update operation must be flushed to non-volatile memory after the database is updated [DLL98]. If the LAW protocol is followed then the point in the log where forward redo processing should start is the beginning point of the most recent complete checkpoint.

Algorithms 1 and 2 assume that the checkpoint and log reside on disk. In our approach, the checkpoint is never written to disk. Instead, the recovery source sends database pages to the recovery target as the checkpoint is being created. The checkpointing part of our approach is shown in algorithm 3.

```
recoverySourceFuzzyCheckpoint()
begin
   mark checkpoint start in log
   for every database page do
       send page to recovery target
   od
   mark checkpoint end in log
   for every log page from checkpoint start to checkpoint end do
       send page to recovery target
   od
end
```

Algorithm 3: Recovery source fuzzy checkpointing algorithm.

Algorithm 3 is executed by the recovery source. The recovery target has to receive and rebuild the checkpoint, algorithm 4 handles this.

```
rebuildFuzzyCheckpoint()
begin
  for every database page received do
      insert page in memory
  od
  //fuzzy checkpoint rebuilt
  for every log page received do
      insert log page in memory
  od
  //the database is now consistent
end
```

Algorithm 4: Recovery target rebuild algorithm.

Algorithms 3 and 4 are used as a basis for constructing complete recovery algorithms for both the recovery source and the recovery target. As stated earlier, the recovery target algorithm must first choose a recovery source and then build a consistent database view, based on the information it receives from the recovery source. The recovery source algorithm, on the other hand, must participate in the recovery source bid, and at the node that is chosen, start sending the database and log to the recovery target. Algorithms 5 and 6 are the complete versions of 3 and 4. For the recovery target to be able to properly reconstruct the fuzzy checkpoint, it is assumed that the a *page* sent from the recovery source is a tuple consisting of a page id and page contents ($page =< id, contents >$).

```
recoveryTarget(rtID)
begin
   Answers := ∅;
   broadcast(requestForBid : ⟨rtID⟩);
   while ¬(heard from all nodes ∨ timeout) do
         receive(answerToBid : ⟨nodeID, selectionParameter⟩, anyNode);
         if ¬timeout
            Answers := Answers ∪ {(nodeID, selectionParameter)};
         fi
   od
   if timeout ∧ Answers = ∅
      raiseException;
   fi
   rsID := chooseRecoverySource(Answers);
   broadcast(rsChosen : ⟨rsID⟩);
   receive(chkpt : ⟨page⟩, rsID);
   while ¬endOfPages(page) do
         insertToMemory(page);
         receive(chkpt : ⟨page⟩, rsID);
   od
   openPropagatedUpdateQueue();
   receive(log : ⟨page⟩, rsID);
   while ¬endOfPages(page) do
         insertToMemory(page);
         receive(log : ⟨page⟩, rsID);
   od
   startProcessingPropagatedUpdateQueue();
   while ¬(heard from all nodes) do
         messageID := waitUntilNodeHeardFrom(nodeID);
         send(stopForwarding : ⟨nodeID, messageID⟩, rsID);
   od
end
```

Algorithm 5: Recovery algorithm run by a recovery target after restart.

```
recoverySource(myID)
begin
  while true do
      receive(requestForBid : ⟨rtID⟩, anyNode);
      send(answerToBid : ⟨myID, selectionParameter⟩, rtID);
      receive(rsChosen : ⟨rsID⟩, rtID);
      if timeout
        break;
      fi
      if rsID = myID
        startLogging();
        for page in Database  do
            send(chkpt : ⟨page⟩, rtID);
        od
        send(chkpt : ⟨endOfPages⟩, rtID);
        startForwardingReplicatedUpdates(rtID);
        stopLogging();
        for page in Log  do
            send(log : ⟨page⟩, rtID);
        od
        send(log : ⟨endOfPages⟩, rtID);
        while (still forwarding) do
              receive(stopForwarding : ⟨nodeID, messageID⟩, rtID);
              stopForwarding(nodeID, messageID, rtID);
        od
      fi
  od
end
```

Algorithm 6: Recovery algorithm run by healthy nodes.

4.2.1 Checkpointing

In this subsection, we show that by using fuzzy checkpointing combined with logging, a consistent database copy can be obtained from the recovery source by the recovery target. If the database is not updated during checkpointing, then the checkpointing process is not problematic. Every database page is copied from the recovery source to the recovery target, after which the recovery target has a consistent copy of the database (given that the recovery source has a consistent copy). During checkpointing, all database updates need to be logged. We show that all updates to the database leave the log in a correct state. Furthermore, we show that applying the log to a fuzzy checkpoint leaves the recovery target with a consistent database copy.

At the memory page level, the only changes that can be made to a database are page modifications, page deletes, and page inserts. More complex changes such as page splits (as a result of, e.g., hashing) are all constructed using such modifications, deletions, and inserts.

In the discussion that follows, we view the database and the log as sets of memory pages.

Definition 1 *DB is the set of all database pages p_i ($p_i \in DB \mid 0 < i \leq n$).*

Definition 2 *L is the set of all log pages l_i ($l_i \in L \mid 0 < i \leq m$).*

Definition 3 *DB_{rs} is the recovery source's copy of DB.*

Definition 4 *DB_{rt} is the recovery target's copy of DB.*

Multiple versions can exists of each database page, p_i. These versions are denoted $p_i^{(1)}, p_i^{(2)}, \ldots, p_i^{(n)}$. Only one of these versions should be present in the database at any given time. In what follows, we assume that p_i is the version currently in the

database DB, i.e. $\exists j(p_i = p_i^{(j)} \in DB)$. Also, we assume that $p_i^{'}$ is a later version of p_i, present in an updated version $DB^{'}$ of the database, i.e., $\exists k > j(p_i^{'} = p_i^{(k)} \in DB^{'})$.

It is assumed that the recovery source state at the start of checkpointing is $DB_{rs} = \{p_1, p_2, \ldots, p_n\}$; $L = \emptyset$. The recovery target state is $DB_{rt} = \emptyset$

During checkpointing every update to the database is logged. If a page is modified or inserted in DB_{rs}, the after image is inserted in L. If page p_i is deleted from DB_{rs}, then a death certificate, d_i, is inserted to L. Table 2 shows how page modifications, inserts, and deletes are done.

Modify page p_i	**Insert page** $p_i^{'}$	**Delete page** p_i
$DB^{'} = DB \setminus \{p_i\} \cup \{p_i^{'}\}$ $L^{'} = L \setminus \{p_i\} \cup \{p_i^{'}\}$	$DB^{'} = DB \cup \{p_i^{'}\}$ $L^{'} = L \setminus \{d_i\} \cup \{p_i^{'}\}$	$DB^{'} = DB \setminus \{p_i\}$ $L^{'} = L \setminus \{p_i\} \cup \{d_i\}$

Table 2: Possible updates to database DB.

After receiving a checkpoint and a log, the recovery target must apply the log to the checkpoint. Applying a log to a checkpoint is done by individually applying each page in the log to the checkpoint. Table 3 shows how log pages are applied to DB.

Apply $p_i^{'} \in L$ **to** DB	**Apply** $d_i \in L$ **to** DB
$DB^{'} = DB \setminus \{p_i\} \cup \{p_i^{'}\}$	$DB^{'} = DB \setminus \{p_i\}$

Table 3: Applying log pages of the type $p_i^{'}$ and d_i to a database DB.

What needs to be shown is that the recovery target receives a consistent state of DB, i.e., DB_{rt} is identical to DB_{rs} at the end of checkpointing. First, we show that all database updates leave the log in a correct state. A *correct log state* exists if for all modified or inserted p_i the latest version of p_i is in the log and no older version of p_i, nor d_i are in the log. Also, for all deleted p_i, d_i is in the log and no version

of p_i is in the log. We assume that the log starts out in a correct state, this is a valid assumption since the log is initially empty, which is a correct state as long as no updates have been made to the database. Second, we show that if the recovery target receives a fuzzy checkpoint and a correct log from the recovery source, then the recovery target will obtain a consistent copy of the database.

As shown in table 2, an update to page p_i does not affect any $p_j \in L$ or $d_j \in L$ such that $j \neq i$. Table 4 shows that all possible database updates leave the log in a correct state as long as the log is initially correct. After modifying p_i or inserting $p_i^{'}$ only $p_i^{'}$ should be in $L^{'}$ and after deleting p_i only d_i should be in $L^{'}$.

	Modify page p_i	**Insert page** $p_i^{'}$	**Delete page** p_i
Precondition	$p_i \in DB$	$p_i \notin DB$	$p_i \in DB$
Operation	$DB^{'} = DB \setminus \{p_i\} \cup \{p_i^{'}\}$ $L^{'} = L \setminus \{p_i\} \cup \{p_i^{'}\}$	$DB^{'} = DB \cup \{p_i^{'}\}$ $L^{'} = L \setminus \{d_i\} \cup \{p_i^{'}\}$	$DB^{'} = DB \setminus \{p_i\}$ $L^{'} = L \setminus \{p_i\} \cup \{d_i\}$
Postcondition	$p_i^{'} \in DB^{'} \wedge p_i \notin DB^{'}$ $p_i^{'} \in L^{'} \wedge p_i \notin L^{'}$	$p_i^{'} \in DB^{'}$ $p_i^{'} \in L^{'} \wedge d_i \notin L^{'}$	$p_i \notin DB^{'}$ $d_i \in L^{'} \wedge p_i \notin L^{'}$

Table 4: Every database update leaves the log in a correct state.

We have shown that every update made to DB_{rs} during checkpointing will leave the log, L, in a correct state. Now we will show that if a page, p_i, is updated by the recovery source during checkpointing and the update is correctly logged, then the update will be reflected in DB_{rt} after the log is applied to it. Table 5 shows that applying the (correct) log to DB_{rt} will leave DB_{rt} in the same state that DB_{rs} was in at the end of checkpointing. Observe that in the preconditions for modify and insert it is possible to have either p_i, $p_i^{'}$, or neither p_i nor $p_i^{'}$ in the database, but not both. This is due to the fact that any number of operations may have been carried out on p_i before it was checkpointed as well as after it was checkpointed but before the

checkpointing process was done. Thus, p_i may have been inserted-modified-deleted-inserted-deleted, all within a single checkpointing interval. When a page has been updated more than once during a single checkpointing interval then last state should be reflected in the log.

	Modify page p_i	**Insert page** $p_i^{'}$	**Delete page** p_i
Precond.	$p_i^{'} \in L \wedge$ $((p_i \in DB \wedge p_i^{'} \notin DB) \vee$ $(p_i \notin DB \wedge p_i^{'} \in DB) \vee$ $(p_i \notin DB \wedge p_i^{'} \notin DB))$	$p_i^{'} \in L \wedge$ $((p_i \in DB \wedge p_i^{'} \notin DB) \vee$ $(p_i \notin DB \wedge p_i^{'} \in DB) \vee$ $(p_i \notin DB \wedge p_i^{'} \notin DB))$	$d_i \in L$
Op.	$DB^{'} = DB \setminus \{p_i\} \cup \{p_i^{'}\}$	$DB^{'} = DB \setminus \{p_i\} \cup \{p_i^{'}\}$	$DB^{'} = DB \setminus \{p_i\}$
Postcond.	$p_i^{'} \in DB^{'} \wedge p_i \notin DB^{'}$	$p_i^{'} \in DB^{'} \wedge p_i \notin DB^{'}$	$p_i \notin DB^{'}$

Table 5: Applying a correct log to DB leaves DB in a correct state.

We have now shown that the recovery source produces a correct log during checkpointing. We have also shown that if the recovery target receives a fuzzy checkpoint and a correct log, it will obtain a DB_{rt} which is identical to DB_{rs} as it was at the end of checkpointing. This means that our mechanism guarantees that the recovery target receives a consistent copy of the database as long as the recovery source has a consistent database copy.

4.2.2 Logging

In this subsection, we consider which transactions need to be logged by the recovery source in order to enable the recovery target to bring the fuzzy checkpoint to a consistent state.

Figure 7 shows how locally committed transactions can relate to a checkpointing

interval. As shown in the figure, transactions of type A commit before the checkpointing interval starts. Changes made by these transactions are present in the database during checkpointing and will be reflected in the checkpoint. Transactions of types E, H, and I begin their commit phase after the checkpointing interval is complete. None of their updates are therefore reflected in the checkpoint and they have to be replicated to the recovery target by the eventual consistency mechanism.

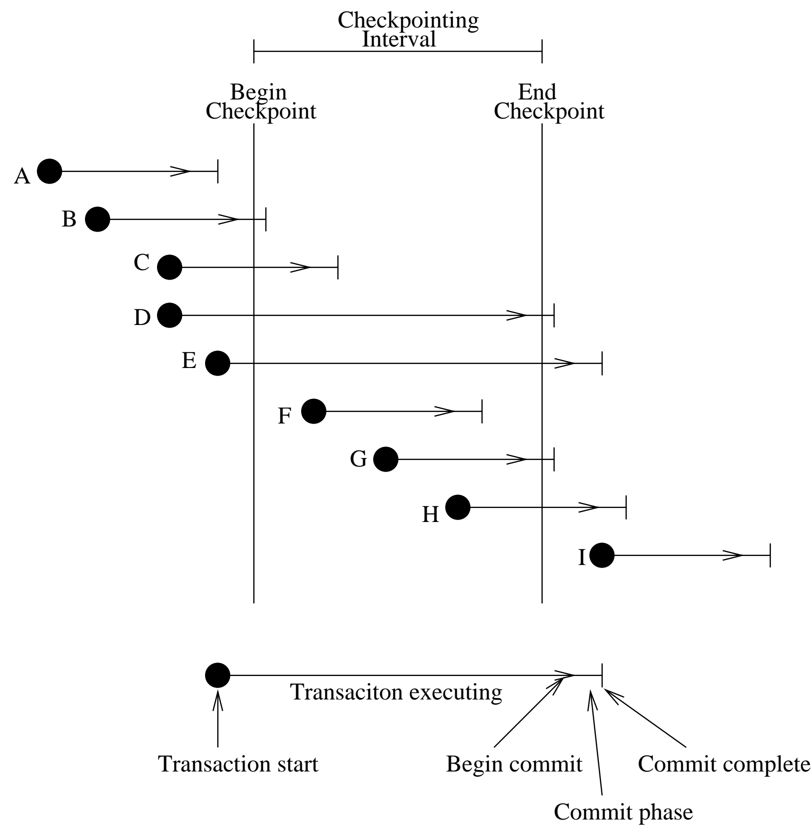

Figure 7: Every possibility in which a locally committed transaction can relate to a checkpointing interval.

Transactions of type B are committing when the checkpointing interval starts. This means that some of their updates may be present in the database at that moment

while others are not. Observe that it is sufficient to start logging changes made by these transactions when the checkpointing interval starts, since changes made before that moment will be reflected in the database.

Changes made by transactions of type C and F need to be logged, since they can be partially reflected in the checkpoint. When one of these transactions commits it can alter both pages that have been checkpointed and pages that have not been checkpointed.

Transactions of types D and G can also change both pages that have and have not been checkpointed. Hence, it is necessary to log these transactions until they have completed their commit, even if that is after the checkpointing interval is complete. It is assumed that only one transaction can commit at any given time. This is necessary in order to make sure that meta-data, such as hash tables, is not partially updated by transactions which begin their commit after checkpointing is complete.

In short, all changes made by transactions of type C, D, F, and G need to be logged, even those that are applied to the database after the checkpointing interval is complete. Also, changes made by transactions of type B after the checkpointing interval starts need to be logged.

4.2.3 Replication Forwarding

In this subsection, we consider which replicated updates need to be forwarded from the recovery source to the recovery target.

So far we have not differentiated between updates and update requests. What actually happens when a transaction commits is that the executing node replicates update requests to the rest of the system. These *update requests* describe which updates have been made by the transaction. When a node receives a replicated

update request, the request is put in a queue, waiting to be processed. When an update request is taken from the queue and processed, the database is updated. What is important to notice, it that there is a delay from the time when a node receives an update request and the time when the database is actually updated. This is the duration that the update request resides in the queue. Conflict detection and resolution is a part of processing an update request from the queue. An update has been replicated when its update request has been taken from the queue and processed. In the remainder of this dissertation, we differentiate between updates and update requests.

Each replicated update request reaches the recovery source either before the checkpointing interval starts, during the checkpointing interval, or after it is completed. We assume that the start and end points of a checkpointing interval are atomic. We also assume that receiving a replicated update request is atomic, i.e. the reception of an update request cannot overlap with the start or end of a checkpoint. Therefore, there are three possibilities for how a replicated update request can relate to a checkpointing interval. Since we are dealing with two nodes which receive the updates independently, we have 3^2 ways in which a single replicated update can relate to the recovery source and the recovery target. We denote the possibilities $ru1$ to $ru9$ (see figure 8). The point in time when a checkpointing interval starts is denoted t_{BC} and the time when it ends t_{EC}. Finally, the time when a replicated update reaches the recovery source is denoted t_{RS} and the time when the update reaches the recovery target t_{RT}.

All updates which are applied at the recovery source before t_{EC} will be reflected in either the checkpoint or the log. Therefore, it is unnecessary for the recovery source to forward the requests for these to the recovery target, i.e. we do not need to worry

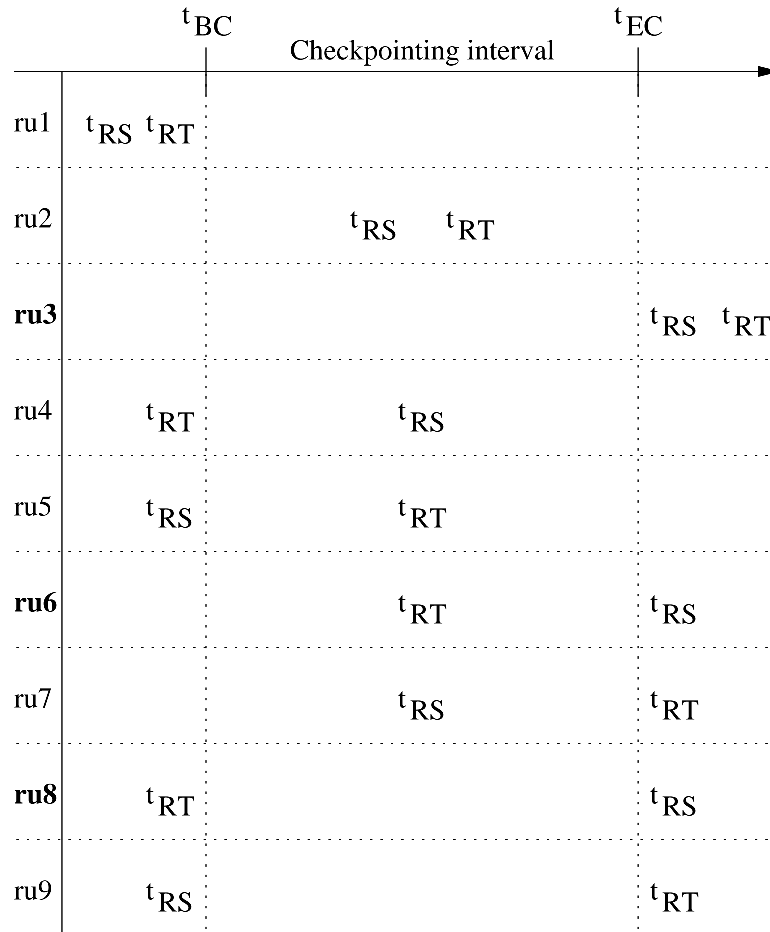

Figure 8: Every way in which t_{RS} and t_{RT} can relate to a checkpointing interval.

about $ru1$, $ru2$, $ru4$, $ru5$, $ru7$ and $ru9$. Also, if the recovery target starts accepting update requests at time t_{EC} we do not need to worry about update requests arriving after that time since they will be applied to the database by the recovery target. Thus, $ru3$ can be added to the list of updates we do not need to worry about.

We are left with $ru6$ and $ru8$. In other words, all replicated update requests which reach the recovery target before t_{EC} and are applied at the recovery source after t_{EC} need to be forwarded to the recovery target by the recovery source. Observe that

it is not possible for the recovery source to know when an update request reaches the recovery target. Therefore, in practice the recovery source needs to forward all update requests it processes after t_{EC} to the recovery target($ru3$, $ru6$, $ru8$). This is the reason why we can assume that t_{EC} is the same point in time at both the recovery source and the recovery target. In reality, there is some time difference between the end of checkpointing at the recovery source and when the last checkpoint page reaches the recovery target. Since the recovery source needs to forward all update requests which it processes after t_{EC} we can ignore this time difference.

We assume that the order of messages between two nodes is preserved and that the recovery target starts to accept replicated update requests to its queue at time t_{EC}. When the recovery target has applied the log to the checkpoint it starts processing update requests from the queue and executing transactions.

We have shown that only update requests that would have arrived at the recovery target before t_{EC} and and are applied at the recovery source after t_{EC} need to be considered. Update requests that are applied at the recovery source after t_{EC} either arrive at the node after t_{EC} or are present in its update queue at time t_{EC}.

Thus, the recovery source needs to send all update requests present in its queue at time t_{EC} to the recovery target. Also, the recovery source needs to forward all replicated update requests, received after t_{EC}, to the recovery target until the recovery target has received at least one message from each node in the system. It can be ensured that no node is quiet for long periods of time by periodically forcing every node to send at a message or by letting the recovery target poll every node. When the recovery target hears from a node for the first time after restart, it logs the message and node identifiers and sends those to the recovery source. The recovery source then knows that it does not have to forward update requests from a certain node, which

have a message identifier that is newer than the one the recovery target received from that node.

4.3 Guaranteeing Durability for Locally Committed Transactions

In this section, we take a closer look at the two methods of guaranteeing durability for locally committed transactions which were identified in section 4.1. We start with the buddy approach and then discuss the non-volatile buffer approach.

The idea behind the buddy approach is to guarantee durability by always having more than one node which knows about the updates a committing transaction is about to perform. Since we assume single-node failures (assumption 4, section 3.2) it is sufficient that two nodes know about the updates that a committing transaction is about to perform. The two nodes are the node executing the transaction and its buddy. This is similar to traditional logging, where the log always knows in advance which updates are about to be carried out [GR93, pp 557].

Figure 9 shows how the buddy system works in the absence of failures. A node executes a transaction and the last thing that happens in the commit phase, i.e., after all local commit processing is done but before the transaction actually commits, is that the buddy is informed of all updates by the transaction. After the buddy has acknowledged that it has received the update requests, the transaction commits and the update requests are replicated to the entire system. Once the replication is complete the buddy is informed and no further involvement is required from it.

If a node crashes after it has informed its buddy of transaction updates, but before it completes replication to the entire system, then the buddy must replicate the update

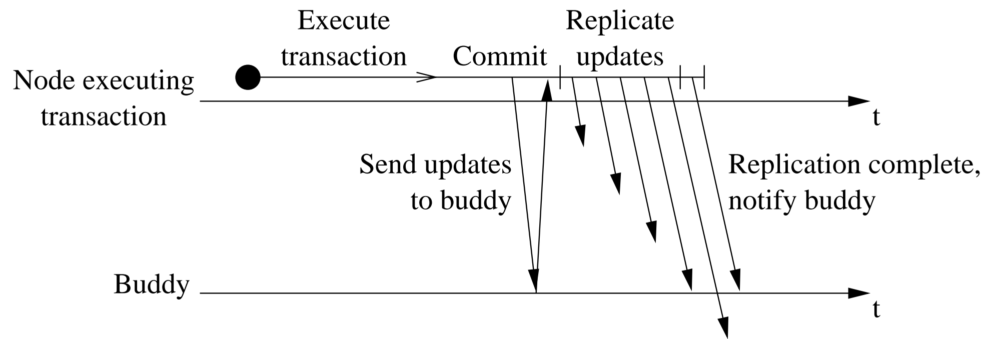

Figure 9: When a transaction commits, updates made by it are sent to a buddy. After the transaction commits, the updates are replicated to every node.

request. This is implemented using a timeout. If the buddy does not get a notification that the replication is complete within the timeout, then the buddy replicates the update request as illustrated in figure 10. Observe that once a node has informed its buddy about transaction updates then the updates will be replicated to the entire system, whether or not the transaction actually commits at the executing node. If the executing node crashes after it has committed the transaction and replicated the update requests to some nodes in the system, then these nodes will receive the same update requests from the buddy when it takes over the replication. These double update requests should be handled by the conflict detection and resolution mechanism at each node.

If the buddy crashes and is unable to acknowledge that it has received the update requests, then the executing transaction should timeout and commit. Once the transaction has committed the executing node then replicates update requests as usual. This is shown in figure 11.

Figure 12 illustrates the case when the buddy crashes after acknowledging that it has received the update requests. In this case, the executing node executes in the

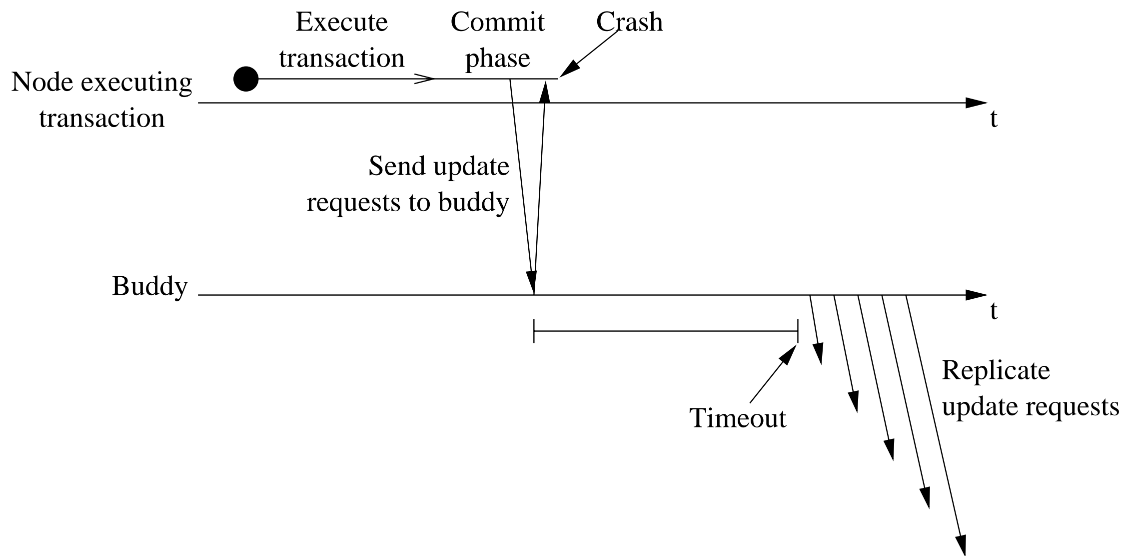

Figure 10: When a node crashes before it has completed replicating the update requests, the buddy must replicate them.

same way as usual, never knowing that the buddy has crashed.

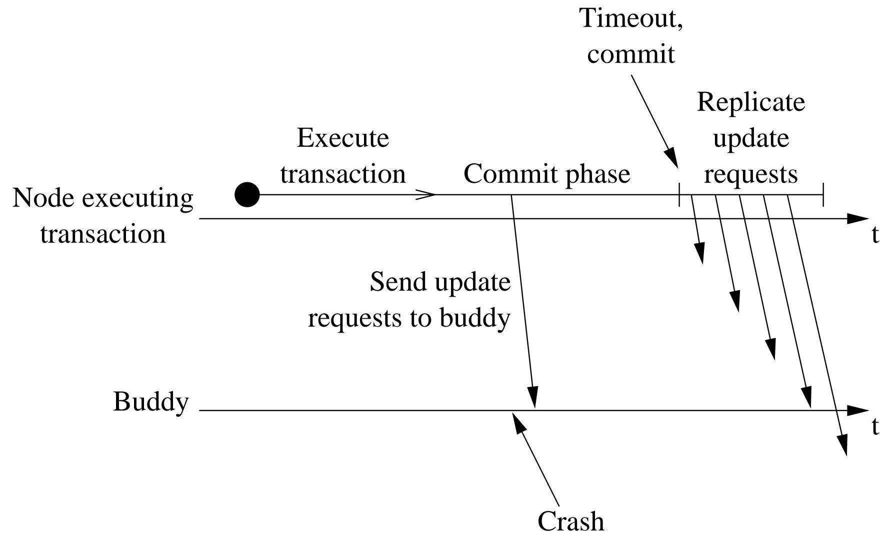

Figure 11: When the buddy crashes during transaction commit, the executing node timeouts and continues transactions processing.

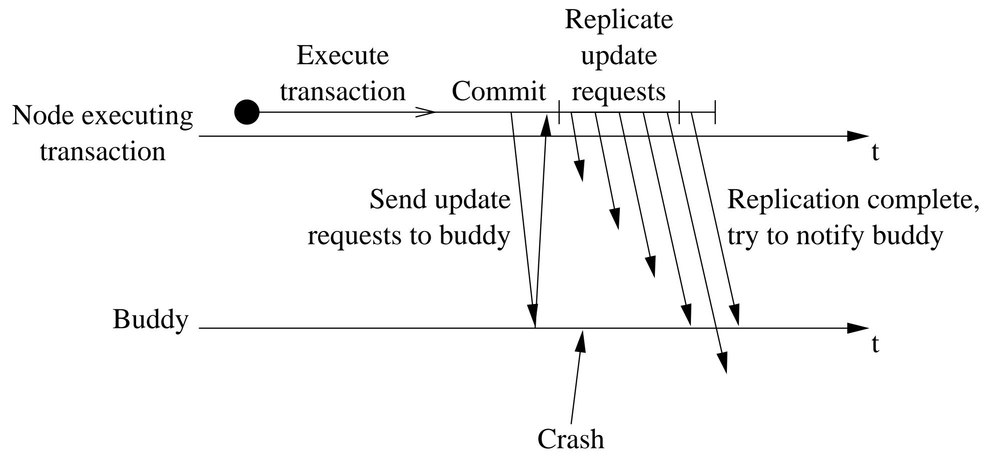

Figure 12: When the buddy crashes after receiving update requests, the executing node replicates the update requests.

Since nodes have to be able to guarantee timeliness for transactions, the time it takes for a node and its buddy to communicate during transaction commit has to be predictable. This means that network resources have to be dedicated to the buddy system. For example, this can be implemented through a real-time network or by having a dedicated link from each node to its buddy. Figure 13 shows how dedicated links can be used to connect buddies.

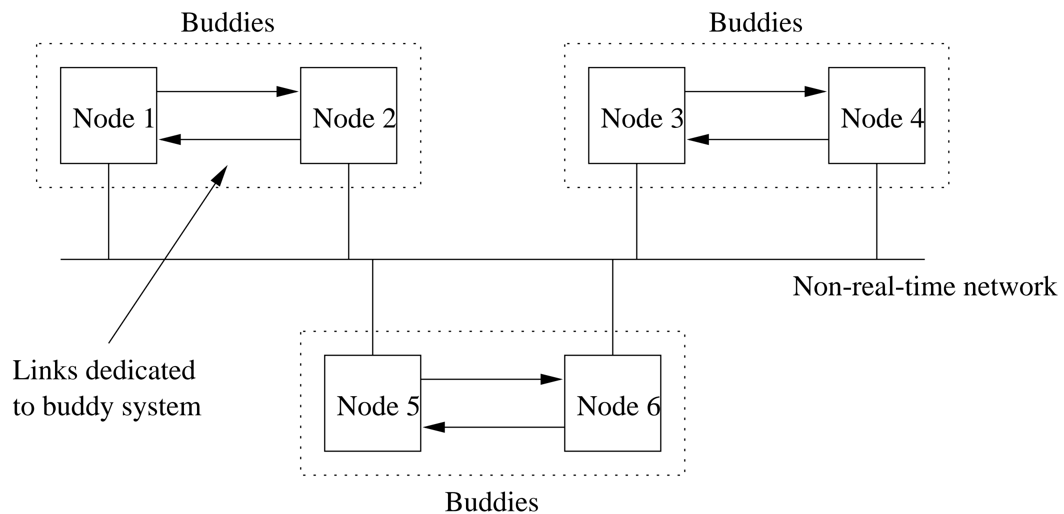

Figure 13: A link dedicated to the buddy system goes from each node to its buddy.

When a buddy system is used, the time it takes for update requests to be replicated can be made predictable since the buddy will replicate them if the node that executed the transaction crashes. This means that the buddy system works under both bounded delay replication and ASAP replication.

The second approach to guaranteeing durability, presented in section 4.1, is to equip every node with a non-volatile memory buffer. Under this approach, every node writes update requests to non-volatile memory as part of commit. Hence, instead of sending the update requests to a buddy as the last part of commit, the update requests

are recorded in non-volatile memory. When a node restarts after a crash, it starts by replicating all update requests present in its non-volatile buffer before starting to rebuild the database. Note that the update requests reach the recovery source and will either be present in the checkpoint or log, or will be forwarded back to the crashed node by the recovery source.

As opposed to the buddy system approach, this approach does not require any dedicated network resources, and since transaction commit is entirely local, worst case transaction execution times should be shorter. For the non-volatile buffer approach to work for bounded delay replication, it must be possible to bound the time it takes for a crashed node to restart, if this is not possible then there is no way of knowing how long it will take before a certain update request is replicated.

For the non-volatile buffer approach to work properly in a real-time system, the size of the buffer must be such that the buffer never fills up. All transaction updates must be written to the buffer and, if it fills up, no transactions can be allowed to commit until space is available in the buffer. This is not acceptable in a real-time system since it can cause critical transactions to be delayed beyond their deadline. Before this approach can be used in a real-system it therefore has to be examined whether the size of the buffer is bounded (at least for critical transactions).

Chapter 5

Evaluation of the Recovery Mechanism

In this chapter, we present an evaluation of the mechanism from chapter 4. We also look at other work which has been done on recovery and compare that with our solution.

This chapter begins with an overview of the evaluation in section 5.1. In sections 5.2 to 5.4, the mechanism is evaluated with respect to timeliness, resource requirements, and the feasibility of implementation. In section 5.5, other work concerning recovery is discussed and contrasted to our work.

5.1 Overview

In this section, we list which attributes are considered in the evaluation, and why these attributes have been selected. Also, we briefly describe the outcome of the evaluation.

We evaluate the recovery mechanism described in chapter 4 with respect to timeliness, resource requirements, and feasibility of implementation. As described in section 2.1.2, timeliness is a fundamental issue in real-time systems. Since the correctness of a real-time system depends on its timeliness, it is important that a recovery mechanism does not violate timeliness. The evaluation indicates that timeliness is attainable under the suggested recovery mechanism.

It is important to be able to predict the resource requirements of the recovery mechanism. This includes, for example, how much memory and processor time is needed. When our algorithms are implemented, we need to know how much resources they require. As described in section 5.3, the recovery mechanism requires memory for the log, and sufficient processor time and network bandwidth to get a checkpoint, log, and necessary update requests to the recovery target.

In section 5.4, we consider whether it is feasible to implement the suggested mechanism in a real system. It is necessary to discuss this since it has not yet been attempted to implement the mechanism. An implementation should not be problematic, since our mechanism is largely based on existing methods, which have been implemented and tested.

5.2 Timeliness

As already stated, it is important for the correctness of a real-time system that timeliness is not violated. In our case, this means that the system has to be designed in such a way that it tolerates the added load caused by recovery. That is, each node should have enough free processor time and memory for recovery, and the network should be able to cope with the additional traffic caused by recovery.

During recovery all nodes, other than the recovery target, should be able to guarantee timeliness for critical transactions. Since transactions execute and commit locally, the timeliness of nodes other than the recovery source and recovery target is not affected by recovery. The recovery source can guarantee the timeliness of critical transactions during recovery, since recovery does not increase data contention in the database. In the recovery source, recovery affects normal transactions only by requiring processor time. If recovery is executed as a non-critical task, it can be preempted in benefit of critical tasks.

Each node should be designed in such a way that it can devote, at least some, processor time to recovery. In our approach, there are no explicit time constraints on recovery, but ensuring that sufficient processing power can be devoted to recovery can help in minimizing the down time of a crashed node. However, it should be noted that the presence of other bottlenecks, i.e. network bandwidth, limit the amount of processor time which can be utilized by recovery.

The recovery target starts executing transactions when it has received the entire database and log, and applied the log to the database. When the recovery target starts executing transactions, it has returned to normal processing and is capable of guaranteeing timeliness for critical transactions. However, the recovery target may be working on old data, since there may be many update requests pending. This does not directly affect local transactions, but many updates may be rejected by the conflict resolution mechanism.

Transaction timeliness should be attainable under both the buddy system approach and the non-volatile buffer approach. As described in chapter 4, the time it takes for a node to communicate with its buddy during transaction commit must be predictable. When this is guaranteed, the communication time must be included in

every transaction's worst case execution time. When the non-volatile buffer approach is taken, extra memory writes are required as a part of commit. The time it takes to perform these, entirely local, memory writes can be made predictable. Since both approaches can be made predictable with respect to time, it is possible to attain timeliness.

In some cases, it may be necessary to use the non-volatile buffer approach in order to reach sufficient performance. Worst case execution times are somewhat longer for all updating transactions under the buddy system approach, since communicating over a network can be assumed to be slower than writing data to non-volatile memory. In fact, for short transactions, the worst case execution time can be increased considerably when the buddy system is used.

5.3 Resource Requirements

The distributed real-time database system must have enough resources to allow recovery. The resources that the recovery mechanism needs are memory, processor time, and network resources.

During recovery, the log can become as big as the database at the beginning of checkpointing plus any pages that are added to be database until the end of checkpointing. Thus, in the worst case, the size of the log can be equal to the size of the database at the end of checkpointing plus the size of log records for every page which has been deleted (see section 4.2.1). Since each page can only occur one in the log as a page image or as a death certificate, the log size is bounded as long as the database size and update rate are bounded. It is possible to optimize logging by only logging changes which occur in the part of the database which has already been checkpointed

(since changes to the part which has not been checkpointed will be reflected in the checkpoint). This does not change the worst case size of the log, but reduces its average size.

Another possible optimization to log processing is to implement the log as an index which points to those database pages which have been updated and holds a death certificate for those that have been deleted. This means that during checkpointing the log only needs memory for this index. However, after checkpointing, when the modified pages are being sent to the recovery target, pages pointed to by the log cannot be released even if they have been replaced in the database by a newer version. Therefore, the worst case size of the log is still the same as before, the same as the size of the database.

When the recovery target starts executing transactions, many update requests can be pending in the update queue. In order to obtain a consistent database, the recovery target must not miss any replicated update requests. Therefore, the update queue must never overflow. The size of the update queue should be a function of the time it takes for the recovery target to resume transaction processing, the arrival rate of update requests, and their size.

For guaranteeing durability for locally committed transactions without preventing the system from attaining timeliness, the buddy system requires dedicated network resources. Furthermore, the communication between a node and its buddy needs to be sufficiently fast for deadlines to be met. In contrast, the non-volatile buffer approach requires non-volatile memory. Under this approach, the size of the non-volatile buffer must be decided. If the buffer were to fill up, transaction execution would have to be suspended, and transactions might miss their deadlines. The size of the non-volatile buffer is a function of the frequency of update transactions, the size

of update requests, and how fast update requests can be dropped from the buffer.

5.4 Feasibility of Implementation

As stated in section 5.1, implementing the mechanism described in chapter 4 should not be problematic. The mechanism is a feasible option for distributed real-time database systems. As described in section 3.1, eliminating disks from real-time databases is desirable. It is therefore necessary to have a recovery method which does not require disks.

The post-crash part of the mechanism consists of bidding, fuzzy checkpointing and update request forwarding. Fuzzy checkpointing is the most important ingredient in our approach. It is conjectured that an implementation of the post-crash mechanism is feasible, since, as mentioned by Gruenwald et al. [GHD+96], fuzzy checkpointing has been implemented and used in real systems. The key parts of our algorithm and conventional fuzzy checkpointing are the same, namely how the database is checkpointed, logged, and finally rebuilt.

Ramamritham et al. [RSZ89] present a simulation of a distributed scheduling mechanism which uses a bidding approach similar to the one used in this work. Their implementation suggests that it is feasible to implement a bidding scheme.

An implementation of update request forwarding should be based on the existing replication mechanism, in which each node replicates update requests to the entire system. During forwarding, all the recovery source has to do is to send certain received update requests on to the recovery target.

The buddy system, as described in chapter 4, can be implemented using standard hardware and relatively small modifications to the transaction model. For example,

a predictable connection can be created between buddies by using two serial links, one for each direction. As stated in section 4.1, sending update requests to a buddy during commit is similar to flushing log records to disk under the force-log-at-commit rule [GR93, pp 557].

When the buddy system is used, it is possible to set a bound on the time it takes to replicate update requests. A buddy system can therefore be used in a system implementing bounded replication as well as ASAP replication. By implementing buddy groups of more than two nodes, it is possible to use a buddy system even if the single-node failure assumption is dropped. However, the buddy system has problems with the case when the entire system crashes. When all the nodes in a distributed database system crash, the database has to exist on disk. This means that under the buddy system it is necessary to either hold multiple disk copies of the database, or have a single node which has a disk and is included in every buddy group.

Using a non-volatile memory buffer to guarantee durability is common in main-memory database systems, see e.g. Eich [Eic87]. In our approach, update requests are written to non-volatile memory and stored there until they have been replicated. This is similar to the approach described by Eich, where log records are written to non-volatile memory and later flushed to disk. Since non-volatile memory is a common approach to guaranteeing durability in main-memory database systems, we believe that our approach is a viable option for implementation. However, as stated in section 4.3, before the non-volatile buffer approach can be used in a real-time system, it has to be examined whether the size of the buffer is bounded. In order to avoid disruptions to transaction processing, it is important that the buffer never fills up.

If the non-volatile buffer approach is used in a system with bounded replication, the down-time of a crashed node has to be bounded. This is because update requests

present in a non-volatile buffer at crash time will not be replicated until the crashed node restarts. Since it may be difficult to bound the down-time, a buddy system seems to be a better choice when bounded replication is used.

5.5 Related Work

We have designed a mechanism for diskless recovery in fully replicated, distributed real-time main-memory databases. In our case, there are no predetermined recovery sources, instead the recovering node must start by selecting a recovery source. We have been unable to find any previous work which addresses this type of approach to recovery.

In our work, we need to make a copy of an entire main-memory resident database. This is exactly what is done by main-memory database checkpointing algorithms. There are, however, differences between our approach and that taken for main-memory databases. In main-memory database recovery it is usually assumed that a disk is used for logging and checkpointing, and the solutions tend to be purely centralized, see e.g. [Hag86, Eic86, HK98, HR98, JSS98, DLL98]. In these approaches, the checkpointing algorithm runs periodically and writes a copy of the database, the checkpoint, to disk. After the system crashes and restarts, the checkpoint is used, along with a log, to recreate the latest consistent database state which existed before the system crashed. In contrast, in our approach the checkpointing algorithm is only executed by a recovery source after a copy of the database is requested by a recovery target. So, while checkpointing is traditionally a pre-crash activity, in this work, checkpointing is only done after a crash has occurred. Also, we never write the checkpoint to disk, instead it is sent from the recovery source to the recovery target.

Gruenwald et al. [GHD+96] classify checkpointing approaches for main-memory databases as fuzzy, non-fuzzy, and log-driven. Fuzzy checkpointers do not lock data items while they are checkpointed. Therefore, partial transaction and action executions can be reflected in the checkpoint. Non-fuzzy approaches on the other hand, lock data objects while they are checkpointed. This is done in order to ensure transaction- or action-consistency. When a checkpoint is transaction-consistent, no partial transaction executions are visible in the checkpoint, and similarly, no partial updates are reflected in an action-consistent checkpoint. Gruenwald et al. state that non-fuzzy checkpoints incur a high overhead during normal transaction processing since data contention is increased. Log-driven checkpointing applies the log to an existing disk-based copy of the database, instead of dumping pages directly from memory to disk. Since the recovery target starts out with an empty database in our approach this method is not applicable.[1]

Since we want the recovery source to be able to guarantee critical deadlines while supplying the database to the recovery target, it is desirable for the checkpointing process to disturb other transaction processing as little as possible. For this reason, we have chosen to build on fuzzy checkpointing rather than a non-fuzzy checkpointing approach.

Fuzzy checkpointing was first described as a checkpointing approach for main-memory databases by Hagmann [Hag86]. Since then, fuzzy checkpointing has been developed further. Two interesting developments of fuzzy checkpointing are described by Dunham et.al. [DLL98]. These approaches both divide the database into sections which are checkpointed independently in order to reduce the amount of log information which has to be processed after restart. Fuzzy checkpointing has also been

[1] Applying the entire log to an empty database could eventually bring the database up-to-date, but this would be extremely inefficient.

implemented in main-memory database systems. One example is the Dali system. Originally the Dali system used action-consistent checkpoints, but due to the inefficiency of that approach, fuzzy checkpoints were later implemented in the system [RBP+98, GHD+96].

As far as distributed recovery is concerned, we have not found any work which attempts to do diskless recovery by copying the database from an arbitrary recovery source. Treiber and Burkes [TB95] describe an approach based on a leader/follower model, which uses the current leader to assist in recovery when a node restarts. This approach, however, uses disks and takes advantage of node pairs, i.e. a recovering node gets help from its leader. When the leader crashes, the follower takes over as the new leader.

A recovery mechanism for centralized real-time databases has been suggested by Song et al. [SKR+99]. In this approach, a main-memory database using shadow-paging is assumed. All shadow pages are placed in non-volatile memory and are flushed to disk when they have been updated. No traditional logging or checkpointing is needed since an up-to-date consistent copy of the database can be constructed using the shadow-pages in non-volatile memory and the database copy on disk. It is claimed that the low overhead during normal processing makes this approach more suitable to real-time main-memory database systems than existing recovery mechanisms.

The use of non-volatile memory in main-memory databases is described by Eich [Eic87]. Eich states that the use of non-volatile memory is a very important factor in designing an efficient main-memory database recovery mechanism. In our case, non-volatile memory is used for update requests, instead of a log tail, but the experiences of, e.g., Eich show that using non-volatile buffers to guarantee durability is a realistic approach. Further, Copeland et al. [CKKS98] have found that the use

of battery-backed-up RAM to implement non-volatile memory is a cost-effective and viable approach.

Chapter 6

Conclusions

The goal of this work is to design a recovery mechanism suitable for distributed real-time database systems. We described the problem in detail in chapter 3, and suggested a solution in chapter 4. The problem was divided into two problems: how a restarted node can obtain a consistent database copy, and how durability can be guaranteed for locally committed transactions. The solution to the first problem is based on fuzzy checkpointing, a method which is popular in main-memory database systems. Two possible solutions were suggested for the second problem: a buddy system, and a non-volatile buffer. In chapter 5 we presented an evaluation of the recovery mechanism, the evaluation indicates that the mechanism is applicable to real systems.

In section 6.1, we give an overview of the distributed recovery problem. The solution and evaluation are also briefly described. The contributions to the real-time database community are discussed in section 6.2. Finally, in section 6.3, we identify possible future research directions for recovery in distributed real-time database systems.

6.1 Summary

The goal of this work is to design a recovery mechanism for distributed real-time database systems which does not require disks. Real-time systems are sometimes used in extreme environments which limit the types of hardware which can be used. An example of this are environments which expose the system to vibrations, such as missiles and some environments including heavy machinery. In such environments, it is desirable to avoid disks, since they do not tolerate vibrations well. Also, reducing the amount of hardware needed in a real-time system can help to reduce their cost, especially in mass-produced products. As indicated by Obermarck [Obe98], database recovery has been researched at least since the 1960s. In spite of this, we have been unable to find any work on recovery in distributed real-time database systems.

We assume a distributed real-time database system in which every node holds a complete copy of the database in main-memory, i.e. the database is fully replicated and main-memory resident. Furthermore, we assume that each node can locally execute and commit transactions, after which the transaction updates are replicated to the entire system. The fact that each node holds the database entirely in main-memory means that when a node crashes it looses the entire database contents. During recovery, we want to utilize that the database is fully replicated, by letting the recovery target obtain the database from a healthy node. The healthy node should be able to execute transactions and guarantee timeliness for critical transactions while recovery is in progress.

Doing recovery in the way described earlier poses two major problems. First, it needs to be examined how to copy the database from the recovery source to the recovery target without preventing timely transaction execution by the recovery source. Second, we have to consider how durability can be guaranteed for locally committed

transactions.

In order to solve the first problem, copying the database from the recovery source to the recovery target, we have suggested a method based on fuzzy checkpointing. The recovery source makes a single sweep through the database and sends every database page to the recovery target, i.e. the recovery source sends a fuzzy database checkpoint to the recovery target. This is done without regard to concurrency mechanisms and, therefore, the checkpoint received by the recovery target can be inconsistent. Therefore, it is necessary for the recovery source to log all database updates during checkpointing, and then send the log to the recovery target after the entire database has been sent. The recovery target must then apply the log to the fuzzy checkpoint in order to obtain a consistent database view. To make sure that the recovery target does not miss any updates being replicated in the system, the recovery source has to forward all replicated update requests to the recovery target until it is guaranteed that the recovery target will not miss any updates.

Two possible solutions have been proposed to the second problem: guaranteeing durability for locally committed transactions. The first one is based on the assumption that a crashed node will have time to recover before any other node crashes, i.e. the assumption of single-node failures. Under this assumption, it is sufficient for two nodes to know about the effects of a transaction in order to make it durable. Therefore, we have suggested a buddy system, in which nodes are organized in pairs. During the commit phase of a transaction, the executing node tells its buddy about all the updates that the transaction is about to make. This is similar to the force-log-at-commit rule, only instead of writing log data to disk during commit, update information is sent to the buddy node.

The other approach, to guarantee durability of locally committed transactions, is

to write the update information to a non-volatile memory buffer instead of sending it to a buddy. Each node has to have a non-volatile memory buffer which can be used to hold update requests which have not been replicated. When a node restarts and finds unreplicated update requests in its non-volatile memory, it starts by replicating these and then proceeds to obtain a copy of the database.

The suggested mechanism is suitable to distributed real-time database systems. It neither requires disks nor prevents nodes other than the recovery target from guaranteeing transaction timeliness. It is, however, necessary for the system to be designed with the possible increased load from recovery in mind. Recovery both increases load on the network and also the load on the recovery source's processor. When a buddy system is used, it is necessary to dedicate network resources to it so that it can be made predictable.

Implementing the mechanism in a real system should be possible. The process of copying a database from a recovery source to a recovery target is based on fuzzy checkpointing, a well known and tested method which has been implemented in main-memory database systems [GHD+96]. Using a non-volatile buffer to guarantee transaction durability is also a technique well known in the main-memory database community. The buddy system approach is based on copying update information to another node instead of to non-volatile memory. This should not be too difficult to implement, but it relies on the fact that network resources can be dedicated to the buddy system.

6.2 Contributions

This work has provided the real-time database community with a method for recovery in distributed real-time database systems. We have created a high-level design of a recovery mechanism, largely based on existing, tested methods.

The contributions of this work are:

- We have discussed the need for a new recovery mechanism for distributed real-time database systems. During this discussion we argued that existing database recovery mechanisms are not suitable for distributed real-time systems. Earlier recovery approaches have relied on disks, but in real-time systems it can be beneficial to avoid disks, for example due to environmental and financial reasons. The need for new recovery mechanisms is the motivation for this work.

- Issues in diskless distributed recovery have been identified. The two major problems are how a database can be rebuilt after a crash, and how durability can be guaranteed for locally committed transactions. Identifying these issues has made it possible to design a diskless distributed recovery mechanism.

- Existing algorithms have been applied to a new problem. We have used bidding and fuzzy checkpointing to construct a method of rebuilding a database copy at a restarted node. In our approach, fuzzy checkpointing is a post-crash activity, in contrast to previous work. Using bidding and fuzzy checkpointing has enabled us to design a recovery mechanism which disturbs other database processing minimally.

- We have shown that using fuzzy checkpointing combined with physical logging does guarantee that the recovery target receives a consistent database copy as

long as the recovery source database is consistent. Considering log operations and possible database, log, and database states it has been shown that after the log has been applied to the received checkpoint the recovery source has a consistent database.

- We have suggested two methods of guaranteeing durability for locally committed transactions in a diskless environment. Firstly, a buddy system in which more than one node always knows about updates which are about to be committed and, secondly, using a non-volatile memory buffer to store committed updates. The non-volatile buffer approach is well known from main-memory databases. Using one of these methods makes it possible to avoid both disks and distributed commit in diskless distributed environments.

- The suggested recovery mechanism has been evaluated with respect to timeliness, resource requirements, and feasibility of implementation. All nodes other than the recovery source can guarantee timeliness of critical transactions during recovery. During recovery, the log can become as large as the database at the beginning of checkpointing plus any pages that are added to the database until the end of checkpointing. We have argued that an implementation of the suggested mechanism is feasible.

6.3 Future Work

In this section, we identify three possible directions for future research in recovery for distributed real-time database systems. Firstly, it may be interesting to investigate the possibility of doing incremental recovery and, thus, allowing the recovery target to start transaction processing before it has obtained the entire database. Secondly,

relaxing the assumptions made in this work may be beneficial when the mechanism is to be implemented in a real system. Finally, formally investigating the requirements of the algorithms presented here is important before they are included in an implementation.

In some cases, it may be desirable for a recovering node to start executing transactions as soon as possible after restart. One way to do this is to allow the node to start transaction processing before it has a complete copy of the database, this is called *incremental recovery*. Incremental recovery for main-memory database systems has been described by for example Levy and Silberschatz [LS92].

Segmenting the database is one way to enable incremental recovery. The database is divided into segments which can be recovered independently. When a database has been segmented it is possible to take a fuzzy checkpoint of each segment independently. Thus, it may be possible to recover the most critical parts of the database first, start transaction processing, and then recover the rest of the database. This will not reduce the overall time required for recovery (more time may actually be required), but the recovery source could start executing transactions on parts of the database as each segment is recovered. Since the entire database and the log have to be transmitted it is probably not possible to reduce the overall time needed for recovery.

Another way to enable incremental recovery is to take a higher level view of recovery. Instead of copying the database page-by-page, the database can be copied database object-by-database object. This would allow transactions to be executed on those objects which have been recovered even if the database is incomplete. The problem with this approach is that the objects have to be locked when they are copied, and thus data contention increases.

Relaxing the assumptions made in this work can be an interesting task. In some cases, e.g. systems with a large number of nodes, it may be desirable to relax the single-node failure assumption. In a large system, with tens or hundreds of nodes, the possibility of multiple node failures increases. As long as we assume that all the nodes in a system do not crash at the same time, we should be able to deal with the situation by making some fairly minor changes to our mechanism. In fact, it is only necessary to add code to protect from the case where the recovery source or the recovery target crash during recovery. The algorithms presented in chapter 4 work for multiple failures as long as we assume that neither the recovery source nor the recovery target crash during recovery.

However, in order to cope with the case when the entire system crashes, some type of non-volatile storage must be used. A complete copy of the database must exists on disk or some other type of non-volatile storage. This might be solved by adding dedicated *recovery nodes* to the system, i.e. nodes whose sole purpose it is to keep a stable copy of the database and help other nodes recover after a systemwide crash. Two problems must however be solved before this can be implemented. First, the dedicated recovery node(s) might crash some time before the rest of the system, all changes made after the crash of the recovery node(s) will then be lost. Second, it must be solved how durability can be guaranteed for locally committed transactions under the assumption that a total system crash can occur. The recovery node(s) would have to receive all update requests a transaction makes during the transaction's commit phase.

When the database becomes large, it may be desirable to make it virtually fully replicated, instead of fully replicated [AHE+96]. That is, assume that each node holds all parts of the database it will ever need, which may be less than the entire database.

This is not supported by our recovery mechanism, but it would be interesting to investigate how best to support virtually fully replicated databases.

We have assumed a distributed system of homogeneous nodes. It would be interesting to try to relax this assumption. For example, is is possible to modify the suggested mechanism to work in an environment were nodes are of different architectures with different internal data representations.

In this work, we have assumed that durability should always be guaranteed for all transactions. It may, however, be possible to relax this assumption in certain types of systems or nodes. Consider, for example, a node which records atmosphere temperature readings every 10 minutes. If this node crashes and is unable to record any readings for 2 hours, it is probably not essential that the last reading made before the crash survives it. The only difference would be whether 12 or 13 readings were missing. Also, a node like this could probably give estimates of the missing values when asked by extrapolating from values around the gap. Of course, this relaxed approach to durability cannot always be used. For example, if an operator makes some adjustments to a system and gets notification that the adjustments were successful, they should not disappear in a crash.

Finally, it is necessary to investigate the hardware requirements of our algorithms before they can be used in a real system. That is, how much processor load does recovery incur on the recovery source and how heavily does recovery load the network. Also, if the non-volatile buffer approach is taken to guarantee durability, it must be investigated whether the buffer size is bounded for critical transactions.

Bibliography

[AHE+96] S. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS towards a distributed and active real-time database system. *Special Issue on Real Time Data Base Systems, SIGMOD Record*, 25(1), March 1996.

[BH95] M. Berndtsson and J. Hansson, editors. *Proceedings of the First International Workshop on Active and Real-Time Databases (ARTDB-95)*, Workshops in Computing. Springer-Verlag, June 1995.

[BLS97] A. Bestavros, K-J Lin, and S. H. Son. *Real-Time Database Systems: Issues and Applications.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1997.

[BW96] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, second edition, 1996.

[CBS98] T. Connolly, C. Begg, and A. Strachan. *Database Systems: A Practical Approach to Design, Implementation, and Management.* Addison-Wesley, second edition, 1998.

[CKKS98] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe RAM. In Kumar and Hsu [KH98], chapter 19, pages 528–573.

[DLL98] M. H. Dunham, J-L Lin, and Xi Li. Fuzzy checkpointing alternatives for main-memory databases. In Kumar and Hsu [KH98], chapter 21, pages 574–616.

[Eic86] M. Eich. Main memory database recovery. In *1986 Proceedings ACM-IEEE Fall Joint Computer Conference*, pages 1226–1232. IEEE, 1986.

[Eic87] M. Eich. A classification and comparison of main memory database recovery techniques. In *Proceedings of the Third International Conference on Data Engineering*, pages 332–339. IEEE, 1987.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., second edition, 1994.

[GHD+96] L. Gruenwald, J. Huang, M. H. Dunham, J-L Lin, and A. C. Peltier. Recovery in main memory databases. *Engineering Intelligent Systems*, 4(3):177–184, 1996.

[GMH95] H. Garcia-Molina and M. Hsu. Distributed databases. In W. Kim, editor, *Modern Database Systems*, chapter 23, pages 477–493. ACM Press, 1995.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[Hag86] R. B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–843, 1986.

[HK98] M. Hsu and V. Kumar. Introduction to database recovery. In Kumar and Hsu [KH98], chapter 2, pages 6–15.

[HR98] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. In Kumar and Hsu [KH98], chapter 3, pages 16–55.

[JSS98] H. V. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In Kumar and Hsu [KH98], chapter 20, pages 548–573.

[KH98] V. Kumar and M. Hsu, editors. *Recovery Mechanisms in Database Systems*. Prentice Hall PTR, 1998.

[KV93] H. Kopetz and P. Verissimo. Real time and dependability concepts. In Mullender [Mul93], chapter 16, pages 411–446.

[LD96] J-L Lin and M. H. Dunham. Segmented fuzzy checkpointing for main memory databases. In *Proceedings of the ACM Symposium on Applied Computing*, pages 158–165. ACM Press, 1996.

[LDN97] J-L Lin, M. H. Dunham, and M. A. Nascimento. A survey of distributed database checkpointing. *Distributed and Parallel Databases*, 5(3):289–319, 1997.

[LS92] E. Levy and A. Silberschatz. Incremental recovery in main memory database systems. Technical Report TR-92-01, Department of Computer Sciences, University of Texas at Austin, 1992.

[Lun97] J. Lundström. A conflict detection and resolution mechanism for bounded-delay replication. Master's thesis, University of Skövde, 1997. Report no: HS-IDA-MD-97-10.

[Mel98] J. Mellin. Predictable event monitoring. Licentiate thesis, Linköping University, 1998. Thesis no: 737.

[Mul93] S. Mullender, editor. *Distributed Systems.* Addison-Wesley, second edition, 1993.

[Obe98] R. Obermarck. IMS/360 and IMS/VS recovery: Historical recollections. In V. Kumar and M. Hsu, editors, *Recovery Mechanisms in Database Systems*, chapter 1, pages 1–5. Prentice Hall PTR, 1998.

[RBP+98] R. Rastogi, P. Bohannon, J. Parker, S. Seshadri, A. Silberschatz, and S. Sudarshan. Distributed multi-level recovery in main-memory databases. *Distributed and Parallel Databases*, 6(1):41–72, 1998.

[RLKL95] B. Randell, J-C Laprie, H. Kopetz, and B. Littlewood. *Predictably Dependable Computing Systems.* Springer-Verlag, 1995.

[RSZ89] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38(8):1110–1123, August 1989.

[RTD88] Special issue on real-time data-base systems. SIGMOD Record 17(1), March 1988.

[Sch93] M. D. Schroeder. A state-of-the-art distributed system: Computing with BOB. In Mullender [Mul93], chapter 1, pages 1–16.

[SKR+99] E-M Song, Y-K Kim, C. Ryu, Y-K Kim, S i Jin, and W. Choi. No-log recovery mechanism using stable memory for real-time main memory database systems. To be presented at the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA99), Hong Kong, 13-15 December 1999.

[SSH99] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time database systems. *IEEE Computer*, pages 29–36, June 1999.

[TB95] R. K. Treiber and D. L. Burkes. Remote site recovery for transaction processing. Research Report RJ 9987 (89075), IBM Research Division, Almaden, 1995.

[VK93] P. Verissimo and H. Kopetz. Design of distributed real-time systems. In Mullender [Mul93], chapter 19, pages 511–530.

List of Figures

1	A simplified database system environment	8
2	State transition diagram for transaction execution	11
3	Data redundancy used in recovery	28
4	Pre- and post-crash activities	33
5	Data sent from the recovery source to the recovery target	39
6	Recovery source bid	42
7	How transactions can relate to checkpoints	52
8	Replicated update requests vs. checkpointing interval	55
9	Transaction commit in a buddy environment	58
10	A node crashes before replicating transaction update requests	59
11	Buddy crashes during commit	60
12	Buddy crashes after receiving update requests	60
13	Buddy system model	61